Serielle Transformationen von XML
Probleme, Methoden, Lösungen

Dissertation

zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften (doctor rerum naturalium)
im Fach Informatik

eingereicht an der

Mathematisch-Naturwissenschaftlichen Fakultät II
der Humboldt-Universität zu Berlin

von

Diplom-Informatiker Oliver Becker
geboren am 29. September 1971 in Bleicherode

Präsident der Humboldt-Universität zu Berlin
Prof. Dr. Jürgen Mlynek

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät II
Prof. Dr. Uwe Küchler

Gutachter / Gutachterin:

1. Prof. Dr. Joachim Fischer
2. Prof. Dr. Robert Tolksdorf
3. Prof. Dr. Nikolaus Klever

Tag der mündlichen Prüfung: 26. November 2004

Zusammenfassung

Die Auszeichnungssprache XML definiert eine einfache Syntax für strukturierte Daten, die sich so applikationsübergreifend einsetzen lassen. Eine der wichtigsten Voraussetzungen für den Austausch solcher XML-Daten ist die Möglichkeit ihrer Transformation.

Unter den derzeit verfügbaren Transformationsmethoden für XML hat die Sprache XSLT als W3C-Standard die größte Verbreitung gefunden. Allerdings skaliert XSLT nicht für große Datenmengen, da hier eine Gesamtsicht auf das XML-Dokument vorausgesetzt wird. Andere existierende Transformationsmethoden besitzen entweder die gleiche Eigenschaft oder erfordern die Programmierung auf der XML-fernen Ebene einer Programmiersprache.

In dieser Arbeit wird mit STX eine Transformationssprache für XML entwickelt, die diese Lücke füllt. STX orientiert sich sehr stark an XSLT, verarbeitet jedoch ein XML-Dokument als Datenstrom. STX kann so prinzipiell beliebig große Dokumente transformieren. Die aus der Sprache XPath 2.0 des W3C abgeleitete STX-Pfadsprache (STXPath) trägt dabei der eingeschränkten Sicht auf die zu transformierenden Daten Rechnung, indem sie nur den Zugriff auf die Vorfahren des jeweiligen Kontextknotens ermöglicht.

Zu den neuartigen Konzepten in STX zählen neben prozeduralen Eigenschaften vor allem Gruppen, Schnittstellen zu externen Transformationsprozessen, die komplexe Transformation von Zeichenketten sowie Sprachmittel zur Fehlerbehandlung.

Diese Arbeit stellt Entwurfsmuster für die wichtigsten Transformationstypen in STX vor und demonstriert an drei Fallbeispielen den Einsatz in realen Projekten. Der dazu verwendete STX-Prozessor *Joost* verfügt zudem über standardisierte Java-Schnittstellen, die dessen Integration in bestehende Java-Applikationen erleichtern.

Abstract

The markup language XML defines a simple syntax for structured data that can be used across application boundaries. One of the most important prerequisites for the interchange of such XML data is the possibility of its transformation.

Among the currently available transformation approaches for XML, the W3C standard XSLT has gained the biggest popularity. However, XSLT doesn't scale for huge amounts of data because it requires an overall view to an XML document. Other existing transformation approaches either have the same character or require low-level programming using a general programming language.

This PhD thesis introduces STX, an XML transformation language that fills this gap. STX is strongly geared to XSLT, though it processes an XML document as a stream. Therefore, STX is able to transform documents of any size. The STX path language (STXPath), derived from the W3C standard XPath 2.0, considers the restricted view to the input data and enables the access only to the ancestors of the current context node.

The new concepts in STX include besides its procedural behaviour mainly groups, interfaces to external transformation processes, complex transformations of strings, as well as language means for error handling.

This work introduces design patterns for the most important transformation types in STX and demonstrates three real-life scenarios. The STX processor *Joost* used for this purpose provides in addition standardized Java interfaces that facilitate its integration into existing Java applications.

Vorwort

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Institut für Informatik an der Humboldt-Universität zu Berlin. Diese Tätigkeit begann im Herbst 1998, nur wenige Monate nach Erscheinen der XML-1.0-Spezifikation. Die intensive Beschäftigung mit XML in der universitären Lehre und in Forschungsprojekten mit Industriepartnern hat sich im Laufe der Zeit fast zu einem Hobby entwickelt. Hier hat sich insbesondere der Austausch mit anderen XML-Entwicklern auf Mailinglisten und Konferenzen als äußerst inspirierend erwiesen.

Die Idee zur Entwicklung einer weiteren Transformationssprache für XML entstand auf einer Mailingliste, der Liste xml-dev@lists.xml.org. Hervorzuheben ist hier die Arbeit von Petr Cimprich, auf dessen Initiative das STX-Projekt im Februar 2002 gegründet wurde und der sich seitdem kontinuierlich um den Fortschritt der STX-Spezifikation verdient gemacht hat. Viele wichtige Ideen entsprangen regen Diskussionen unter den Mitgliedern der neu gebildeten STX-Mailingliste. Hier sind vor allem Christian Nentwich, Honza Jiroušek, Manos Batsis, Paul Brown und Michael Kay zu nennen. Ohne deren Mitwirkung am Entwurf von STX wäre diese Arbeit nicht möglich gewesen. Danken möchte ich ebenfalls Anatolij Zubow, dessen Engagement bei der Implementierung wichtiger Teile des STX-Prozessors *Joost* von unschätzbarem Wert war und der alle Designentscheidungen immer wieder kritisch hinterfragte.

Schließlich danke ich Prof. Joachim Fischer, Prof. Nikolaus Klever, Prof. Christoph Polze sowie Prof. Robert Tolksdorf für die konstruktiven Anregungen und die wohlwollende Betreuung.

Berlin im Juni 2004

In Erinnerung an ξ
(1971-2003)

»*I've come across (while reading a document on WAP) a term – "XML Middleware". What does it mean?*«

Middleware is a kind of American stretchable undergarment, for restraining bellies and all the big fat bottoms. For obvious reasons it does not come in a size S (small): the little man doesn't really need middleware. So it only needs to fit sizes X, M, and L.

In Canada, where there are strict laws prohibiting one thing and enforcing the other, they have to have S, and also G (Grande) instead of the American X. Probably because of the cold, the rich mix of cultures, etc. etc., this SGML middleware is rather more elaborate than the XML middleware.

Because it is based on ISO standards, XML middleware can be enjoyed by Europeans and people from many different countries too, though in some Asian countries there are attempts to relax the design, since the original is often felt to be too loose for some types and to restricting for others. Pundits complain that XML middleware is just a silly fashion, because of its Draconian design: if anything gets out-of-place the whole thing falls apart (often with the characteristic sound "WAP" so amusing to children).

I hope this helps.

Rick Jelliffe

Frage und Antwort auf der Mailingliste xml-dev@lists.xml.org
(http://lists.xml.org/archives/xml-dev/200008/msg00489.html)

Inhalt

Kapitel 1
Einleitung

Im Februar des Jahres 1998 erblickte XML, die Extensible Markup Language, das Licht der Welt. Diese Spezifikation ist aus heutiger Sicht zur Standardtechnologie für Datendarstellung und -austausch geworden.

Charakterisierung von XML

Dies war damals kaum vorhersehbar. Zu den wichtigsten Gründen für diesen Erfolg zählen:

1. **XML hat mächtige Vorfahren.**
   XML ist kompatibel mit SGML, der Standardized General Markup Language. Die Wurzeln von SGML reichen bis in die 60er Jahre des 20. Jahrhunderts zurück. Im Laufe der Zeit wurden einige spezialisierte Softwareapplikationen für SGML entwickelt, die heute ebenso auf XML-Daten eingesetzt werden können. Der Durchbruch gelang mit der Sprache des World Wide Web – HTML (Hypertext Markup Language), da mit ihr erstmalig eine konkrete SGML-Anwendung inklusive passender Werkzeuge (HTML-Browser und -Editoren) für jeden Internet-Nutzer zur Verfügung standen.
2. **XML ist einfach.**
   XML wurde mit dem Ziel geschaffen, SGML zu vereinfachen und die durch HTML gesetzten Grenzen zu überwinden. Bereits mit geringen Kenntnissen und einer kurzen Einarbeitungszeit lässt sich ein korrektes XML-Dokument erstellen. XML erfordert weder eine steile Lernkurve, noch ist eine hohe Anfangsschwelle zu überwinden.

   Mit XML lassen sich strukturierte Daten erfassen, ohne dass dazu eine vorherige formale Definition dieser Struktur in Form eines Schemas zwingend erforderlich ist.
3. **XML unterstützt semistrukturierte Daten.**
   In XML lassen sich ebenfalls sehr einfach Daten beschreiben, die nur eine lockere und unregelmäßige Struktur besitzen. XML kann auf diese Weise flexibel für eine große Bandbreite von Anwendungsfällen eingesetzt werden.
4. **XML ist unabhängig.**
   Diese Unabhängigkeit von XML lässt sich in vielerlei Hinsicht feststellen. Die XML-Spezifikation ist frei verfügbar. Es fallen keinerlei Lizenzgebühren für ihre Nutzung an: XML ist *herstellerunabhängig*. Darüber hinaus ist XML weder an eine bestimmte Hardware, noch an ein Betriebssystem, eine Programmiersprache oder einen bestimmten Zeichensatz gebunden: XML ist *plattformunabhängig*.
5. **Es existieren viele Werkzeuge für XML.**
   Die einfache Struktur von XML führte zu einer Vielzahl von Programmen, die XML-Daten verarbeiten können. Die Entwicklungen umfassen allgemeine Werkzeuge wie Parser oder Editoren, Bibliotheken für spezielle XML-Techniken

wie XML-Schemata oder XLinks sowie große Geschäftsanwendungen wie XML-Datenbanken oder Applikationsserver.[1]

6. **Die Entwicklung der Hardware geht schnell voran.**
   Obwohl in XML repräsentierte Daten in der Regel mehr Speicherplatz als andere Datenformate benötigen und dementsprechend ein größerer Aufwand für die Übertragung und Verarbeitung solcher Daten erforderlich ist, werden diese vermeintlichen Nachteile hardwareseitig durch die enorme Entwicklung bei Speichermedien, Bandbreiten und Prozessoren schnell kompensiert.

   Es hat sich gezeigt, dass spezialisierte, für eine bestimmte Architektur oder Domäne optimierte Datenformate zwar in der Anwendung effizienter sein können, diese jedoch einen erheblich höheren Aufwand und damit Kosten für die Entwicklung und Wartung geeigneter Werkzeuge erfordern.

Letztendlich lassen sich die besonderen Eigenschaften von XML auf zwei Punkte reduzieren:

- XML ist standardisiert genug, sodass eine Vielzahl von vorhandenen Werkzeugen zur Verarbeitung von XML-Daten eingesetzt werden kann. Der Austausch von Daten im XML-Format gestaltet sich unproblematisch.
- XML ist flexibel genug, sodass jeder Anwenderkreis eine »eigene« XML-Sprache definieren und benutzen kann. Für jede Anwendung lässt sich auf diese Weise ein eigenes, auf den jeweiligen Zweck spezialisiertes Format definieren.

Eine konkrete XML-Sprache wird im folgenden auch als *Dokumenttyp* bezeichnet, die in dieser Sprache erlaubten Begriffe bilden das *Vokabular*. Eine genauere Begriffsbestimmung sowie die in XML verfügbaren Mittel zur Festlegung eines Vokabulars werden in Kapitel 2 dargestellt.

XML als Austauschformat

Die Unabhängigkeit von XML macht es im Besonderen als Austauschformat interessant. Unterschiedliche Applikationen, zwischen denen Daten ausgetauscht werden sollen, können dies sehr gut über eine XML-Schnittstelle erreichen. XML kann in dieser Hinsicht als standardisierter größter gemeinsamer Nenner für alle Software-Anwendungen verstanden werden: *XML ist das ASCII des 21. Jahrhunderts*.

Insbesondere ist XML das Format der Wahl, wenn Daten der Öffentlichkeit zur Verfügung gestellt werden sollen. In diesem Fall findet ein Datenaustausch statt, bei dem über den Empfänger jedoch nichts bekannt ist. Die Bereitstellung spezieller, an eine bestimmte Software gebundener Datenformate, würde dagegen immer nur für einen Teil der Öffentlichkeit nutzbar sein.

XML-Datenbestände

Die folgende Liste enthält zwei Beispiele für Datenbestände, die im XML-Format im WWW veröffentlicht wurden.

- **Open Directory Project (ODP)**
  Das ODP ist ein Open-Source-Web-Verzeichnis, in dem Web-Ressourcen in derzeit über 580.000 Kategorien verwaltet werden, siehe http://www.dmoz.org. Die XML-Repräsentation des aktuellen Datenbestandes benötigt etwa 1,2 GByte.

  Die Verarbeitung von ODP-Daten wird in Kapitel 6.2 beispielhaft gezeigt.

[1] Die Website http://www.xmlsoftware.com/ listet derzeit über 500 Produkte aus verschiedenen Kategorien auf, darunter Parser, Editoren, Transformationswerkzeuge, Präsentationswerkzeuge, etc. Mehr als die Hälfte davon (56 %) können frei benutzt werden, etwa 100 sind sogar als Open-Source verfügbar.

- **UniProt**
  Das UniProt-Konsortium unterhält eine öffentlich zugängliche Datenbank für Proteinsequenzen, siehe http://www.uniprot.org/. Der als XML repräsentierte Datenbestand umfasst etwa 968 MByte.

Mit der wachsenden Verfügbarkeit geeigneter Software für XML-Dokumente dieser Größenordnung ist damit zu rechnen, dass weitere öffentliche Datenbanken ihren Datenbestand zukünftig in XML anbieten.

Transformationen

Die Verständigung auf XML als Basis für den Datenaustausch allein bedeutet noch nicht, dass jede Applikation die in einem beliebigen XML-Vokabular ausgedrückten Daten richtig interpretieren kann. Genauso wie für die menschliche Kommunikation häufig Übersetzungen zwischen verschiedenen natürlichen Sprachen notwendig sind, müssen XML-Daten zwischen verschiedenen Vokabularen transformiert werden.

Ein wichtiger Bestandteil einer XML-basierten Infrastruktur sind daher Transformationskomponenten, die XML-Daten in das jeweils benötigte XML-Vokabular übersetzen. Solche Komponenten müssen sich leicht erstellen und anpassen lassen.

Wie das Kapitel 3 zeigen wird, sind die heute gebräuchlichen XML-Transformationswerkzeuge für Datenmengen der oben genannten Größenordnungen nicht einsetzbar. Dies liegt im Wesentlichen darin begründet, dass der für die Ausführung einer Transformation benötigte Hauptspeicher proportional zur XML-Dokumentgröße anwächst. Eine ressourcenschonende Transformationsalternative kann nicht nur die kostengünstige Entwicklung von Transformationen für umfangreiche XML-Daten ermöglichen, sie würde darüber hinaus eine effizientere XML-Verarbeitung für solche Anwendungen erlauben, die XML als interne Datenrepräsentation verwenden. Insbesondere können auch XML-Middleware-Plattformen[2] davon profitieren, da diese in besonderem Maße XML-Daten zwischen unterschiedlichen Anwendungen austauschen und gegebenenfalls anpassen müssen.

Zielsetzung

Ziel dieser Arbeit ist die daher die Entwicklung einer skalierbaren Transformationssprache für XML.

Skalierbarkeit

Das Thema Skalierbarkeit besitzt dabei auch über den aktuellen XML-Bezug dieser Arbeit hinaus Bedeutung. So sind ganz allgemein bei der Verarbeitung von »großen« Daten solche Methoden von besonderem Interesse, die ein Ergebnis unabhängig von der aktuellen Eingabegröße liefern können. Diese Skalierbarkeit muss dabei unter zwei Gesichtspunkten gesehen werden:

1. Wird ein Ergebnis mit angemessener Speichernutzung geliefert?
2. Wird ein Ergebnis in angemessener Zeit geliefert?

Was hier unter »angemessen« zu verstehen ist, hängt vom konkreten Anwendungsgebiet ab. Speicherkritische Anwendungen benötigen unabhängig von der Datenmenge (der Problemgröße) bei begrenztem Speicher ein Berechnungsergebnis. Zeitkritische

[2] An dieser Stelle soll nun eine seriöse Definition des Begriffes *XML-Middleware* angegeben werden: Als *Middleware* bezeichnet man im Allgemeinen eine Softwareschicht, die in verteilten Systemen alle Aspekte der Verteilung und Kommunikation für die beteiligten Komponenten transparent auf Anwendungsebene abwickelt. Von einer *XML-Middleware* spricht man, wenn eine oder mehrere der beteiligten Komponenten die zu übertragenden Daten als XML benötigen.

Anwendungen benötigen ein solches Berechnungsergebnis in einer vorhersagbaren Zeitspanne.

In dieser Arbeit steht der erste Aspekt im Vordergrund: die zu entwickelnde Sprache soll unabhängig von der Dokumentgröße die Transformation von XML-Daten auf einem begrenzten Speicher ermöglichen. Der dafür benötigte Berechnungsaufwand soll sich allerdings höchstens linear proportional zur Eingabegröße verhalten. Ein höherer Aufwand (polynomiell oder gar exponentiell) würde die neue Transformationssprache in der Praxis kaum anwendbar machen.

Das Erreichen von Skalierbarkeit ist in der Regel mit Einbußen bei der Gesamtsicht auf das zu lösende Problem verbunden. Da das Problem beliebig groß werden kann (hier: die Eingabedaten beliebig groß werden können), kann eine skalierbare Lösung immer nur mit einem Ausschnitt des Gesamtproblems (der Gesamtdaten) arbeiten. Für das Problem bei XML-Transformationssprachen wird in dieser Arbeit eine Lösung entwickelt, die auf einem XML-Datenstrom arbeitet und den aktuell sichtbaren Ausschnitt direkt transformiert.

Ursprünge von STX

Die Grenzen der Standard-Transformationssprache XSLT waren bereits seit ihrer Entstehung im Jahr 1999 bekannt. Angesichts des wachsenden Einsatzes von XML sowohl für die Repräsentation großer Datenmengen als auch für die Datenübertragung innerhalb verteilter Anwendungen begann im Frühjahr 2002 eine Gruppe von XML-Entwicklern auf einer eigenen Mailingliste mit der Entwicklung einer speziellen XML-Transformationssprache. Diese soll eine serielle Verarbeitung der XML-Daten als Datenstrom ermöglichen, sodass unabhängig von der Dokumentgröße stets nur eine begrenzte und vorhersagbare Menge an Arbeitsspeicher benötigt wird.

Neben Petr Cimprich beteiligte sich der Autor der vorliegenden Arbeit federführend an der Entwicklung der daraus entstandenen Spezifikation der Sprache *Streaming Transformations for XML (STX)* [STX][3] und erstellte eine erste prototypische Implementierung in Form des STX-Prozessors *Joost* [Joost], die im Internet veröffentlicht wurde. Diese Dissertationsschrift beruht auf den Ergebnissen dieser Forschungstätigkeit.

Die inspirierenden Diskussionen auf den STX- und *Joost*-Mailinglisten trugen in starkem Maße zum Entstehen dieser Arbeit bei. Daneben gaben Fachvorträge des Autors auf den XML-Konferenzen *XML Europe 2003* in London und *Extreme Markup Languages 2003* in Montréal sowie die dort geführten Fachdiskussionen der Arbeit wichtige Impulse. Die Resonanz der *Joost*-Nutzer beweist die große praktische Relevanz der entstandenen Sprache STX.

Aufbau der Arbeit

Kapitel 2 behandelt die Grundlagen von XML. Neben der XML-Syntax werden insbesondere XML-Transformationen charakterisiert und die für die XML-Verarbeitung notwendige abstrakte Sicht in Form eines XML-Datenmodells vorgestellt.

Das Kapitel 3 diskutiert die derzeit existierenden Transformationsmethoden für XML. Unter ihnen hebt sich die Sprache XSLT als W3C-Standard deutlich heraus. Sie ist das am häufigsten eingesetzte Transformationsmittel. Da sich die in dieser Arbeit entwickelte neue Transformationssprache stark an XSLT orientiert, werden deren grundlegende Eigenschaften in Kapitel 4 vorgestellt.

[3] Literaturverweise sind in dieser Arbeit durch die Angabe eines Kürzels in eckigen Klammern gekennzeichnet. Die dazugehörige Quelle ist im Literaturverzeichnis im Anhang E zu finden.

Den größten Umfang besitzt das Kapitel 5, das der neu entwickelten Sprache *Streaming Transformations for XML (STX)* gewidmet ist. Dieses Kapitel stellt im Detail die STX-Konzepte dar, führt in die Pfadsprache *STXPath* ein und diskutiert einige Lösungsansätze für typische Transformationsaufgaben.

Das Kapitel 6 enthält drei Fallbeispiele für STX-Transformationen. Insbesondere die Verarbeitung der ODP-Daten als auch der Zugriff auf einen Web Service demonstrieren die Praxisrelevanz der entwickelten Sprache.

Das vorletzte Kapitel dieser Arbeit behandelt schließlich verschiedene Aspekte der Integration von STX-Transformationen in Java-Anwendungen. Grundlage dafür ist der in Java geschriebene STX-Prozessor *Joost*, in dem prototypisch die hier vorgestellten Konzepte realisiert wurden.

Die Arbeit schließt in Kapitel 8 mit einem Ausblick auf weitere Forschungsvorhaben. Der Anhang enthält die vollständige STX-Sprachreferenz sowie die Quelltexte der im Kapitel 6 vorgestellten Fallbeispiele.

Terminologie

In der Arbeit wurden, soweit es sinnvoll erschien, englische Fachbegriffe ins Deutsche übersetzt. Jedoch ließ sich nicht für alle Begriffe eine gängige deutsche Entsprechung finden. Insbesondere durfte die Verständlichkeit die Arbeit nicht durch die Verwendung unüblicher Übersetzungen beeinträchtigt werden. In Zweifelsfällen wurde der englische Begriff zusätzlich in Klammern angegeben.

Die folgenden Begriffe wurden im englischen Original belassen und werden in der Arbeit mit den folgenden Artikeln benutzt: das Framework, das Markup, die Middleware, der Parser, das Matching, die Pipeline, das Sheet, der Stack, das Stylesheet, das Template, das Toolkit. Sie werden unveränderlich dekliniert.

Kursiv- und **Fettschrift** dienen allein der Hervorhebung. Sie besitzen keine eigene semantische Bedeutung.

Kapitel 2
XML: Alles, was man wissen muss

Dieses Kapitel enthält eine Einführung in die für das Verständnis der vorliegenden Arbeit wesentlichen XML-Konzepte. Die mit den XML-Feinheiten bereits vertrauten Leser sollten es ohne Weiteres überspringen können.

Kapitel 2.1 motiviert die Verwendung von expliziten Textauszeichnungen. Kapitel 2.2 stellt die Grundkonzepte der XML-Syntax vor. In Kapitel 2.3 werden XML-Transformationen besprochen. Kapitel 2.4 schließlich geht auf die feinen, aber wichtigen Unterschiede zwischen der konkreten Textform von XML und den auf diese Weise beschriebenen Informationen ein.

2.1 Sprache und Schrift

»*Im Anfang war das Wort*«. Menschen bildeten aus Worten Sätze und entwickelten zur gegenseitigen Verständigung Sprachen. Gesprochene Wörter wurden niedergeschrieben, um sie festzuhalten und der Vergänglichkeit des menschlichen Erinnerungsvermögens zu entziehen. Schriftliche Überlieferungen geben uns ein recht zuverlässiges Bild vom Wissensstand zum Zeitpunkt der Niederschrift. Mündliche Überlieferungen hingegen wurden abgewandelt und ausgeschmückt; häufig lässt sich nicht einmal ihr Ursprung mit Sicherheit bestimmen.

Die Vervielfältigung von Schriften war jedoch lange ein teures Unterfangen. Wurden bis ins Mittelalter die Originale mühevoll abgeschrieben (in der Regel war die Kunst des Schreibens ausschließlich innerhalb klösterlicher Mauern bekannt), wurde mit der Erfindung des Buchdrucks durch Johannes Gutenberg um 1450 eine Revolution der Schriftsprache ausgelöst. Zeitungen wurden erst durch Gutenbergs Erfindung ermöglicht. Nicht zuletzt trug Luthers Übersetzung der Bibel ins Deutsche im Jahre 1545 und deren gedruckte Verbreitung entscheidend zur Alphabetisierung und zur Bildung einer einheitlichen deutschen Sprache bei.

Texte waren lange Zeit das einzige Medium, um Wissen festzuhalten. In ihnen materialisierten sich jegliche Informationen, seien es die Bücher der Bibel, kurzlebige Zeitungsmeldungen oder so profane Dinge wie Werbeplakate oder Einkaufszettel. Seit der Erfindung von Radio und Fernsehen als neue Massenmedien und dem Siegeszug des Internets gewinnen andere Kommunikationsformen heute zunehmend an Bedeutung. Nichtsdestoweniger werden Texte als Grundform des Informationsaustauschs wohl immer Teil der menschlichen Kommunikation bleiben, nicht zuletzt weil zum Erstellen und zum Lesen eines Textes kaum zusätzliche technische Hilfsmittel erforderlich sind, was etwa für Videos schon nicht mehr gilt.

Ein Text ist nicht allein die schriftliche Repräsentation der Sprache in Worten und Sätzen. Texte sind gegliedert, sie besitzen Struktur. Genauso wie der Informationsgehalt eines Satzes beim Sprechen durch seine Intonation entscheidend beeinflusst wird, gehören Hervorhebungen und Strukturierungen eines schriftlichen Textes zu dessen Informationsgehalt. Überschriften und Fußnoten können vom Leser erkannt und entsprechend gedeutet werden, weil sie mit Hilfe typografischer Konventionen als solche eindeutig gekennzeichnet wurden.

In den meisten Fällen geben Formatierungen jedoch nur zusätzliche Hinweise. *Metainformationen* über einen Text lassen sich dagegen erst durch das *Verstehen* des Textes erhalten. Erst dann lässt sich die Frage beantworten, wovon dieser Text handelt. So erschließt sich aus einem fremdsprachigen Text im Allgemeinen nicht, ob es sich um eine Wegbeschreibung, um ein Kochrezept oder um einen Liebesbrief handelt. Es könnte jedoch wichtig sein, diese Metainformation zu besitzen. Auch wenn ein Tourist kein einziges Wort eines umfangreichen fremdsprachigen Textes versteht, kann es im Notfall äußerst hilfreich sein, genau die Zeilen herauszufinden, die die Adresse des nächsten Arztes enthalten, um sie dem nächsten einheimischen Taxifahrer zeigen zu können.

Für Menschen ist das Verstehen eines Textes eine leichte Übung (die Kenntnis der Sprache und einen verständlichen Text einmal vorausgesetzt). Dies erweist sich jedoch für die Verarbeitung mit Computern als überraschend schwierig. Mit der Computerlinguistik hat sich eine eigene Fachdisziplin entwickelt, die sich ausschließlich mit der Verarbeitung natürlicher Sprachen mit Hilfe des Computers befasst. Doch auch hier gibt es natürliche Grenzen, die durch das Wesen der menschlichen Sprache bedingt sind. Ein Satz wie »Die Frau sah den Mann am Fenster mit dem Fernglas.« kann auf ganz unterschiedliche Weise verstanden werden. Ein Satz jedoch, der keine eindeutige Bedeutung besitzt, kann erst recht niemals durch einen Computer eindeutig interpretiert werden.

Die einfachste Lösung dieses Problems besteht darin, die einem Text innewohnenden Metainformationen explizit anzugeben. Würde man sowohl »den Mann am Fenster« als auch »mit dem Fernglas« als Attribute der Tätigkeit »sah« kennzeichnen, wären alle Mehrdeutigkeiten ausgeräumt. Dieses Anbringen von Metainformationen an einen Text kann mit Hilfe von *Markup* (*Textauszeichnung*) geschehen. Ein auf diese Weise ausgezeichneter Text ist immer noch ein Textdokument, jedoch gehören bestimmte, vom eigentlichen Inhalt unterscheidbare »Worte« nun zu den Metainformationen.

Markup-Sprachen[1] lassen sich auf verschiedene Weise bilden. Die angebrachten Metainformationen können vorwiegend auf Darstellungseffekte ausgerichtet sein (wie beispielsweise bei troff, TEX oder HTML) oder sich auf inhaltliche Aspekte beziehen. Im zweiten Fall spricht man von *generischem Markup*, d.h. Auszeichnungen, die sich auf das Wesen des ausgezeichneten Texts beziehen.

SGML

Bereits Ende der 60er Jahre des 20. Jahrhunderts begann die Arbeit an einer *Generalized Markup Language (GML)*, die eine Notation für generisches Markup einführte. GML wurde 1986 durch die ISO[2] zur *Standard Generalized Markup Language (SGML)* standardisiert, allerdings blieb das Einsatzgebiet von SGML auf die professionelle Dokumentenverarbeitung beschränkt. Erst die Vereinfachung zu XML führte zu einer starken Verbreitung, sodass sich XML in den vergangenen sechs Jahren zum Mittel der Wahl bei der Auszeichnung von textuellen Daten und zur *Lingua Franca* des Internet entwickeln konnte.

[1] Auch das Markup selbst genügt gewissen grammatischen Regeln und bildet damit eine Sprache.

[2] ISO = International Organization for Standardization, siehe http://www.iso.org/

2.2 Ausgezeichnete Texte

XML (Extensible Markup Language – zu deutsch etwa *erweiterbare Auszeichnungssprache*) [W3C04a] ist eine Metasprache, mit der sich konkrete Markup-Vokabulare definieren lassen. XML selbst definiert damit keine Begriffe für konkrete Metainformationen, sondern legt ausschließlich die syntaktischen Regeln fest, nach denen Markup einem Text hinzugefügt wird. Die Definition eines konkreten XML-Vokabulars, d.h. einer für bestimmte Zwecke geeigneten und mit einer festen Semantik versehenen Markup-Sprache, liegt im Verantwortungsbereich des Anwenders.

Betrachten wir beispielsweise die nicht ganz ernst gemeinte Antwort aus dem Vorspann dieser Arbeit:

Beispiel 1

FAQ (Frequently Asked Questions)

> *»I've come across (while reading a document on WAP) a term – "XML Middleware". What does it mean?«*
>
> Middleware is a kind of American stretchable undergarment, for restraining bellies and all the big fat bottoms. For obvious reasons it does not come in a size S (small): the little man doesn't really need middleware. So it only needs to fit sizes X, M, and L.
>
> …
>
> I hope this helps.
>
> Rick Jelliffe

Einem menschlichen und der englischen Sprache mächtigen Leser sollte sofort klar sein, dass im ersten Absatz eine Frage gestellt und diese in den folgenden Absätzen beantwortet wird. Unter die Antwort hat der hilfsbereite Kollege eine Grußformel und seinen Namen gesetzt. Diese unterschiedlichen Bedeutungen der einzelnen Absätze können mit Hilfe von XML explizit gekennzeichnet werden, um auch einer Softwareanwendung die Auswertung dieser Bedeutungen zu ermöglichen, siehe Listing 1.

FAQ mit XML-Markup **Listing 1**

```
<?xml version="1.0" encoding="iso-8859-1"?>
<faq quelle="http://lists.xml.org/archives/xml-dev/200008/msg00489.html">
  <frage>I've come across (while reading a document on WAP) a term &#x2013;
    <begriff>XML Middleware</begriff>. What does it mean?</frage>
  <antwort>
    <!-- Vorsicht, Satire -->
    <absatz>Middleware is a kind of American stretchable undergarment, for
      restraining bellies and all the big fat bottoms. For obvious reasons it
      does not come in a size S (small): the little man doesn't really need
      middleware. So it only needs to fit sizes X, M, and L.</absatz>
    <absatz>...</absatz>
    ...
    <gruß>I hope this helps.</gruß>
    <name>Rick Jelliffe</name>
  </antwort>
</faq>
```

Alle fett dargestellten Bestandteile bilden das Markup. Es handelt sich hier um ein reines Textdokument ohne jegliche Formatierungen (keine Kursivschrift, keine speziellen Zeilenabstände, etc.). Der Grad der Strukturierung und damit der Umfang des hinzugefügten Markup hängen von der beabsichtigten Verwendung des Dokuments ab. Da hinsichtlich des Strukturierungsgrades eine große Bandbreite zu finden ist, werden solche in XML ausgezeichneten Daten als *semistrukturierte Daten* bezeichnet. Insbesondere sind rekursive, sich wiederholende oder auch sehr unregelmäßige Strukturen möglich. Solche sehr lockeren Strukturen, bei denen das Markup relativ frei einem Dokument hinzugefügt wird, werden *dokumentenzentriert* genannt (wie z.B. in Listing 1). Demgegenüber nennt man sehr regelmäßige Strukturen (etwa für Personaldaten oder Bestelllisten), bei denen eine vorgegebene Struktur mit Daten gefüllt wird, *datenzentriert*.

In der ersten Zeile des Listing 1 wird durch die XML-Deklaration gekennzeichnet, dass es sich dabei um ein XML-Dokument handelt. Die Einrückung wurde allein zur besseren Darstellung der XML-Struktur benutzt, sie besitzt (in diesem Beispiel) keine eigene Bedeutung.

Elemente

Wie zu sehen ist, benutzt XML spitze Klammern (<, >), um das Markup syntaktisch vom Inhalt des Textes zu trennen. Beispielsweise wird in Zeile 14 der Name des Autors explizit als solcher gekennzeichnet: `<name>Rick Jelliffe</name>`. Die Struktur, die ein solchermaßen ausgezeichneter Text besitzt, ist immer strikt hierarchisch. Jedes korrekte XML-Dokument lässt sich aus Bausteinen (den *Elementen*) der Art

```
<name> Inhalt </name>
```

zusammensetzen. Der beginnende Teil `<name>` ist das Start-Tag des Elements, der abschließende Teil `</name>` das End-Tag. Die hierarchische Struktur von XML bedingt damit, dass ein End-Tag immer zu dem zuletzt geöffneten Start-Tag passen muss.

Der Inhalt eines Elements kann aus Text oder weiteren Elementen bestehen. In obigem Beispiel besteht das Element `faq` aus den Elementen `frage` und `antwort`; das Element `antwort` wiederum besteht aus mehreren `absatz`-Elementen, einem `gruß`- und einem `name`-Element. Das Element `name` besitzt ausschließlich Textinhalt. Das Element `frage` besitzt dagegen gemischten Inhalt, d.h. Text mit eingebetteten Unterelementen, in diesem Fall das Element `begriff`. Falls ein Element keinerlei Inhalt besitzt, kann dies verkürzt als `<name/>` notiert werden.

Attribute

Zu jedem Element können in Form von *Attributen* im Start-Tag zusätzliche Informationen angegeben werden. So wird hier durch `<faq quelle="http://lists.xml.org/archives/xml-dev/200008/msg00489.html">` die Quelle dieses Zitats gekennzeichnet.

Während die im Inhalt eines Elements enthaltenen Unterelemente durchaus mehrfach auftreten können (siehe die Folge der `absatz`-Elemente in Listing 1), kann ein bestimmtes Attribut maximal einmal angegeben werden. Darüber hinaus ist die Reihenfolge, in der Unterelemente angegeben werden, in der Regel relevant, während die Reihenfolge bei Attributen keine Rolle spielt.

Weitere Arten von Markup innerhalb von XML-Dokumenten umfassen Kommentare (begrenzt durch die Zeichenfolgen `<!--` und `-->`, siehe Zeile 6) sowie in diesem Beispiel nicht auftretende Verarbeitungsanweisungen (*Processing Instructions*, begrenzt durch `<?` und `?>`) und die Dokumenttyp-Deklaration (`<!DOCTYPE ...>`).

Zeichen

Die in einem XML-Dokument erlaubten Zeichendaten umfassen den gesamten Unicode-Bereich [Unicode] mit Ausnahme der ASCII-Steuerzeichen im Bereich #x00 bis #x1F und den Sondercodes #xFFFE und #xFFFF.[3] Die Zeichen Tabulator (#x9), Zeilenwechsel (#xA) und Wagenrücklauf (#xD) gelten nicht als Steuerzeichen, sondern bilden zusammen mit dem Leerzeichen (#x20) den so genannten Leerraum (*white-space*). Während Erweiterungen des Unicode-Standards sich automatisch auf die im Textinhalt erlaubten Zeichen auswirken, definiert XML 1.0 detailliert die innerhalb von XML-Bezeichnern (z.B. Element- und Attributnamen) zulässigen Zeichen. Die Version 1.1 von XML [W3C04b] lockert diese Beschränkung, sodass nun weitaus mehr Zeichen ebenfalls innerhalb von Bezeichnern benutzt werden dürfen.

Entities und Entity-Referenzen

Enthält ein einfacher Text bereits Passagen, die als XML-Markup missverstanden werden könnten, muss die öffnende spitze Klammer innerhalb der Textdaten eines XML-Dokuments maskiert werden. XML bietet dazu so genannte *Entities*. Die am häufigsten verwendeten analysierten Entities wirken wie ein einfacher Textersetzungsmechanismus. So bedeutet die Notation `<`, dass an dieser Stelle der Inhalt (der Ersetzungstext) des Entity `lt` eingesetzt werden soll; in diesem Fall das Zeichen < (*less than*). Eine Entity-Referenz besteht immer aus dem Ampersand (`&`), dem Namen des Entity und einem Semikolon. Das Ampersand selbst muss nun seinerseits maskiert werden, soll es nur für sich selbst stehen und nicht den Beginn einer Entity-Referenz anzeigen. XML stellt dazu das Entity `amp` zur Verfügung. Weitere vordefinierte Entities sind `gt` (>), `apos` (') und `quot` (").

Mittels externer Entities kann ein logisches XML-Dokument in mehrere Dateien aufgeteilt werden. Der Inhalt des Entity stammt dann aus einer separaten Datei; über die Entity-Referenz wird diese Datei textuell eingefügt. Die Definition eigener Entities ist innerhalb der Dokumenttyp-Definition (DTD) möglich, auf die hier jedoch nicht weiter eingegangen wird.

CDATA-Abschnitte

Eine spezielle Form, ganze Textpassagen zu maskieren, stellen CDATA-Abschnitte dar. Innerhalb eines CDATA-Abschnitts werden weder die öffnende spitze Klammer noch das Ampersand als Sonderzeichen erkannt. Auf diese Weise kann relativ einfach ein XML-Text als purer Text in ein Dokument aufgenommen (beispielsweise in dieser Dissertation, in der sehr häufig XML-Beispiele angegeben werden). Ein CDATA-Abschnitt beginnt mit der Zeichenfolge `<![CDATA[` und endet mit der Zeichenfolge `]]>`.

Zeichenreferenzen

Spezielle Zeichen, die nicht direkt in einen Text eingegeben werden können (etwa, weil es für sie keine Taste auf der Tastatur gibt), oder die nicht in der gewählten Kodierung des Dokuments repräsentiert werden können (z.B. wenn ein griechischer Buchstabe in einen ASCII-Text eingefügt werden soll), können als Zeichenreferenz durch die Angabe ihres Zeichencodes notiert werden. Eine Zeichenreferenz besteht aus den beiden Zeichen `&#`, dem Zeichencode (dezimal oder hexadezimal) und einem abschließenden Semikolon. So bezeichnen sowohl `&#958;` als auch `&#x03BE;` den kleinen griechischen Buchstaben ξ. Das häufig benötigte geschützte Leerzeichen (*no-break space*) lässt sich als ` ` notieren. Listing 1 enthält in Zeile 3 die Zeichenreferenz `&#x2013;` für den Gedankenstrich (*en dash*).

Die hier vorgestellten Varianten ermöglichen unterschiedliche lexikalische Repräsentationen des gleichen Zeichens. Eine Applikation, die solche XML-Daten verarbeitet,

[3] Die hier und im Folgenden verwendete Notation #x*H* bezeichnet das Unicode-Zeichen mit dem Hexadezimalcode *H*.

arbeitet jedoch mit den eigentlichen Unicode-Zeichen und kann diese Repräsentationen nicht mehr unterscheiden. Dies ist in etwa mit der Möglichkeit in C oder Java vergleichbar, verschiedene Zeichenliterale für das gleiche Zeichen verwenden zu können.

Korrektheit

XML-Dokumente, die den syntaktischen Regeln für XML genügen, werden als *wohlgeformt* (*well-formed*) bezeichnet.[4] An dieser Stelle muss noch zwischen *XML-Dokumenten* und *XML-Fragmenten* unterschieden werden. Ein XML-Dokument muss immer genau ein äußerstes Wurzelelement enthalten, das alle anderen Elemente und den eigentlichen Textinhalt umschließt. Außerhalb dieses Wurzelelementes dürfen weder Text noch andere Elemente auftreten. Für ein XML-Fragment gilt diese Beschränkung nicht. Vereinfacht gesagt ist ein XML-Fragment dann wohlgeformt, wenn durch die Ergänzung eines umschließenden Elements ein wohlgeformtes XML-Dokument entstehen würde.

Dokumente und Fragmente

Neben der Wohlgeformtheit existiert als nächste Stufe der Korrektheit für XML-Dokumente das Kriterium *Gültigkeit*. Ein XML-Dokument ist dann gültig, wenn es den in einem Schema[5] formulierten Bedingungen genügt. Diese Bedingungen beschreiben, welche Elemente und Attribute im Dokument verwendet werden können und welchen Inhalt diese jeweils besitzen dürfen. Ein Schema definiert damit ein konkretes Vokabular. Ein Dokument ist somit gültig, wenn dieses Vokabular korrekt benutzt wird.

Der Begriff Gültigkeit sei hier ganz bewusst breiter gefasst, als ihn die XML-1.0-Spezifikation [W3C04a] beschreibt. Insbesondere existieren mit XML Schema [W3C01b], RelaxNG [OASIS01] oder Schematron [Schtrn] Schemasprachen, deren Möglichkeiten weit über die Mächtigkeit von DTDs in XML 1.0 hinausgehen.

Namensräume

Namensräume in XML erlauben es, Gruppen von Elementen oder Attributen unter einem gemeinsamen Oberbegriff zusammenzufassen. Auf diese Weise werden gleich lautende Elementnamen aus verschiedenen Anwendungsbereichen und mit unterschiedlicher Semantik unterscheidbar und können innerhalb desselben Dokuments verwendet werden. Das Konzept der Namensräume wurde erst nachträglich zu XML hinzugefügt und wird in einer eigenen Spezifikation beschrieben [W3C99a].

URIs

Ein Namensraum ist ein (potenziell) weltweit eindeutiger Bezeichner. Um diese Eindeutigkeit zu gewährleisten, werden für Namensräume so genannte URIs (*Uniform Resource Identifiers*) verwendet. Ein Spezialfall der URIs sind URLs (*Uniform Resource Locators*), die der Adressierung von Ressourcen im World Wide Web dienen. Benutzt man eine (fiktive) Adresse aus einer eigenen Domäne als Bezeichner für einen XML-Namensraum, ist damit dessen Einmaligkeit sichergestellt. Ein Beispiel für einen solchen Namensraum wäre *http://stx.sourceforge.net/2002/ns*. Allerdings sollte die Tatsache, dass hier eine Ortsbezeichnung (ein URL) verwendet wird, nicht dazu verleiten, an diesem Ort eine sinnvolle Ressource zu vermuten. In der Praxis hat die URL-Konvention allerdings schon zu vielerlei Missverständnissen und falschen Vorstellungen, insbesondere bei XML-Neulingen geführt. Ein Namensraum ist nur ein Bezeichner, dessen konkrete URL-Syntax jedoch nicht interpretiert wird.

[4] Genau genommen handelt es sich bei dem Begriff *wohlgeformtes XML* um eine Tautologie. Ein Dokument, das im XML-Sinn nicht wohlgeformt ist, ist per Definition kein XML.

[5] Der Begriff *Schema* ist hier als Oberbegriff für DTDs und andere Schemasprachen zu verstehen.

Namensraum-deklaration

Bevor ein Namensraum in einem XML-Dokument benutzt werden kann, muss er deklariert werden. Dies geschieht mit speziellen Attributen, deren Namen mit `xmlns:` beginnen. Durch eine solche Deklaration wird ein Kürzel vereinbart, das als Präfix in Element- oder Attributnamen den jeweiligen Namensraum angibt. Beispielsweise deklariert `xmlns:stx="http://stx.sourceforge.net/2002/ns"` das Präfix `stx` für den Namensraum *http://stx.sourceforge.net/2002/ns*. Der Elementname `stx:transform` bezeichnet dann das Element `transform` aus diesem Namensraum. Namen, die ein Präfix enthalten und in Bezug auf einen deklarierten Namensraum interpretiert werden müssen, werden als *qualifizierte Namen* bezeichnet. Die Präfixe für Namensräume können frei gewählt werden. Entscheidend zur Identifikation des Namensraums ist der jeweilige URI. Einen Spezialfall stellt die Deklaration mit dem Attribut `xmlns` dar, das einen Namensraum für alle Elemente ohne Präfix festlegt. Namensraumdeklarationen gelten für den gesamten Inhalt des Elements, das den Namensraum deklariert, es sei denn, eine andere Namensraumdeklaration weist dem verwendeten Präfix einen neuen Namensraum zu.

Das folgende Beispiel in Listing 2 demonstriert die Verwendung von Namensraumdeklarationen und Präfixen:

Ein XML-Dokument mit Namensräumen **Listing 2**

```
1 <?xml version="1.0"?>
2 <stx:transform xmlns:stx="http://stx.sourceforge.net/2002/ns" version="1.0">
3   <stx:template match="begriff">
4     <em>
5       <stx:process-children />
6     </em>
7   </stx:template>
8 </stx:transform>
```

In Zeile 2 erfolgt die Deklaration eines Namensraums für das Präfix `stx`. Alle enthaltenen Elemente mit diesem Präfix gehören damit zu diesem Namensraum. Das in Zeile 4 auftretende Element `em` ohne Präfix befindet sich demgegenüber in keinem Namensraum (bzw. dem leeren Namensraum). Zur semantischen Bedeutung dieses Dokuments vergleiche Kapitel 5.

Ab der Version 1.1 der Namensraumspezifikation [W3C04d] kann eine Namensraumdeklaration im Inneren eines Elements auch wieder rückgängig gemacht werden. Als Namensräume selbst können nun auch internationalisierte URIs, so genannte IRIs (*Internationalized Resource Identifiers*) verwendet werden.

Für sehr viele weitere Spezifikationen aus dem XML-Umfeld werden Namensräume als Basis vorausgesetzt, unter anderem auch für die in dieser Arbeit entwickelte Transformationssprache STX. Der Begriff der Wohlgeformtheit wird daher entsprechend erweitert. Ein XML-Dokument gilt dann und nur dann als wohlgeformt, wenn es gemäß XML 1.0 wohlgeformt ist und zusätzlich den Anforderungen für Namensräume genügt. Dies bedeutet konkret, dass der Doppelpunkt innerhalb von Element- und Attributnamen ausschließlich als Trennsymbol zwischen Präfix und lokalem Namen verwendet werden darf und alle Präfixe vor ihrer Verwendung deklariert werden müssen. XML-Dokumente, die diese Bedingung nicht erfüllen, werden hier nicht weiter betrachtet. Dies stellt keine Einschränkung dar, da das W3C von der Verwendung solcher Dokumente abrät. Solche Dokumente lassen sich weder mit der

Transformationssprache XSLT verarbeiten, noch kann für sie ein XML-Schema angegeben werden.

Vokabulare

Als Vokabular wird eine feste Menge von XML-Elementen und -Attributen bezeichnet. Ein Vokabular kann mit Hilfe eines Schemas formal definiert oder auch nur verbal beschrieben sein. Mit der Festlegung eines Vokabulars wird auch immer eine dazugehörige Semantik definiert, die Sinn und Zweck der jeweiligen Elemente und Attribute beschreibt. Wenn XML-Dokumente nicht dieser Semantik entsprechen, wird dies als *tag abuse* (Tag-Missbrauch) bezeichnet. Die semantisch falsche Verwendung von XML-Markup ist problematisch, weil sie kaum durch Computer-Software erkannt werden kann.

XML erlaubt es jedem Anwender, ein eigenes Vokabular zu erfinden. Für viele Zwecke existieren jedoch bereits Vokabulare, für die es ebenfalls geeignete Verarbeitungswerkzeuge gibt. Die folgende Liste enthält einige typische Anwendungsgebiete und erhebt keinen Anspruch auf Vollständigkeit.

- **Präsentation**
  Präsentationsvokabulare dienen der Beschreibung eines beabsichtigten Layouts. In diesem Fall handelt es sich daher um visuelles und nicht um generisches Markup. Typische Vertreter sind SVG (Scalable Vector Graphics) für die Beschreibung von Vektorgrafiken und XSLFO (XSL Formatting Objects) für die Beschreibung von druckbaren Texten. Als Grenzfall kann auch HTML (bzw. XHTML) zu dieser Kategorie gezählt werden, wobei visuelle Effekte mittlerweile vorrangig über externe Stylesheets (CSS), d.h. außerhalb von HTML, beschrieben werden.
- **Dokumentation**
  Das am weitesten verbreitete Vokabular für die Auszeichnung technischer Dokumentationen ist DocBook. DocBook eignet sich zum Schreiben von Artikeln, Manual-Seiten oder ganzer Handbücher. Weitere Vertreter sind DiML (Dissertation Markup Language), ein Vokabular für die digitale Repräsentation von Dissertationen,[6] sowie XMLSpec, das vom W3C als Quellformat aller eigenen Dokumente und Spezifikationen verwendet wird.
- **Repräsentation von komplexen Strukturen**
  In XML können eine Vielzahl naturwissenschaftlicher Strukturen ausgedrückt werden. So existiert mit CML (Chemical Markup Language) ein Vokabular zur Repräsentation chemischer Verbindungen. Mittels MathML lassen sich mathematische Formeln repräsentieren. GAME (Genome Annotation Markup Elements) ermöglicht die Kodierung von Genom-Informationen, wie sie in der DNA enthalten sind. Die Liste von Vokabularen gerade aus dem Bereich der Bioinformatik ist zu umfangreich, um sie hier vollständig aufführen zu können.
- **Konfiguration**
  Größere Software-Anwendungen können oft über spezielle Dateien konfiguriert werden. Insbesondere Web- und Applikationsserver verwenden dafür oftmals eigene XML-Vokabulare. Die Erstellung Java-basierter Projekte wird durch das

[6] Diese Arbeit wurde in XML mit Hilfe des DiML-Vokabulars verfasst und über XSLFO in ein les- und druckbares Format (in diesem Fall PDF) umgewandelt.

Technologiewerkzeug *ant* unterstützt, dessen Regeln als XML in der Datei *build.xml* abgelegt werden.

- **Datenaustausch**
  Applikationen, die untereinander Daten austauschen, verwenden dafür in zunehmenden Maße XML. XML ist wegen seiner standardisierten Syntax und frei wählbaren Semantik geradezu prädestiniert für diesen Anwendungsfall. Solche XML-Dokumente sind allerdings nicht dazu gedacht, von Menschen direkt gelesen oder bearbeitet zu werden.

  Ein Beispiel ist das Serialisierungsformat für UML namens XMI,[7] über das in UML erstellte Modelle zwischen verschiedenen UML-Anwendungen ausgetauscht werden können. Für Web Services wird in der Regel SOAP als Serialisierungsformat für die zwischen den beteiligten Partnern verschickten Daten verwendet.

  Im weiteren Sinne kann eine Schicht, die den XML-basierten Datenaustausch zwischen unterschiedlichen Applikationen ermöglicht, als *XML-Middleware* bezeichnet werden. Die Details der Datenrepräsentation als XML werden durch diese Schicht gekapselt und bleiben den beteiligten Applikationen verborgen.

- **XML-Infrastruktur**
  Das W3C hat seit der Verabschiedung von XML 1.0 eine Reihe spezieller Vokabulare entwickelt, deren Zweck in der Beschreibung und Verarbeitung anderer XML-Vokabulare besteht. In erster Linie sind hier XML-Schema (als eine in XML ausgedrückte Schemasprache für XML) und XSLT als XML-Transformationssprache zu nennen.

- **Wissensrepräsentation**
  Die Vision des zukünftigen World Wide Web wird als *Semantic Web* bezeichnet. Alle im weltweiten Netz verfügbaren Informationen sollen so repräsentiert sein, dass sie mit Hilfe von Computer-Programmen gefunden, aufbereitet und miteinander verknüpft werden können. Als Vokabular steht RDF (Resource Description Framework) [RDF] zur Verfügung, wobei die Herausforderung darin besteht, das vorhandene Wissen adäquat mit RDF auszuzeichnen.

2.3 Transformationen

Unter einer *XML-Transformation* kann vereinfacht die Umwandlung eines XML-Dokuments in ein anderes XML-Dokument verstanden werden. Eine solche Transformation ist somit eine Funktion, die ein Eingabe-XML-Dokument auf ein Ausgabe-XML-Dokument abbildet. Diese strenge Definition kann dahingehend verallgemeinert werden, dass anstelle allein stehender XML-Dokumente jeweils eine Folge von XML-Fragmenten auf Eingabe- und Ausgabeseite erscheinen darf, mathematisch ausgedrückt als

$$trans: \Xi^* \rightarrow \Xi^*$$

wobei $trans$ eine Transformationsfunktion und Ξ die Menge der XML-Fragmente bezeichnen soll.[8]

[7] UML = Unified Modeling Language; XMI = XML Metadata Interchange

[8] Die Notation Ξ^* steht für die Vereinigung aller Ξ^n, $n \in N$ und ist damit die Menge aller geordneten Folgen von XML-Fragmenten.

Die Transformation von XML-Dokumenten bzw. -Fragmenten kann aus unterschiedlichsten Gründen notwendig sein:

- **Überführung in ein Präsentationsformat**
  Die in XML repräsentierten Daten sollen veranschaulicht werden. Texte können so in XHTML oder XSLFO transformiert werden, um eine formatierte Darstellung zu erzeugen; Zahlen können über Diagramme in SVG visualisiert werden; jegliche in einem streng strukturierten Vokabular ausgezeichneten Daten können zur Ansicht in eine XHTML-Tabelle übertragen werden.
- **Austausch von Dokumenten**
  Sender und Empfänger von XML-Dokumenten verwenden häufig verschiedene Vokabulare. Dafür ist es jedoch nicht erforderlich, dass die beteiligten Softwarekomponenten viele verschiedene Vokabulare beherrschen. Vielmehr können die zu übermittelnden Daten unabhängig von Quelle und Ziel in einem separaten Transformationsschritt übersetzt werden.
- **Migration von Daten**
  Die Einbindung neuer Software kann die Anpassung der vorhandenen Daten an neue bzw. geänderte Vokabulare erfordern.
- **Filtern von XML-Inhalten**
  Wenn für die Weiterverarbeitung nur ein Teil der ursprünglichen XML-Daten relevant ist, kann eine Transformation einen Ausschnitt dieser Daten produzieren, indem sie bestimmte Bestandteile herausfiltert.
- **Anreichern der Daten**
  Im Zuge einer Transformation können Informationen hinzugefügt werden, beispielsweise Verweise auf Grafiken, eindeutige IDs, u.ä.
- **Kombination von Daten**
  Daten, die auf mehrere Einzeldokumente verteilt sind, können in ein gemeinsames Dokument überführt werden. So könnte ein spezielles Dokument die Funktion eines Wörterbuches übernehmen, das sprachabhängige Begriffe für die Erzeugung eines Präsentationsvokabulars enthält. Ein anderer Anwendungsfall wäre die Erzeugung eines gemeinsamen Dokuments, das durch die Vereinigung mehrerer gleichstrukturierter Einzeldokumente entsteht.
- **Umstrukturierung**
  Für viele Anwendungen ist es notwendig, Daten nach verschiedenen Kriterien zu sortieren oder nach unterschiedlichen Gesichtspunkten zusammenzufassen (gruppieren).

In der Praxis findet man häufig eine Kombination aus den genannten Fällen. Die Erzeugung eines Stichwortverzeichnisses in XHTML aus einem DocBook-Dokument beinhaltet beispielsweise das Herausfiltern von Daten (der Stichwörter), eine Anreicherung mit zusätzlichen Informationen (Einbindung von kleinen Grafiken als Icons), eine Umstrukturierung (die alphabetische Sortierung) und schließlich die Umwandlung in ein Präsentationsformat (XHTML).

Abhängig davon, auf welcher Strukturierungsebene auf die Eingangsdaten zugegriffen werden muss, kann zwischen *strukturellen* und *inhaltlichen Transformationen* unterschieden werden. Bei einer strukturellen Transformation werden die enthaltenen Daten selbst kaum modifiziert; sie finden sich zwar neu strukturiert aber inhaltlich weitgehend unverändert im Transformationsergebnis wieder. Diese Art von Transfor-

mation ist die reine Änderung von Markup an einem ansonsten unveränderten Text. Inhaltliche Transformationen hingegen werten darüber hinaus den textuellen Inhalt während der Transformation aus und modifizieren ihn gegebenenfalls.

Zur Illustration seien hier zwei Beispiele angegeben. Die folgende Abbildung wird von einer strukturellen Transformation vorgenommen:

Beispiel 2

Strukturelle Transformation

Eingabe

```
<faq quelle="http://lists.xml.org/archives/xml-dev/...">
  <frage>I've come across (while reading a document on WAP)
    a term &#x2013; <begriff>XML Middleware</begriff>.
    ...
```

Ausgabe

```
<faq>
  <source>http://lists.xml.org/archives/xml-dev/...</source>
  <question>I've come across (while reading a document on WAP)
    a term &#x2013; XML Middleware.
    ...
```

Hier wurden der Inhalt des Attributs `quelle` in ein zusätzliches Unterelement `source` verschoben und das Element `frage` in `question` umbenannt. Das Element `begriff` wurde durch seinen Inhalt ersetzt, d.h. im Ergebnis fehlt das Markup von `begriff`.

Ein Beispiel für eine inhaltliche Transformation hingegen wäre das Folgende:

Beispiel 3

Inhaltliche Transformation

Eingabe

```
<faq quelle="http://lists.xml.org/archives/xml-dev/...">
  <frage>I've come across (while reading a document on WAP)
    a term &#x2013; <begriff>XML Middleware</begriff>.
    ...
```

Ausgabe

```
<faq quelle="http://lists.xml.org/archives/xml-dev/...">
  <frage>Ich bin (beim Lesen eines Dokumentes über WAP) auf
    einen Begriff gestoßen &#x2013; <begriff>XML
    Middleware</begriff>.
    ...
```

Hier bleibt die Struktur des Dokumentes erhalten. Der textuelle Inhalt wurde jedoch fast komplett ausgetauscht.

Reale Anwendungsfälle für Transformation beinhalten in der Regel beide Komponenten – das Dokument wird sowohl strukturell als auch inhaltlich verändert. Inhaltliche Änderungen setzen jedoch Wissen über die Bedeutung des Dokuments und eine Interpretation seines Inhalts voraus. Oftmals lassen sich keine einfachen Regeln für solche Änderungen formulieren. Eine sprachliche Übersetzung wie in Beispiel 3 ist sicher ein Extremfall. Ein in der Praxis häufiger anzutreffender Fall ist die Transformation eines Elements

```
<date>2004-03-12</date>
```

in den Text

```
12. März 2004
```

Hier ist lediglich eine Abbildung von Zahlen auf Monatsnamen und die Umordnung von Jahr und Tag notwendig. Soll eine solche Transformation jedoch auch andere Datumsformate richtig interpretieren, so besteht wieder das Problem, aus einem prinzipiell beliebig aussehenden Datumstext dessen Struktur abzuleiten. Aber gerade zur Lösung solcher Probleme wurde generisches Markup, und also auch XML erfunden. Bei XML-Transformationen steht daher immer die Transformation der XML-Struktur im Vordergrund.

Formal betrachtet sind natürlich Fälle, in denen kein Zusammenhang zwischen XML-Eingabe und XML-Ausgabe besteht, ebenfalls Transformationen. Solche Transformationen sind jedoch uninteressant, da für sie keine Transformations-Regeln formuliert werden können. Für die Ausführung von XML-Transformationen mit Hilfe eines Computers besteht die Herausforderung darin, die in der Realität vorhandenen Regeln möglichst einfach zu notieren und durch eine entsprechende Transformations-Software ausführen zu lassen.

2.4 Abstrakte Syntax und Datenmodelle

Die im Kapitel 2.2 kurz vorgestellte XML-1.0-Spezifikation definiert die konkrete XML-Syntax. Für die Verarbeitung der in einem XML-Dokument enthaltenen Daten ist jedoch die abstrakte Syntax in Form einer *Informationsmenge* relevant. Diese Informationsmenge abstrahiert von bestimmten, rein syntaktischen Eigenschaften und wird durch das W3C als das *XML Information Set* [W3C04c] spezifiziert, im Folgenden kurz *Infoset* genannt.

XML-Dokument, -Text und -Daten

Wenn von nun an von einem *XML-Text* die Rede sein wird, so bezieht sich dieser Begriff explizit auf die textuelle Repräsentation, d.h. auf die konkrete Syntax. Mit *XML-Daten* sind dagegen Instanzen eines beliebigen Datenmodells gemeint. Solche XML-Daten entstehen in der Regel durch das Parsen eines XML-Textes, sie können aber auch auf anderem Wege konstruiert werden. In Abhängigkeit vom gewählten Parser, von der benutzten Programmiersprache und insbesondere vom konkreten Datenmodell kann es für den gleichen XML-Text eine Vielzahl unterschiedlicher Ausprägungen von XML-Daten geben. Das Infoset stellt eine gemeinsame Ausgangsbasis für alle diese Datenmodelle dar. Die Begriffe *XML-Dokument* und *XML-Fragment* werden im Folgenden als Oberbegriffe verwendet. Sie beziehen sich nicht auf eine spezielle Repräsentation der XML-Daten als XML-Text oder als Instanz eines Datenmodells.

Das Infoset definiert, welche Informationen in einem die Namensraumbedingungen [W3C99a] erfüllenden XML-Text als relevant gelten. Nur die im Infoset repräsentierten Informationen sollten sinntragend in einer XML-Applikation verarbeitet werden. Unterschiedliche XML-Texte, die jedoch zu identischen Informationsmengen bezüglich des Infoset führen, sind als semantisch gleich anzusehen; sie beschreiben dieselben Daten. Im Folgenden seien zur Illustration einige Beispiele textuell unterschiedlicher, jedoch semantisch identischer XML-Textfragmente angegeben.

`<faq quelle="http://xml.com" sprache="en">`	`<faq sprache='en' quelle='http://xml.com' >`
` </br>`	` `
`<x>å</x>`	`<x>&#xE5;</x>`
`<![CDATA[&]]>`	`&`

Diese Beispiele zeigen, dass bei Attributen die Reihenfolge, die Art der Anführungszeichen (doppelte oder einfache) und der Leerraum dazwischen nicht repräsentiert werden, nicht zwischen den beiden Varianten für leere Elemente unterschieden werden kann, Zeichen und Zeichenreferenzen gleichbedeutend sind und schließlich die Grenzen von CDATA-Abschnitten sich nicht im Infoset wieder finden.

Eine Applikation, die jedoch solch rein syntaktischen Unterschieden eine semantische Bedeutung zuerkennt, kann nicht mit anderen Applikationen interoperabel XML-Texte austauschen. Da gemäß Infoset die oben dargestellten Ausschnitte jeweils äquivalent sind, darf jede XML-verarbeitende Applikation frei eine textuelle Repräsentation auswählen bzw. eine Darstellung durch eine äquivalente andere Darstellung ersetzen. Für Applikationen, die eine eindeutig definierte textuelle Darstellung benötigen (beispielsweise für die Verwendung digitaler Signaturen), hat das W3C eine eigene Spezifikation herausgegeben, die eine kanonische Form für XML-Texte beschreibt, siehe [W3C01a].

Baumrepräsentation

Das Infoset modelliert ein XML-Dokument in Form eines Baumes, dessen Knoten Informationseinheiten genannt werden. Insgesamt existieren 11 Typen solcher Informationseinheiten. Für jeden Typ beschreibt das Infoset, welche Eigenschaften durch ihn repräsentiert werden. CDATA-Abschnitte werden zum Beispiel nicht durch eine eigene Informationseinheit repräsentiert. Zur Illustration zeigt Abbildung 1 einen Ausschnitt aus dem zum Listing 1 auf Seite 9 gehörenden XML-Baum.

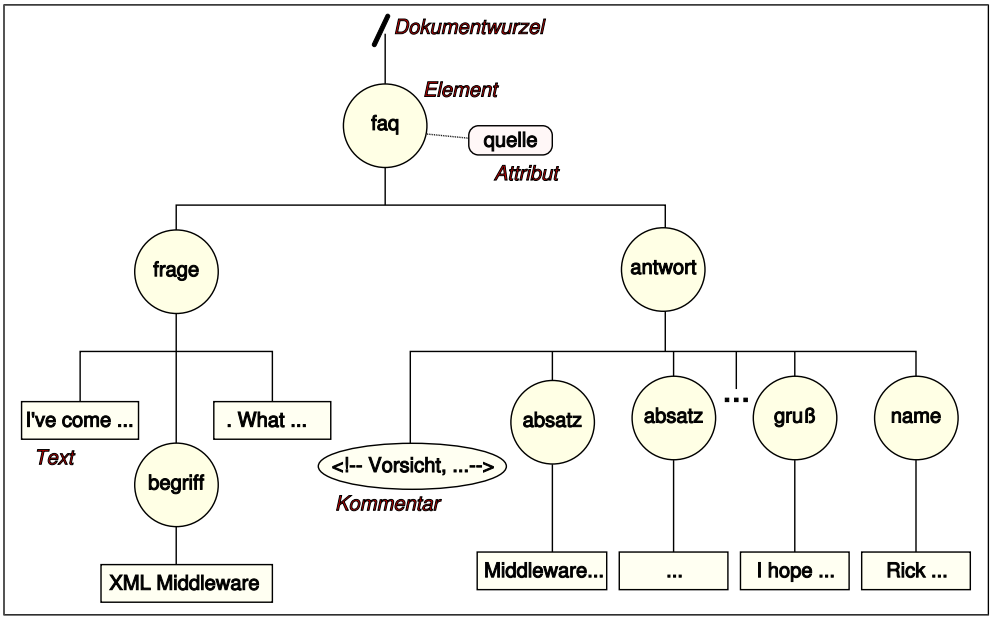


Abbildung 1

Die abstrakte Syntax als XML-Baum

Dieser Baum enthält nur Knoten fünf verschiedener Typen. Hier sind außerdem Textknoten, die ausschließlich Leerraum enthalten, nicht dargestellt. Solche Textknoten treten beispielsweise zwischen benachbarten Elementknoten auf. Wie außerdem

zu sehen ist, werden XML-Kommentare durchaus im Infoset repräsentiert. XML unterscheidet sich in dieser Beziehung von vielen Programmiersprachen, in denen Kommentare keinerlei semantische Bedeutung besitzen und deshalb in der jeweiligen abstrakten Syntax nicht berücksichtigt werden.

Das Infoset ist selbst kein API.[9] Es enthält keinerlei Vorgaben, wie die Eigenschaften der einzelnen Knoten zugänglich gemacht werden sollen. Insbesondere verzichten viele APIs darauf, die im Infoset modellierten Eigenschaften vollständig abzubilden. Auf der anderen Seite empfiehlt das W3C, keine über das Infoset hinausgehenden Eigenschaften einer Applikation zugänglich zu machen. Dass dies selbst dem W3C nicht hundertprozentig gelingt, ist dem evolutionären Entstehungsprozess der einzelnen Spezifikationen geschuldet. So existierte zum Zeitpunkt der Entwicklung des Infoset[10] bereits das Document Object Model (DOM) [W3C00] als Level 1. DOM beschreibt ein API für XML- und HTML-Dokumente und sieht dabei unter anderem einen eigenen Knotentyp für CDATA-Abschnitte vor.

Für Applikationen, die XML-Daten ausschließlich als Instanz eines konkreten Datenmodells verarbeiten, ist der dazugehörige XML-Text unerheblich. Insbesondere können solche Daten innerhalb der Applikation konstruiert werden, ohne dass es eines primären XML-Textes bedarf. Somit kann ein XML-Text als reine Serialisierung der Daten verstanden werden. Weitergehend ließe sich sogar von *einer möglichen* Serialisierung sprechen. Jede Darstellung in einem anderen Serialisierungsformat (einer anderen konkreten Syntax) wäre in gewissem Sinn immer noch XML. Beispielsweise wurde durch Stephen Williams ein binäres, Infoset-konformes Serialisierungsformat entworfen [Wil03], das sehr effizient verarbeitet werden kann. Dieses Format ist damit nicht mehr so »langatmig« (*verbose*) wie ein XML-Text (das Hauptargument der XML-Kritiker), allerdings lässt es sich auch nicht mehr mit einfachen Textwerkzeugen lesen und schreiben.

Darüber hinaus können Daten, die primär nicht als XML-Text vorliegen, jedoch strukturell auf das Infoset abgebildet werden können, in ein gewähltes XML-Datenmodell überführt werden. Dies geschieht in der Regel durch das Parsen anderer Formate oder durch die Überführung aus anderen Datenmodellen, etwa bei Daten, die aus einer relationalen Datenbank stammen. Damit steht auch für solche primär nicht-XML-Daten der große Fundus an XML-Werkzeugen zur Verfügung.

Die standardisierte XML-Syntax ist letzten Endes nur einer der Gründe, die zur starken Verbreitung von XML führten. Wenigstens genauso wichtig ist die Verständigung auf das Infoset, da nur so die Interoperabilität von Anwendungen, die XML-Daten austauschen, sichergestellt ist. Darüber hinaus können XML-Daten als programmiersprachliche Repräsentation eines XML-Datenmodells ausgetauscht werden, ohne dass es notwenig ist, einen XML-Text erst zu generieren und anschließend wieder einzulesen.

XML-Datenmodelle

Während das Infoset als abstraktes Datenmodell keinerlei Schnittstellen definiert, wie auf die in ihm repräsentierten Informationen zugegriffen werden kann, hat das

[9] API = Application Programming Interface

[10] Die Infoset-Spezifikation wurde erst im Oktober 2001 als W3C-Empfehlung erstmals verabschiedet; dreieinhalb Jahre nach XML 1.0, drei Jahre nach DOM Level 1 und gut zweieinhalb Jahre nach der Namensraum-Spezifikation.

W3C auch einige konkrete Datenmodelle spezifiziert, die im Folgenden kurz aufgeführt werden.

DOM

Das *Document Object Model (DOM)* [W3C00] entstand bereits ein halbes Jahr nach XML 1.0. Es definiert ein Objektmodell für XML-Dokumente in Form von Interface-Definitionen in OMG-IDL.[11] Dies ermöglicht eine sprachunabhängige Beschreibung des Modells. Für viele Programmiersprachen existieren geeignete Abbildungen dieser Interfaces auf programmiersprachliche Konstrukte. Allerdings können so sprachspezifische Eigenschaften der jeweiligen Programmiersprache nicht berücksichtig werden. DOM wirkt an vielen Stellen daher etwas schwerfällig und ineffizient. Mittlerweile wurden sprachspezifische Alternativen zu DOM entwickelt, beispielsweise für Java die APIs JDOM [JDOM] und DOM4J [DOM4J]. Beide verwenden Java-typische Klassen und erlauben somit eine effizientere Programmierung. Dessen ungeachtet hat DOM als W3C-Standard eine große Verbreitung gefunden.

Wie bereits erwähnt, lässt sich das DOM nicht vollständig auf das Infoset zurückführen, da DOM für CDATA-Abschnitte ein eigenes Interface vorsieht.

XPath 1.0

Die Pfadsprache *XPath 1.0* [W3C99b] spezifiziert eine eigene Syntax für den Zugriff auf die Bestandteile eines XML-Dokuments. Dazu wird ein eigenes Datenmodell definiert, das sieben Knotentypen enthält. Das XPath-Datenmodell lässt sich vollständig auf das Infoset abbilden, es spiegelt jedoch nicht alle im Infoset beschriebenen Details wider. So können beispielsweise keine Informationen aus der Dokumenttyp-Deklaration abgefragt werden. Auch zum DOM gibt es Unterschiede, da z.B. der Wert eines Knotens (sein Textinhalt) in DOM und XPath für Elementknoten unterschiedlich definiert wurde.

XPath 1.0 ist eine W3C-Empfehlung, die in anderen Spezifikationen benutzt wird. An erster Stelle ist hier die Transformationssprache XSLT 1.0 [W3C99c] zu nennen, aber auch XPointer [W3C02] und XML-Schema [W3C01b] greifen auf XPath zurück.

XPath 2.0

Im Zuge der Weiterentwicklung von XPath [W3C03a] und XSLT [W3C03b] sowie der Spezifikation einer Anfragesprache für XML-Daten namens XQuery [W3C03c] hat das W3C seine Spezifikationen in mehrere Dokumente zerlegt, um sie sprachübergreifend nutzen zu können. Grundlage für XPath 2.0 und XQuery 1.0 wird ein gemeinsames Datenmodell sein, das als *XQuery 1.0 and XPath 2.0 Data Model* [W3C03d] beschrieben ist. Dieses Datenmodell wird im Kapitel 5.3 genauer betrachtet.

Transformation von XML-Daten

Unter dem Aspekt, dass bestimmte syntaktische Details keinen Einfluss auf den Informationsgehalt eines XML-Textes haben, erscheint es sinnvoll, XML-Transformationen ebenfalls nur auf der abstrakten Syntaxebene zu betrachten. Eine solche Transformation kann damit Daten eines beliebigen (Infoset-konformen) Datenmodells verarbeiten, sie kann jedoch nicht auf Informationen zurückgreifen, die im Infoset nicht repräsentiert werden.

Beispielsweise ist es nicht möglich, mit einer solchen Transformation die Attribute eines Elements alphabetisch nach Namen zu sortieren oder die doppelten Anführungszeichen um den Attributwert durch einfache Anführungszeichen zu ersetzen. Mehr noch: es ist nicht einmal möglich, bestimmte im Eingabetext vorhandenen Eigenschaften zu erhalten. Wenn im verwendeten Datenmodell beispielsweise nicht repräsentiert

[11] OMG-IDL = Interface Definition Language der OMG (Object Management Group), siehe auch http://www.omg.org/gettingstarted/omg_idl.htm

ist, ob ein Zeichen als Zeichenreferenz notiert wurde oder nicht, so kann die gewählte Notation im Ergebnis der Transformation auch nicht reproduziert werden.

Wenn eine Transformation die gewählte konkrete Syntax möglichst unverändert lassen soll, müssen Datenmodelle benutzt werden, die über das Infoset hinausgehen.

Kapitel 3
Transformationsmethoden für XML

Wie bereits in Kapitel 2.3 erläutert wurde, soll unter einer XML-Transformation eine Funktion verstanden werden, die als Eingabe XML-Fragmente verarbeitet und im Ergebnis XML-Fragmente produziert. Für die Praxis relevant sind hier nur die algorithmisch berechenbaren Funktionen. Die Berechnung einer Transformationsfunktion geschieht in einem Transformationsprozess.

Abhängig von der eingesetzten Technologie lassen sich drei große Kategorien von Transformationsmethoden unterscheiden:

1. **Transformation auf der lexikalischen Ebene**
   Ein solcher Transformationsprozess verarbeitet XML als Text. Alle lexikalischen Eigenschaften sind sichtbar und können durch die Transformation modifiziert werden.

   Allerdings werden die durch das Infoset beschriebenen Informationseinheiten (die Knoten in der XML-Baumstruktur) nicht oder nur ansatzweise modelliert.

2. **Transformation mit Hilfe eines API**
   Mit Hilfe geeigneter Programmierschnittstellen (APIs) für XML kann in den verbreiteten Programmiersprachen von der Textform eines XML-Dokuments abstrahiert werden. Ein solches XML-API bildet den XML-Text auf eigene spezifische Datenstrukturen ab, die mehr oder weniger den Informationseinheiten des Infoset entsprechen. Ein Programmierer muss die Transformationslogik mit den Mitteln der jeweiligen Programmiersprache umsetzen.

3. **Einsatz einer speziellen Transformationssprache**
   XML-Transformationssprachen wurden speziell für die XML-Verarbeitung entworfen. Sie besitzen ein spezielles XML-Datenmodell und ermöglichen sowohl den einfachen Zugriff auf die zu transformierenden Daten als auch eine einfache Erzeugung des Ergebnisses. Der bereitgestellte Sprachumfang orientiert sich an den Anforderungen der XML-Verarbeitung.

Jede dieser drei Kategorien wird im Folgenden näher untersucht und durch konkrete Beispiele illustriert.

Anforderung: Skalierbarkeit

Im Vordergrund steht dabei die Anforderung, beliebig große XML-Datenmengen transformieren zu können. Idealerweise sollte die für die Transformation benötigte Zeit höchstens linear zur Dokumentgröße steigen und nicht von der Größe des zur Verfügung stehenden Hauptspeichers abhängen.

Die traditionelle Sicht auf ein XML-Dokument als Baum führt klassischerweise dazu, dass intern ebenfalls eine Repräsentation des gesamten Dokumentes aufgebaut wird. Bei wachsenden Dokumentgrößen ist dieser Ansatz jedoch ungeeignet, da die Größe des zur Verfügung stehenden Speichers den Umfang der transformierbaren Dokumente begrenzt. Transformationsmethoden, die eine Baumrepräsentation aufbauen, sind nicht skalierbar.

Nichtfunktionale Eigenschaften

Neben der Skalierbarkeit sind weitere nichtfunktionale Kriterien interessant, anhand derer die folgenden konkreten Transformationsmethoden bewertet werden sollen:

- **Abstraktionsgrad**
  Auf welcher logischen Ebene wird die Transformation beschrieben? Kann sich der Entwickler auf die eigentlichen Transformationsregeln konzentrieren, oder spielen andere Dinge (Datenstrukturen, Speichermanagement, Ablauflogik) eine nicht zu unterschätzende Rolle? Ermöglicht die gewählte Methode eine XML-typische Sicht auf die zu transformierenden Daten?

  Eine gute Transformationsmethode sollte die nicht direkt mit der Transformationslogik verbundenen Details vor dem Entwickler verbergen und eine XML-nahe Syntax für die XML-Daten verwenden.

- **Einfachheit**
  Wie kompliziert ist es, Regeln in der gewählten Transformationsmethode auszudrücken? Welchen Lernaufwand erfordert deren Anwendung für Anfänger?

  Eine gute Transformationsmethode sollte es Einsteigern ermöglichen, einfache Transformationsregeln auch ohne umfangreiche Vorkenntnisse umsetzen zu können. Einfache Transformationen sollten sich in einfachem Code ausdrücken lassen.

- **Wartbarkeit**
  Wie gut lässt sich der Transformationscode pflegen? Wie einfach lassen sich die Regeln im Code erkennen? Welchen Aufwand bedeutet es für andere Entwickler, sich in die Transformationslogik einzuarbeiten? Wie aufwändig ist es, Änderungen an den Ausgangsdaten oder im angestrebten Ergebnis in den Transformationsregeln nachzuvollziehen? Lassen sich die Transformationsregeln leicht erweitern?

  Eine gute Transformationsmethode sollte einen modularen Aufbau besitzen, der die Implementierung neuer Regeln erlaubt, ohne dass dazu vorhandene Regeln modifiziert werden müssen.

- **Mächtigkeit**
  Lassen sich mit der gewählten Methode alle berechenbaren XML-Transformationen umsetzen? Besitzt sie selbst genügend Mittel, um gegebenenfalls komplexe Berechnungen ausführen zu können? Existieren gegebenenfalls Schnittstellen zu universellen Programmiersprachen?

  Abhängig vom angestrebten Zweck der Transformation kann es sinnvoll sein, auf den vollen Umfang zugunsten von Effizienz zu verzichten. Eine gute Transformationsmethode sollte möglichst gut auf ihr angestrebtes Einsatzgebiet abgestimmt sein.

- **Robustheit**
  Wie robust ist die Methode gegenüber Programmierfehlern? Ist es möglich, dass das Ergebnis kein korrektes (wohlgeformtes) oder gültiges XML darstellt?

  Idealerweise sollte eine gute Transformationsmethode solche Fehler erkennen oder von vornherein vermeiden.

Nicht alle Kriterien werden sich gleichermaßen gut erfüllen lassen. Häufig muss ein Kompromiss zwischen den Kriterien, beispielsweise Mächtigkeit und Einfachheit, gefunden werden. Unnötige »Sprachfeatures« sollten vermieden werden. Eine gute Transformationsmethode sollte wie eine gute wissenschaftliche Theorie gemacht

sein: »so einfach wie möglich, aber nicht einfacher.«[1] Ebenso steht das Hauptkriterium Skalierbarkeit in der Regel im Konflikt mit der Mächtigkeit. Es ist kaum möglich, eine Transformation zu beschreiben, die freien Zugriff auf alle Daten eines XML-Dokumentes besitzt, aber trotzdem mit beliebig großen Dokumenten umgehen kann.

Neben diesen qualitativen Kriterien bestimmen jedoch sehr häufig sowohl betriebswirtschaftliche als auch subjektive Kriterien die Auswahl einer Transformationsmethode. Dazu zählen die Kosten von Hard- und Software, bereits vorhandene Kenntnisse der Entwickler und damit die zu erwartende Entwicklungszeit sowie persönliche Sympathien für die eine oder andere Vorgehensweise.

3.1 Transformationen auf lexikalischer Ebene

Eine Transformation auf lexikalischer Ebene verarbeitet die Zeichen, aus denen ein XML-Text besteht. Eine solche Transformation arbeitet in der Regel ohne XML-Parser, der die repräsentierte XML-Struktur erkennen würde. Es handelt sich somit um eine reine Zeichenmanipulation, die von der Bedeutung der Zeichen als XML keine Kenntnis nimmt.

Die Unterscheidung zwischen Markup und Inhalt erfordert damit einen zusätzlichen Aufwand für die Beschreibung der gewünschten Transformation. Ansonsten könnten möglicherweise unbeabsichtigt Element- oder Attributnamen verändert werden, wenn tatsächlich nur der Textinhalt modifiziert werden soll oder umgekehrt.

Transformationen auf lexikalischer Ebene sind dann sinnvoll und notwendig, wenn gerade die Zeichenrepräsentation beeinflusst werden soll. Sie sind insbesondere deshalb wichtig, weil eine XML-Applikation bei der Erstellung eines XML-Textes im Zuge der Serialisierung gewisse Freiheiten hat (vgl. Kapitel 2.4).

Erzeugen von Zeichenreferenzen

Sollen z.B. gewisse ISO-Latin1-Zeichen außerhalb des ASCII-Zeichensatzes durch geeignete Zeichenreferenzen ersetzt werden, ändert das nicht den logischen Inhalt des Dokumentes. Für eine anschließende Bearbeitung des Dokumentes mit einem ASCII-Editor kann eine solche Transformation, die als rein textbasierter Nachbearbeitungsschritt durchgeführt werden kann, aber durchaus sinnvoll sein. Das folgende *gawk*-Programm ersetzt zum Beispiel jedes Vorkommen des geschützten Leerzeichens #xA0 (*no-break space*) durch die dazugehörige Zeichenreferenz ` `:[2]

```
#!/usr/local/bin/gawk -f
{ gsub(/\xA0/,"\\ "); print }
```

Erhalt von Entity- und Zeichenreferenzen

Ein weiterer Anwendungsfall besteht darin, bestimmte lexikalische Eigenschaften während einer XML-Transformation zu bewahren, beispielsweise die Verwendung von Entity- oder Zeichenreferenzen. Ein XML-Parser würde bei der Überführung des Textes in die dem XML-Infoset entsprechende Repräsentation solche Referenzen durch ihren Inhalt ersetzen. Die Referenz selbst ist für die Applikation nicht mehr

[1] Albert Einstein, Physiker, 1879–1955

[2] *gawk* ist die GNU-Version des vor allem unter Anwendern von UNIX-Systemen sehr beliebten Werkzeugs *awk*. Das zitierte Beispielskript verarbeitet jede Eingabezeile (das Pattern vor der öffnenden geschweiften Klammer fehlt). Es ersetzt darin mit der Funktion `gsub` alle Vorkommen des Zeichens `\xA0` (das Zeichen mit dem Hexadezimalcode A0) durch die Zeichenfolge ` ` und gibt das Ergebnis durch Ausführung der Anweisung `print` aus. Da das Ampersand-Zeichen `&` für `gsub` eine spezielle Bedeutung hat, muss es durch das Voranstellen eines Backslash-Zeichens maskiert werden, welches seinerseits in *awk* als Doppel-Backslash notiert werden muss.

sichtbar. Ein anschließender Serialisierungsschritt kann die Referenzen nicht wieder rekonstruieren, weil die entsprechenden Informationen nicht mehr vorhanden sind.

Nutzer, die XML-Dokumente manuell mit einfachen Texteditoren erstellen, möchten jedoch in der Regel die volle Kontrolle über ihren Text behalten. Beispielsweise lässt sich ein wiederkehrender Text (z.B. eine temporäre Adresse oder URL) in einem internen Entity definieren und dann mehrfach über die entsprechende Referenz der Form `&adresse;` verwenden. Wird die Adresse später geändert, ist dies nur an einer einzigen Stelle im Dokument notwendig. Auch nach einer XML-Transformation sollte diese Referenz noch als solche vorhanden sein.

Ein anderer Fall ist die Aufteilung eines großen Dokumentes in mehrere Dateien mit Hilfe externer Entities. Auch hier gibt es Fälle, in denen automatisch Änderungen an den XML-Dateien vorgenommen werden sollen, die gewählte Aufteilung jedoch erhalten bleiben soll.

Die Vorgehensweise in diesen Fällen besteht darin, die Referenz vor dem folgenden Transformationsschritt zu maskieren, indem sie aus Sicht des XML-Parser in normalen Textinhalt verwandelt wird. Dazu reicht es aus, das führende Ampersand-Zeichen durch eine spezielle Zeichenkette zu ersetzen, die sonst nicht im Dokument auftreten darf und als Platzhalter fungiert. Nach der Transformation, die den Platzhalter nicht verändert haben darf, wird eine einfache Rückersetzung vorgenommen.

In einer UNIX-Shell könnte man diese drei Schritte in einer Pipe miteinander verknüpfen. Das folgende Beispiel zeigt einen entsprechenden Aufruf. Die Ersetzung des Ampersand-Zeichens durch den Platzhalter `§§§` leistet hier jeweils ein kleines *sed*-Skript. Das fiktive Programm *trans* repräsentiert ein beliebiges Programm, das einen XML-Text von der Standardeingabe liest und das transformierte Ergebnis auf die Standardausgabe schreibt:

```
sed -e 's/&/§§§/g' XML-Quelle | trans | sed -e 's/§§§/\&/g'
```

Umschreiben von Namensräumen

Ein weiterer Anwendungsfall besteht in der Änderung von Namensräumen. Namensräume werden im XML-Text in der Regel nur einmal deklariert, gelten dann aber für den gesamten Teilbaum, in dessen Wurzel die Deklaration erfolgt. Eine Änderung eines Namensraums auf der Datenmodellebene würde die Änderung in allen Elementen des entsprechenden Teilbaums bedeuten. Diese potenziell aufwändige Transformation lässt sich bereits mit einer einfachen Textersetzung realisieren. Allerdings muss dabei sichergestellt werden, dass nicht versehentlich die gleiche Zeichenfolge ungewollt an anderen Stellen ebenfalls ersetzt wird.

Bewertung

Transformationen auf lexikalischer Ebene können für beliebig große XML-Texte angewendet werden, da sie von sich aus keine interne Repräsentation aufbauen. Allerdings sind sie nur für solche Aufgaben geeignet, in denen die Manipulation der Zeichen in einem XML-Text im Vordergrund steht. Eine Unterscheidung zwischen Inhalt und Struktur erfordert dagegen einen unverhältnismäßig großen Programmieraufwand. Dementsprechend sind auch keinerlei Mechanismen zur Fehlererkennung vorhanden, mit denen sichergestellt werden kann, dass das Transformationsergebnis als XML syntaktisch korrekt ist. Lexikalische Transformationen stellen jedoch zu den übrigen XML-Transformationen, d.h. zu solchen, die auf einem XML-Datenmodell beruhen, eine durchaus sinnvolle Ergänzung dar.

3.2 Transformationen mit Hilfe von XML-APIs

APIs sind Schnittstellen zu Modulen oder Code-Bibliotheken, die eine spezielle Funktionalität anbieten. Ein API besitzt abhängig von der verwendeten Programmiersprache immer eine konkrete Ausprägung, beispielsweise in Form von Klassen und Funktionssignaturen.

Allen APIs für die XML-Verarbeitung ist gemeinsam, dass sie einen Parser beinhalten, der einen XML-Text analysiert und der Applikation den Zugriff auf die enthaltenen Daten ermöglicht. Sie unterscheiden sich jedoch in der konkreten Repräsentation dieser XML-Daten. Die Transformationsregeln selbst müssen vom Entwickler mit den Mitteln der verwendeten Programmiersprache realisiert werden.

Eine XML-Transformation auf dieser Ebene beinhaltet die folgenden Schritte:

1. **Einlesen und Analysieren des XML-Textes (Parsen)**
   Dieser Vorgang erkennt XML-Markup in einem Text und zerlegt den Text in seine Bestandteile. Ein XML-Parser, der sich am XML-Infoset orientiert, löst bereits Entity-Referenzen auf und überliest irrelevante Informationen (wie z.B. Leerraum innerhalb von Start-Tags).
2. **Aufbau einer internen Repräsentation der XML-Daten**
   Die Applikation fügt die durch den XML-Parser im vorherigen Schritt gelieferten Daten in eine größere Struktur ein. In der Regel handelt es sich dabei um ein Objektmodell, das das XML-Dokument oder Teile davon repräsentiert.
3. **Umwandlung dieser internen Repräsentation**
   Diesen Schritt könnte man auch als die eigentliche Transformation bezeichnen. Der Transformationsalgorithmus arbeitet auf den in der internen Repräsentation vorliegenden Daten und erzeugt daraus ein Ergebnis, das in der Regel ebenfalls zunächst in Form von Datenstrukturen der verwendeten Programmiersprache repräsentiert ist.
4. **Serialisierung des Ergebnisses**
   Als abschließender Schritt muss aus der internen Darstellung des Ergebnisses wieder XML-Text erzeugt werden. Dieser Schritt kann entfallen, wenn nachfolgende Verarbeitungsschritte mit der internen XML-Repräsentation weiterarbeiten.

Abhängig davon, in welchem Umfang der zweite Schritt (Aufbau der internen Datenrepräsentation) durch das API durchgeführt wird, lassen sich drei prinzipielle Herangehensweisen identifizieren:

1. Bereitstellung eines Datenstroms ohne den Aufbau einer zusammenhängenden Struktur (streambasiert)
2. Aufbau einer generischen Struktur, die im weiteren Sinne dem Baum des Infoset entspricht (baumbasiert)
3. Aufbau einer spezifischen, meist durch ein Schema definierten Baumstruktur (schemabasiert)

3.2.1 Streambasierte Transformationen

Eine streambasierte XML-Transformation verarbeitet die durch einen XML-Parser oder eine XML-Applikation gelieferten XML-Daten als Strom. Die Transformationslogik kann dabei

- *passiv* die Daten in Form von Events vom Parser entgegennehmen (Push-Modell) oder
- *aktiv* die Daten vom Parser abfragen (Pull-Modell)

In beiden Fällen verarbeitet die Transformationslogik pro Schritt nur ein weiteres Stück des Dokuments. Der Aufbau größerer Datenstrukturen liegt im Verantwortungsbereich des Entwicklers, der solche streambasierten APIs benutzt.

SAX

Der bekannteste Vertreter des Push-Modells ist das *Simple API for XML (SAX)* [SAX]. SAX entstand als Open-Source-Entwicklung und unterliegt keinem Urheberrecht (*public domain*). Ursprünglich für Java entworfen, existieren mittlerweile Versionen für viele andere Programmiersprachen. Sun hat SAX in das Paket JAXP (*Java API for XML Processing*) aufgenommen, welches seit der Version 1.4 zur Java-Standardausgabe (J2SE) gehört.

Pull-Parser

Die Entwicklung von APIs für das Pull-Modell begann erst nach der Veröffentlichung von SAX. Die gegenwärtige Arbeit konzentriert sich auf die Spezifikation eines standardisierten Java-API innerhalb des Java Specification Request 173 [JSR173], als dessen Vorläufer das *XmlPull*-API [XmlPull] angesehen werden kann.

Leider existiert in der Praxis keine eineindeutige Zuordnung der Informationseinheiten des XML-Infoset zu den durch die verschiedenen Parsertypen gemeldeten Informationen. Beispielsweise meldet ein SAX-Parser nicht, in welcher Kodierung das XML-Dokument vorliegt, obwohl diese Information im Infoset vorgesehen ist. Demgegenüber werden CDATA-Abschnitte im XML-Infoset nicht repräsentiert, wohingegen ein SAX-Parser diese Information liefert.

Neuere Ideen gehen sogar soweit, möglichst jede Information auf lexikalischer Ebene einer Applikation zur Verfügung zu stellen, beispielsweise im Gorille-Projekt von Simon St. Laurent [StL03].

Eine XML-Transformation, die einen XML-Datenstrom verarbeitet, muss ein eigenes Modell für die XML-Daten aufbauen. Speicherung und Verarbeitung der Daten liegen im Verantwortungsbereich der Applikation. Sie bestimmt, wie umfangreich eine solche interne Datenstruktur ausfällt. Insbesondere kann eine solcherart programmierte Transformation dynamisch genau die Daten im Speicher halten, die für den aktuellen Transformationsschritt notwendig sind, und Speicher für nicht mehr benötigte Daten wieder freigeben.

Eine rein streambasierte Lösung ist dann sinnvoll, wenn für die auszuführende Transformation wenige Kontextinformationen, also Daten aus anderen Teilen des Eingabedokuments benötigt werden. Ein typischer Anwendungsfall dafür ist die einfache Umbenennung von bestimmten Elementen.

Serialisierung

SAX ist ein reines Parser-API, das keine Funktionen zur Generierung von XML beinhaltet. Demzufolge gehört es zum Aufgabenbereich des Programmierers, korrektes XML im Ergebnis der Transformation auszugeben. Insbesondere erfordert die Maskierung der Zeichen < und & als `<` bzw. `&` erhöhte Aufmerksamkeit. Eine

SAX-basierte Transformation, die das Ergebnis direkt selbst als Text ausgibt, gewährt keine Sicherheit, dass dieser Text fehlerfrei ist.

Eine Lösung ist die Benutzung ergänzender APIs, die den durch einen SAX-Parser gelieferten XML-Datenstrom wieder in einen XML-Text serialisieren. In diesem Fall muss der zu programmierende Transformationscode ebenfalls einen SAX-Datenstrom produzieren. Werden die transformierten Daten an eine Serialisierungskomponente gegeben, erzeugt diese daraus korrektes XML und kann bereits Verstöße gegen die Anforderungen der Wohlgeformtheit erkennen.

Bewertung

Transformationen auf der Basis eines Stream-API sind prinzipiell für beliebig große Dokumente möglich, da diese keine zusammenhängende Repräsentation des gelesenen XML-Dokumentes erzeugen. Es liegt allein in der Verantwortung des Programmierers, eigene Datenstrukturen für die zu speichernden XML-Daten aufzubauen und zu verwalten. Dies bedeutet gleichzeitig einen generell erhöhten Programmieraufwand. Für nichttriviale Transformationen kann der zu erstellende Code leicht komplex und unübersichtlich werden. Die XML-Daten selbst werden als Folge von Funktionsaufrufen repräsentiert und haben in dieser Form nichts mehr mit der XML-Syntax gemein. Darüber hinaus handelt es sich hier um reine Parser-APIs, die den Aspekt der XML-Generierung nicht berücksichtigen und daher keine Unterstützung für die Erzeugung von korrektem XML bieten.

3.2.2 Baumbasierte Transformationen

Baumbasierte APIs stellen dem Entwickler eine Baumansicht der XML-Daten zur Verfügung. Im Gegensatz zu streambasierten APIs wird kein Datenstrom, sondern eine komplette Datenstruktur erzeugt. Diese repräsentiert das gesamte XML-Dokument entsprechend einer abstrakten Syntax und ermöglicht den freien Zugriff auf alle enthaltenen Informationen.

Eine Transformation mit Hilfe eines baumbasierten API muss somit Änderungen an dem bereitgestellten Baum vornehmen. Es können neue Objekte erzeugt, andere entfernt oder Eigenschaften der Objekte geändert werden. Wenn sich das gewünschte Ergebnis strukturell sehr stark von den Eingangsdaten unterscheidet (etwa bei der Transformation in ein anderes Vokabular), kann es günstiger sein, eine neue Objektstruktur für das Resultat der Transformation aufzubauen und die Eingangsdaten unverändert zu lassen.

Das W3C hat mit dem *Document Object Model (DOM)* ein sprachunabhängiges XML-Datenmodell spezifiziert, das für mehrere Programmiersprachen in Form konkreter APIs verfügbar ist. Das DOM stellt das bekannteste baumbasierte API für XML dar. Wie bereits in Kapitel 2.4 erwähnt wurde, haben sich darüber hinaus alternative APIs entwickelt, die effizienter und sprachspezifischer sind als DOM. Für Java sind hier JDOM [JDOM] und DOM4J [DOM4J] zu nennen. Eine XML-Transformation bedeutet in jedem dieser APIs die Manipulation von Objektstrukturen mit den Mitteln der gewählten Programmiersprache. Die anschließende Erzeugung eines XML-Textes aus einer solchen Objektstruktur übernehmen in der Regel ebenfalls Funktionen des API.

Bewertung

Die Repräsentation des gesamten XML-Dokuments als Objektstruktur hat zur Folge, dass nur Dokumente mit begrenzter Größe auf diese Weise verarbeitet werden können. In Abhängigkeit von der konkreten Implementation belegt der Objektbaum das fünf- bis zehnfache des Speicherplatzes, den das dazugehörige Textdokument benötigt.

Mögliche Auswege für dieses Problem sind APIs, die die Baumstruktur verzögert erst bei Bedarf erzeugen (z.B. *deferred DOM* im Xerces [ASFa]) oder den benötigten Speicher virtuell auf externen Speichermedien simulieren. Beides schlägt sich in einer geringeren Performance nieder.

Die Repräsentation von XML in Form von Objekten und deren Transformation mit den Mitteln einer objektorientierten Programmiersprache erschweren das Erkennen der XML-Daten und der implementierten Transformationsregeln. Allerdings können durch die Benutzung einer universellen Programmiersprache beliebig komplexe Berechnungen durchgeführt werden. Die Objektstruktur stellt eine gewisse Konsistenz der Daten sicher. So werden Fehler in dieser Struktur spätestens bei der Serialisierung durch das API gemeldet.

Seit der Entwicklung spezieller Transformationssprachen reduzierte sich der Anwendungsbereich für Transformationen in DOM erheblich. Dies gilt um so mehr, da inzwischen ebenfalls Transformations-APIs existieren, denen ein DOM- oder JDOM-Baum übergeben werden kann, und die dann einen solchen Baum mit den Mitteln einer speziellen Transformationssprache transformieren. Das für Java entwickelte Transformations-API TrAX wird in Kapitel 7.2 genauer vorgestellt.

3.2.3 Schemabasierte Transformationen

Während DOM und ähnliche APIs entsprechend den Informationseinheiten des Infoset einen generischen Baum modellieren, ermöglichen so genannte *Data-Binding-Frameworks* wie zum Beispiel Castor [Castor] die Erzeugung eines spezifischen, auf ein konkretes Vokabular zugeschnittenen Modells. Dieses lässt sich mit Hilfe eines Schemas automatisch generieren. In einem solchen Modell existiert anstelle des generischen Knotentyps *Element* für jeden XML-Elementtyp ein eigener Objekttyp. Dieser Objekttyp lässt von vornherein nur solche Kindelemente oder Attribute zu, die auch im Schema beschrieben worden sind.

Die Programmierung auf dieser Ebene unterscheidet sich damit nur unwesentlich von der in Kapitel 3.2.2 beschriebenen. Allerdings muss bei Transformationen in ein anderes Vokabular eine vollständig neue Baumstruktur aufgebaut werden. Die Manipulation des Eingabebaumes ist nur bei inhaltlichen Transformationen, d.h. bei Änderungen des Inhalt unter Beibehaltung der vorhandenen Struktur möglich. Der einfache Fall einer reinen Elementumbenennung ist bereits eine strukturelle Transformation, die damit den Aufbau eines vollständig neuen Ergebnisbaumes erfordert.

Schemabasierte Transformationen erfordern ein Schema für Eingabe- und Ausgabedaten. Der Zugriff auf XML-Markup, das nicht durch ein Schema beschrieben wird, ist mit einem schemabasierten API nicht möglich. So können beispielsweise weder Kommentare noch Verarbeitungsanweisungen transformiert werden. Die Existenz eines Schemas hat des Weiteren zur Folge, dass im Ergebnis immer gültige XML-Dokumente erzeugt werden. Verstöße gegen das Schema können bereits durch das API gemeldet werden.

In der Praxis sind strukturelle Transformationen mit Hilfe schemabasierter APIs eher selten anzutreffen. Data-Binding-Frameworks sollen vor allem einen typisierten Zugriff auf XML-Daten ermöglichen. Für pure inhaltliche Transformationen eignet sich diese Transformationsmethode jedoch sehr gut.

3.2.4 Funktionale Sprachen

In den vorherigen Kapiteln wurde vorausgesetzt, dass von prozeduralen und imperativen Sprachen die Rede ist. Tatsächlich verhindert insbesondere ein durch DOM suggerierter Objektcharakter von XML oftmals den unvoreingenommenen Blick auf alternative Möglichkeiten. Doch XML ist nicht per se objektorientiert. Gerade eine Vielzahl funktionaler Programmiersprachen harmoniert sehr gut mit XML. Diese sind häufig ebenso wie XML deklarativ. Sie bieten Datenstrukturen, die eine direkte Repräsentation der XML innewohnenden Baumstruktur erlaubt. Einen Überblick über verschiedene Ansätze gibt Parsia in [Par01].

Für die Sprache Haskell wurde beispielsweise eine Sammlung von Funktionen namens HaXml [HaXml] entwickelt. XML-Daten werden hier durch Haskell-eigene Typen beschrieben. Verschiedene Werkzeuge, wie z.B. ein Parser und ein Pretty-Printer ermöglichen die Ein- und Ausgabe von XML. Für die so repräsentierten XML-Daten können beliebige Funktionen in Haskell definiert und aufgerufen werden. Über die Kombination solcher Funktionen lassen sich mehrere Transformationsschritte hintereinander ausführen. Dies fördert eine klare Struktur und die Wartbarkeit der beschriebenen Transformation.

Darüber hinaus unterstützt HaXml den schemabasierten Ansatz, indem aus einer DTD spezifische Haskell-Typen generiert werden können. Damit ist es möglich, in Haskell ebenfalls typsichere Transformationen (bezogen auf den Dokumenttyp) zu programmieren.

Haskell erfordert für den mit dem Konzept der funktionalen Programmierung unerfahrenen Entwickler sicher einen hohen Lernaufwand. Hat man diesen jedoch bewältigt, steht mit HaXml eine mächtiges Werkzeug für die Verarbeitung von XML-Daten bereit.

Der Nachteil des funktionalen Ansatzes besteht darin, dass ein solches Programm die zu verarbeitenden Daten in der Regel vollständig als Eingabe für die zu berechnende Funktion benötigt. Zwar existiert in vielen funktionalen Sprachen das Konzept der Bedarfsauswertung von Ausdrücken (*lazy evaluation*), das die Bewältigung theoretisch unendlich großer Datenstrukturen ermöglicht. Die Anwendung für den Bereich der Datentransformation[3] ist jedoch derzeit noch Gegenstand der Forschung, siehe Gibbons in [Gib04]. Insbesondere erfordern solche speziellen Algorithmen neben der (bisher fehlenden) Unterstützung durch die XML-APIs einen durchdachten Entwurf der eigenen Transformationslogik, da diese ansonsten ein inkrementelles Verarbeiten der XML-Daten verhindert.

3.3 Spezielle Transformationssprachen

XML-Transformationssprachen wurden speziell für XML entworfen. Sie besitzen ihre eigene Syntax und sind anders als APIs nicht an eine bestimmte »Wirtssprache« gebunden.

[3] Bezogen auf allgemeine Datenstrukturen in funktionalen Sprachen wird hier auch der Begriff *Metamorphosen* verwendet.

3.3.1 DSSSL

Die Abkürzung DSSSL steht für *Document Style Semantics and Specification Language*. Sie wurde ursprünglich als Stilsprache für SGML entworfen, kann aber ebenso für XML-Dokumente verwendet werden. Allerdings berücksichtigt DSSSL nicht die besondere Bedeutung von Namensraumdeklarationen in XML. DSSSL ist wie SGML ein ISO-Standard [ISO96].

DSSSL besteht aus zwei Komponenten: einer Transformationssprache und einer Stilsprache. Diese bildeten die Vorlage für die spätere Entwicklung der XML-Stilkomponente XSL. Allerdings existiert in DSSSL noch keine strikte Trennung dieser beiden Komponenten. Obwohl DSSSL und XSL semantisch sehr ähnlich sind, unterscheiden sie sich doch sehr stark in ihrer Syntax. DSSSL basiert auf Scheme, einem Lisp-Dialekt. Dies bedeutet für viele mit Lisp nicht vertraute Anwender eine recht hohe Anfangshürde.

Somit ist DSSSL auch eine vollwertige, funktionale, seiteneffektfreie Programmiersprache. Ihr Datenmodell modelliert die SGML-Eingabe als *Groves*, eine Menge aus Knoten, die aus SGML-Dokumenten stammen und jeweils Unterbäume aufspannen können.[4] DSSSL benötigt daher konzeptionell eine Gesamtsicht auf die Eingabe, d.h. die vollständige Repräsentation der Eingabedokumente im Speicher.

Da einerseits die wesentlichen semantischen Konzepte aus DSSSL in XSL übernommen worden sind und zum anderen in DSSSL keine XML-Namensräume verarbeitet werden können, wird DSSSL an dieser Stelle nicht genauer untersucht.

3.3.2 XSLT

XSLT ist eine Transformationssprache für XML, die im Zusammenhang mit XSL (*Extensible Stylesheet Language*) entwickelt wurde. Diese Sprache sollte sowohl DSSSL vereinfachen und für die speziellen XML-Eigenschaften anpassen als auch alle Möglichkeiten der HTML-Stilkomponente CSS (*Cascading Stylesheets*) bieten und erweitern. Im Ergebnis des Standardisierungsprozesses beim W3C entstanden schließlich drei Spezifikationen:

- **XSL**, das besser XSLFO heißen sollte und mit den so genannten *Formatting Objects (FO)* ein spezielles layout-orientiertes XML-Vokabular beschreibt [W3C01e]
- **XSLT** als Transformationsteil innerhalb der XSL-Sprachfamilie [W3C99c]
- **XPath**, das eine innerhalb von XSLT benötigte einfache XML-Abfragesprache spezifiziert [W3C99b]

In XSL existieren XSLT und XSLFO als voneinander unabhängige Spezifikationen. Beide sind selbst in XML ausgedrückt. XPath verwendet eine kompakte Nicht-XML-Syntax und wird neben XSLT von weiteren Technologien benutzt (z.B. XPointer [W3C02] und XML Schema [W3C01b]).

Das Akronym XSLT steht für *XSL Transformations* und beschreibt damit den ursprünglichen Zweck der Sprache: Transformationen für XSL[5]. XSLT enthält jedoch

[4] Ein *grove* ist ein Hain bzw. eine Baumgruppe.

[5] Die vollständige Auflösung als *Extensible Stylesheet Language Transformations* demgegenüber verwirrt eher.

keinerlei stilspezifische Eigenschaften. Vielmehr handelt es sich um eine allgemeine XML-Transformationssprache, auch wenn die XSLT-Spezifikation selbst im Vorwort eine andere Intention statuiert. Jede als XML vorliegende Eingabe kann in eine beliebige andere XML-Ausgabe umgeformt werden. Einige in der ersten XSLT-Version nur umständlich zu lösenden Transformationsaufgaben werden sich in XSLT 2.0 kürzer und effizienter programmieren lassen. Dass XSLT-Programme üblicherweise generell als Stylesheets bezeichnet werden, hat allein historische Ursachen.

XSLT ist das Mittel der Wahl für allgemeine XML-Transformationen. Aufgrund seiner Bedeutung und großen Verbreitung wird diese Sprache im Kapitel 4 ausführlich analysiert.

3.3.3 XQuery

XQuery [W3C03c] ist eine komplexe Anfragesprache für XML. Sie basiert auf XPath 2.0 [W3C03a] und ist semantisch verwandt mit den Anfragesprachen SQL (*Structured Query Language*) und OQL (*Object Query Language*). XQuery benutzt in erster Linie eine SQL bzw. OQL verwandte Nicht-XML-Syntax. Darüber hinaus sieht das W3C eine semantisch äquivalente XML-Syntax vor [W3C03g], die das Einlesen oder das Generieren von XQuery-Anfragen mit Hilfe reiner XML-Werkzeuge ermöglicht. Gegenwärtig ist die Arbeit an XQuery noch nicht abgeschlossen. Die Spezifikation wird vom W3C in einer gemeinsamen Arbeitsgruppe zusammen mit der Nachfolgeversion 2.0 von XSLT [W3C03b] entwickelt.

XQuery ist eine deklarative und funktionale Sprache. Ein XQuery-Programm stellt einen Ausdruck dar, der durch einen XQuery-Prozessor ausgewertet wird und als Ergebnis eine Folge von einfachen (atomaren) Werten oder XML-Fragmenten liefert. Die Berechnung einer XQuery-Anfrage auf einem Eingabe-XML-Dokument bzw. XML-Fragment ist somit eine XML-Transformation.

XQuery besitzt viele Gemeinsamkeiten mit XSLT 2.0, insbesondere das Datenmodell [W3C03d], die Funktions- und Operatorbibliothek [W3C03e] und nicht zuletzt die Pfadsprache XPath 2.0. Tatsächlich lassen sich alle XQuery-Konstrukte ebenfalls durch XSLT ausdrücken [Len01], sodass innerhalb der XML-Gemeinde keine einheitliche Meinung besteht, ob das W3C zwei Spezifikationen entwickeln sollte, die sich in direkter Konkurrenz zueinander befinden [Dod01].

Der Hauptunterschied zwischen XSLT und XQuery besteht in deren Verarbeitungsmodellen. XSLT arbeitet template-basiert, d.h. dass der XSLT-Prozessor für die Menge der aktuell zu bearbeitenden Knoten das jeweils am besten passende Template bestimmt, welches Anweisungen für die Generierung des Ausgabe-XML enthält (näheres dazu in Kapitel 4). Die Ausführung eines XQuery-Programmes dagegen besteht in der Berechnung eines Ausdruckes. Das hat zur Folge, dass in XQuery der Transformationsablauf explizit sichtbar ist und über geeignete Funktionsaufrufe durch den Programmierer gesteuert wird. Der in XSLT eingebaute Auswahlmechanismus der Templates anhand des aktuellen Knotens muss in XQuery explizit über Fallunterscheidungen notiert werden.

XQuery-Anfragen lassen sich damit sehr gut auf datenzentrierten XML-Strukturen formulieren, da in diesem Fall eine bekannte, gleichförmige und regelmäßige Struktur vorliegt. Die Anfrage auf dokumentenzentrierten Daten führt hingegen zu vielen Tests und Fallunterscheidungen und damit potenziell zu sehr komplexen XQuery-Programmen.

Eines der Design-Ziele für XQuery ist deren Optimierbarkeit. Erfahrungen aus dem Datenbankbereich haben gezeigt, dass sich funktionale Anfragesprachen sehr gut optimieren lassen. Unter anderem kann das Wissen über das Schema eines XML-Dokumentes von einem XQuery-Prozessor dazu genutzt werden, Anfragen umzuschreiben und zu vereinfachen. XQuery ist deshalb eine stark typisierte Sprache, die das umfangreiche, durch XML-Schema [W3C01b] definierte Typsystem benutzt.

Mit der Implementierung eines *Streaming XQuery Processor* zeigen Daniela Florescu et al. in [FHK+03], dass XQuery-Anfragen so umgeschrieben werden können, dass sie sich effizient auf einem XML-Datenstrom ausführen lassen. Sowohl die Ein- als auch die Ausgabe werden als so genannter Token-Strom repräsentiert. Die Berechnung der (optimierten) XQuery-Anfrage nutzt das Prinzip der Bedarfsauswertung, indem Token aus dem Eingabestrom erst dann konsumiert werden, wenn sie in der Anfrage benötigt werden. Entsprechend werden berechnete Teilergebnisse sofort in Form eines Token-Stroms ausgegeben. Die Speicheranforderungen für die Berechnung einer XQuery-Anfrage lassen sich auf diese Weise minimieren. Anfragen, die seriell ausgeführt werden können, sind so auf beliebig großen Datenmengen möglich.

Voraussetzung ist jedoch eine intelligente Implementierung des Optimierers, der XQuery-Ausdrücke geeignet umschreibt. Für den Autor einer XQuery-Anfrage gibt es daher keine Gewähr, dass diese mit allen Implementierungen seriell ausgeführt wird. Die Skalierbarkeit ist somit stark implementationsabhängig. Ansonsten berücksichtigt XQuery die verwandten W3C-Standards, insbesondere Infoset und Namensräume. Durch die Benutzung des Typsystems lässt sich die Übereinstimmung mit einem Ergebnisschema sicherstellen.

3.3.4 XML Script

XML Script [XST] ist eine durch Perl inspirierte und auf die XML-Verarbeitung spezialisierte Skriptsprache. Sie operiert konzeptionell auf einem DOM; ihre Syntax ist in XML ausgedrückt. Sowohl der Sprachentwurf als auch eine frei verfügbare Implementation namens *XTract* wurden von der Firma DecisionSoft entwickelt. Die aktuelle Version wurde im Oktober 2002 veröffentlicht.

Die Entwickler von XML Script verweisen als Motivation für ihr Engagement auf die Schwächen von XSLT und dessen Nichteignung für universelle Transformationen. Wie sich gezeigt hat, ist diese Analyse nicht zutreffend. Des Weiteren wird der funktionale und deklarative Charakter von XSLT genannt, der für Entwickler, die ausschließlich prozedurales Programmieren gewohnt sind, eine Hürde darstellt.

Konsequenterweise ist XML Script daher prozedural und imperativ. Der Programmierer ist für den Kontrollfluss verantwortlich. Es existieren Gegenstücke zu aus imperativen Sprachen bekannten Kontrollstrukturen wie `if`, `switch`, `while` oder `for`. Variablen sind wie in XSLT schwach typisiert, lassen sich jedoch ändern (anders als in XSLT).

Ein XML-Script-Prozessor verwaltet intern drei XML-Bäume: einen für die Eingabe, einen für das Skript (das Programm) und einen für die erzeugte Ausgabe. Mittels einer Pfadsprache kann frei auf diese Bäume zugegriffen werden. XML Script stellt dazu die Eigenentwicklung DSLPath sowie seit der letzten Version auch den W3C-Standard XPath zur Verfügung.

Bewertung

XML Script mag vielen Programmierern zunächst vertrauter erscheinen als XSLT. Wie in allen imperativen Sprachen können allerdings auch hier für komplexere Auf-

gaben sehr leicht unübersichtliche und damit schwer wartbare Programme entstehen. XML Script arbeitet auf einem DOM und unterliegt damit den gleichen Beschränkungen hinsichtlich der Dokumentgröße wie andere DOM-Programme. Die Erzeugung von syntaktisch fehlerhaftem XML ist mit XML Script nicht möglich.

3.3.5 XMLTK

Hinter der Bezeichnung XMLTK (*XML Toolkit*) [AGG+02] verbirgt sich eine Sammlung von Werkzeugen, die sich an aus der Unix-Welt bekannten Programmen wie *grep*, *sort*, *tail* etc. orientieren. Die Werkzeuge des XML-Toolkit arbeiten jedoch im Gegensatz zu ihren Vorbildern nicht zeilenbasiert mit Hilfe regulärer Ausdrücke, sondern benutzen spezielle XPath-Ausdrücke[6] zur Identifikation der Knoten in der XML-Struktur der zu verarbeitenden Dateien.

Damit stellen die XMLTK-Werkzeuge keine allgemeine XML-Transformationssprache zur Verfügung, sondern bieten separate Programme bzw. Kommandos, die sich auf der Befehlszeile über ein Pipe-Symbol zu größeren Transformationsaufgaben verbinden lassen. Ziel ist es, einfache, schnelle und skalierbare Werkzeuge zu entwerfen, die elementare Aufgaben der XML-Verarbeitung lösen. Jedes der Werkzeuge verarbeitet die XML-Daten als Strom und kann daher beliebig große XML-Dokumente handhaben.

Zu XMLTK gehören derzeit *xsort* (Sortieren), *xagg* (Datenaggregation), *xnest* (Gruppieren), *xflatten* (Auflösen von Hierarchieebenen), *xdelete* (Löschen), *xpair* (Wiederholung bestimmter Knoten), *xhead* (Beginn eines Dokuments) und *xtail* (Ende eines Dokuments). Jedes dieser Programme lässt sich über Unix-typische Kommandozeilenparameter steuern.

Um eine möglichst große Performance der Verarbeitung zu erreichen, wurde in XMLTK ein effizienter Pattern-Matching-Algorithmus auf einem XML-Datenstrom mit Hilfe deterministischer endlicher Automaten implementiert. Auf diese Weise lässt sich ein konstanter Datendurchsatz erreichen, der insbesondere nicht von der Anzahl der zu überprüfenden Patterns abhängt. Darüber hinaus können mit Hilfe eines speziellen Stream-Index, der Anfang- und Endposition der Eingabeknoten enthält, ganze Unterbäume während des Pattern-Matching überlesen werden, sobald klar ist, dass keines der Patterns auf die Knoten des Unterbaumes passt.

Der für jedes an einer Pipeline beteiligte Werkzeug erforderliche Parse-Schritt kann mit Hilfe eines binären Zwischenformates weitestgehend entfallen. Dieses binäre Format enthält für jeden auftretenden Elementnamen eine eindeutige Nummer und vereinfacht so außerdem Vergleiche zwischen XML-Bezeichnern.

Die Werkzeuge des XMLTK berücksichtigen derzeit keine Namensräume. Sie sind auf die effiziente Ausführung einfacher Transformationen spezialisiert und lassen sich gut miteinander kombinieren. Eine Integration mit anderen XML-Werkzeugen ist nur auf Betriebssystemebene durch den Austausch von XML-Text möglich.

[6] Dabei handelt es sich hier um eine Teilmenge der Pattern-Grammatik aus XSLT, die ihrerseits eine Teilmenge aus XPath beschreibt. In XMLTK wird etwas irreführend ganz allgemein von XPath-Ausdrücken gesprochen, welche jedoch beispielsweise auch arithmetische Ausdrücke beinhalten oder mit Hilfe von Achsen die freie Navigation im Dokument ermöglichen.

3.3.6 fxt

fxt (*Functional XML Transformation Tool*) [BS01] ist eine eng an die funktionale Programmiersprache SML[7] gekoppelte Transformationssprache für XML. Zwar wird für die Beschreibung einfacher Transformationen kein SML-Vorwissen benötigt, jedoch erfordern komplexere Aufgaben die Einbettung und Ausführung von SML-Code. Dies hat auf der einen Seite den Vorteil, dass fxt eine natürliche Schnittstelle zu SML und damit zu einer vollen funktionalen Programmiersprache enthält, auf der anderen Seite aber den Nachteil, dass die Integration in SML-fremde Umgebungen erschwert wird.

XML-Daten werden als generische Baumstruktur ähnlich zum Infoset modelliert. Ein fxt-Programm besteht ähnlich wie XSLT aus einer Menge von Regeln, wobei ein so genanntes *Match-Pattern* bestimmt, für welche Knoten eine Regel angewandt werden kann. In der dazugehörigen Aktion wird der jeweilige Transformationsschritt notiert. fxt-Programme sind XML-Dokumente, allerdings derzeit noch ohne Namensraumunterstützung. Vor der Ausführung der Transformation wird der fxt-Code nach SML übersetzt. Dies ermöglicht die Ausführung einer XML-Transformation direkt aus anderem SML-Code heraus.

fxt zeichnet sich durch spezielle reguläre Match-Patterns aus, vergleichbar mit den regulären Ausdrücken für Zeichenketten. Diese Patterns ermöglichen es, nach maximal zweimaligem Traversieren des Eingabe-Baumes das passende Pattern für alle enthaltenen Knoten zu bestimmen.[8] Laut Aussage der Autoren kann sich fxt in der Verarbeitungsgeschwindigkeit mit ausgereiften XSLT-Implementationen messen lassen.

Bewertung

fxt ist vom Abstraktionsgrad und der Wartbarkeit vergleichbar mit XSLT. Die Einbindung von SML stellt eine zusätzliche Lernschwelle dar, ermöglicht aber die Programmierung beliebig komplexer Algorithmen. Der fxt-Prozessor (bzw. der kompilierte SML-Code) kann die Wohlgeformtheit des Ergebnisses sicherstellen. Da fxt konzeptionell die XML-Eingabe als Baum betrachtet, ist die Größe der transformierbaren XML-Dokumente durch den Hauptspeicher begrenzt. Eine Einschränkung der Pattern-Sprache, um ebenfalls serielle Transformationen zu ermöglichen, ist zwar derzeit angedacht, aber noch nicht realisiert.

3.3.7 XMλ

Mit XMλ [MS99] wurde Ende 1999 eine Sprache entworfen, die allein als theoretischer Entwurf ohne Implementierung existiert. Sie ist deshalb interessant, weil die dort vorgestellten Ideen in die spätere Entwicklung anderer Sprachen für die XML-Verarbeitung eingeflossen sind.

Ausgangspunkt ist die Ansicht der Autoren, dass die Typisierung von XML-Dokumenten mit Hilfe eines Dokumenttyps (repräsentiert durch ein Schema bzw. eine DTD) und die damit mögliche Validierung der XML-Dokumente den Schlüssel zum

[7] Bei SML handelt es sich nicht um eine *Markup Language*, wie die Abkürzung vielleicht vermuten lassen könnte. SML kann zu *Standard ML* bzw. *Standard Meta Language* aufgelöst werden ohne spezielle Beziehung zu XML oder SGML.

[8] Zum Vergleich: die in XSLT verwendeten Patterns und die darin erlaubten XPath-Ausdrücke können beliebig viele Zugriffe (d.h. proportional zur Anzahl der Patterns) auf andere Knoten des Baumes erfordern.

Erfolg von XML darstellt.[9] Allerdings bieten die meisten der (damals) in der Praxis eingesetzten Sprachen oder Methoden für die dynamische Generierung von XML keine Gewähr für die Gültigkeit des Ergebnisses. Eine Validierung muss hier immer als separater Schritt durchgeführt werden.

Konsequenterweise sieht XMλ deshalb die Definition von Datentypen für XML vor, die inhaltlich von den als DTD notierten Dokumenttypen abgeleitet werden können. Solche Datentypen sind *reguläre Typen*, die durch endliche Zustandsautomaten ausgewertet werden können. Eine gültige XML-Instanz ist ein Wert eines solchen XMλ-Typs. Die Berechnung einer Funktion auf diesen Typen entspricht damit einer XML-Transformation, die streng typisiert ausgeführt wird. Verstöße gegen den Ergebnis-Dokumenttyp können bereits statisch als Typfehler festgestellt werden. Darüber hinaus ist die XML-Syntax die natürliche Repräsentation von XML-Daten in XMλ. Eine weitere sprachabhängige Datenrepräsentation für XML wird auf diese Weise vermieden.

XMλ wurde als funktionale Sprache entworfen, die sich syntaktisch an Haskell orientiert. Die XML-Syntax wird also nicht (wie in XSLT oder fxt) zur Notation von Programmen verwendet. Inwieweit der XMλ-Ansatz zur seriellen Verarbeitung großer XML-Datenmengen genutzt werden kann, lässt der Entwurf offen.

3.3.8 CDuce

CDuce [BCF03] wurde von ihren Autoren als »XML-zentrierte, universelle Sprache« entworfen. Das bedeutet, dass es sich um eine vollständige funktionale Sprache mit Eigenschaften wie einem umfangreichen Typsystem, Funktionen höherer Ordnung und Pattern-Matching handelt, die jedoch zusätzlich die Definition eigener Typen für XML-Elemente erlaubt. XML-Daten gehören damit wie in XMλ zu einem eigenen Datentyp und werden nicht, wie bei der Verwendung von XML-APIs in anderen Programmiersprachen, auf Nicht-XML-Typen abgebildet. CDuce baut auf der Sprache XDuce [HP03] auf, die sich allein auf den XML-Aspekt konzentriert und damit nicht für universelle Anwendungen geeignet ist. Eine andere Weiterentwicklung namens Xtatic [GP03] strebt die Anwendung der XDuce-Konzepte in der Programmiersprache C# an.

Variablen und Funktionen, die mit XML-Daten umgehen, sind in CDuce statisch typisiert. Fehlerhafte XML-Instanzen (ungültig im Sinne des Dokumenttyps) werden beim Einlesen zurückgewiesen. Das Generieren von ungültigem XML ist nicht möglich, da bereits der Compiler einen Typfehler melden würde. Da außerdem bereits zur Compile-Zeit der Typ und damit die Struktur der XML-Daten bekannt ist, können Operationen auf den XML-Daten effizienter ausgeführt werden. Von den Autoren durchgeführte Benchmark-Messungen mit dem in C geschriebenen XSLT-Prozessor *xsltproc* zeigten zum Teil erhebliche Geschwindigkeitsvorteile.

CDuce verwendet eine kompakte Nicht-XML-Syntax. XML-Daten werden in der Sprache ebenfalls in einer XML-ähnlichen, jedoch abweichenden Notation repräsentiert, die einen minimalen zusätzlichen Lernaufwand erfordert. CDuce enthält keine

[9] Dem kann entgegengehalten werden, dass sich XML-Dokumente im Gegensatz zu SGML auch ohne Kenntnis eines Dokumenttyps verarbeiten lassen. Eine der Vereinfachungen in XML gegenüber SGML besteht gerade darin, eine Validierung nicht zwingend vorzuschreiben. Dies ermöglicht eine größere Flexibilität und einfachere XML-Werkzeuge für viele unkritische Anwendungen.

Unterstützung für die Verarbeitung von Datenströmen. XML-Daten werden immer als vollständige Struktur behandelt.

3.4 Zusammenfassung

Die in diesem Kapitel diskutierten Transformationsmethoden werden in der folgenden Tabelle noch einmal gegenübergestellt. Die Sprache XMλ wurde nicht berücksichtigt, da für sie keine praktischen Erfahrungen vorliegen. Die Bewertungsskala reicht von ❍ (nicht zutreffend) über ✓ (etwas zutreffend) bis ✓✓✓✓ (sehr stark zutreffend). Das Kriterium Skalierbarkeit wurde hier gesondert hervorgehoben.

	Skalierbarkeit	Abstraktion	Einfachheit	Wartbarkeit	Mächtigkeit	Robustheit
Text	✓✓✓✓	❍	✓	✓✓	✓+	❍
Stream	✓✓✓✓	✓	✓	✓✓	✓✓+	✓
Baum	✓	✓✓	✓✓	✓	✓✓✓✓	✓✓✓
Schema	❍	✓✓	✓✓✓	✓✓✓	✓✓✓✓	✓✓✓✓
Funktional	✓✓	✓✓	✓✓	✓✓✓	✓✓✓✓	✓✓✓
DSSSL	✓✓	✓✓	✓✓	✓✓✓✓	✓✓✓	✓✓
XSLT	✓✓	✓✓✓✓	✓✓✓✓	✓✓✓✓	✓✓✓	✓✓✓
XQuery	✓✓✓	✓✓✓✓	✓✓✓✓	✓✓✓	✓✓✓	✓✓✓✓
XML Script	❍	✓✓✓✓	✓✓✓✓	✓✓✓✓	✓✓✓✓	✓✓
XMLTK	✓✓✓✓	✓✓	✓✓✓	✓✓✓✓	✓✓	✓✓
fxt	✓✓✓	✓✓✓✓	✓✓✓	✓✓✓✓	✓✓✓✓	✓✓
CDuce	✓✓	✓✓✓	✓✓✓	✓✓✓	✓✓✓✓	✓✓✓✓

Skalierbarkeit

Die Methoden, die ein XML-Dokument in Form eines Datenstroms verarbeiten, können naturgemäß sehr gut für große Datenmengen eingesetzt werden und sind deshalb sehr gut skalierbar. Für XQuery und fxt existieren erste Ansätze, wie durch eine geeignete Einschränkung des jeweiligen Sprachumfangs eine serielle Verarbeitung ermöglicht werden kann.

Funktionale Sprachen bieten das Konzept der Bedarfsauswertung von Ausdrücken, das jedoch in der Praxis für die Verarbeitung von XML-Daten derzeit keine Relevanz besitzt. Weder DSSSL noch XSLT verlangen beispielsweise, dass eine Implementierung dieses Konzept unterstützt. Die jeweiligen Sprachspezifikationen, insbesondere die Eigenschaft der Freiheit von Seiteneffekten, lassen die Möglichkeit der Bedarfsauswertung jedoch prinzipiell zu. In der Praxis lesen alle Implementationen das gesamte XML-Dokument vollständig in den Speicher, bevor sie mit der Transformation beginnen. Mit CDuce verhält es sich ähnlich: eine Bedarfsauswertung ist möglich, aber nicht implementiert. Voraussetzung ist jedoch in allen Fällen, dass der Transformationscode geeignet programmiert worden ist, um von den Vorteilen der Bedarfsauswertung profitieren zu können. Insbesondere besteht keine Sicherheit, dass sich der gleiche Code mit allen Interpretern bzw. Compilern gleich verhält.

Die vergleichbare, in baumbasierten APIs realisierte Idee des verzögerten Baumaufbaus bietet keine echte Skalierbarkeit, da dieser Ansatz auf Kosten der Performance geht.

Abstraktion

Bezüglich des Abstraktionsgrades wurden diejenigen Techniken besser bewertet, die XML-Daten in der XML-Syntax oder einer XML-nahen Syntax repräsentieren. Alle Lösungen auf der Basis von APIs müssen hier Datenstrukturen der zugrunde liegenden

Programmiersprache benutzen. Die speziellen Transformationssprachen bieten an dieser Stelle eine bessere Unterstützung.

Einfachheit, Wartbarkeit

Die Kriterien Einfachheit und Wartbarkeit bewertet möglicherweise jeder Entwickler etwas anders, abhängig vom eigenen Wissensstand und der Kenntnis vergleichbarer Techniken. Ganz allgemein sind diejenigen Techniken besser einzuschätzen, die sich stärker auf den XML-Aspekt konzentrieren. Eine moderate Verwendung der XML-Syntax kann das Verständnis erleichtern, während die für eine Reihe von funktionalen Sprachen typische kompakte Syntax für manchen Anfänger eine Hürde darstellt. Bezüglich der Wartbarkeit sind diejenigen Techniken am besten einzuschätzen, die regel- bzw. template-basiert eine Zerlegung der Transformation ermöglichen. Funktionale Sprachen unterstützen von vornherein eine klarere Programmstruktur als imperative Sprachen. Text- und streambasierte Transformationen lassen sich schließlich über Pipelining-Techniken gut miteinander verbinden und daher in einzelne Transformationsaufgaben zerlegen.

Mächtigkeit

Für die Mächtigkeit gilt, dass die Methoden, die die Benutzung einer universellen Programmiersprache ermöglichen, den vollständigen Umfang dieser Sprache nutzen können. DSSSL, XSLT und XQuery erheben nicht den Anspruch, universell einsetzbar zu sein. Text- und streambasierte Methoden sind durch ihren seriellen Charakter zunächst in ihrer Mächtigkeit beschränkt, da sie keine Gesamtsicht auf die Eingabedaten bieten. Jedoch steht es jedem Entwickler frei, diese Grenzen auf Kosten der Skalierbarkeit zu erweitern (gekennzeichnet durch ein + in der Tabelle).

Robustheit

Für das Kriterium Robustheit (im Sinne von Korrektheit des Ergebnisses) lassen sich die folgenden vier Stufen identifizieren: Erkennen der XML-Struktur der Eingabe, XML-Wohlgeformtheit der Ausgabe, Namensraumwohlgeformtheit der Ausgabe und Gültigkeit der Ausgabe. Die Transformationsmethoden mit Schema-Unterstützung bieten hier die größte Sicherheit.

Kapitel 4
Die Transformationssprache XSLT

Die Sprache XSLT (XSL Transformations) entstand als Teil der Stilsprache XSL (Extensible Stylesheet Language) und hat sich zum Mittel der Wahl für die Transformation von XML-Dokumenten entwickelt. Eine Einordnung wurde bereits in Kapitel 3.3.2 gegeben. Da sich eine Vielzahl von Eigenschaften in der neuen Sprache STX wieder finden wird, wird XSLT im Folgenden genauer vorgestellt und charakterisiert.

Ausgangspunkt ist dabei die im Jahr 1999 verabschiedete Version 1.0 [W3C99c]. Derzeit arbeitet das W3C an einer Nachfolgeversion 2.0, die Anfang 2004 noch den Status eines Entwurfs hatte [W3C03b]. Kapitel 4.6 gibt auf die zu erwartenden Änderungen einen Ausblick.

4.1 Charakterisierung

XSLT lässt sich durch die folgenden Eigenschaften charakterisieren:

- **XSLT ist funktional**
  XSLT gleicht semantisch sehr stark DSSSL (siehe auch Kapitel 3.3.1) und besitzt daher Eigenschaften einer funktionalen Programmiersprache. Tatsächlich können die Bausteine eines XSLT-Stylesheet, die so genannten *Templates*, als Funktionen verstanden werden, die für den aktuell betrachteten Knoten der Eingabe einen Teil des Ergebnisbaumes berechnen. Diese Berechnung ist frei von Seiteneffekten (unter anderem können Variablen nach ihrer Initialisierung nicht geändert werden), sodass ein XSLT-Prozessor die aus anderen funktionalen Sprachen bekannten Optimierungen vornehmen kann. So lässt sich beispielsweise die Auswertungsreihenfolge von Ausdrücken, die nicht voneinander abhängen, frei wählen. Variablenwerte müssen erst dann berechnet werden, wenn deren Wert benötigt wird. Der Wert eines Ausdrucks hängt folglich nicht von der Auswertungsreihenfolge ab und spiegelt insbesondere keinen Programmzustand wider.

  XSLT wurde nicht als eine vollständig funktionale Sprache entworfen. So fehlen ihr explizite Funktionen höherer Ordnung. Dennoch wurde durch Dimitre Novatchev gezeigt, dass sich diese mit einer kleinen Erweiterung darstellen lassen [Nov03]. Diese Erweiterung betrifft die Überwindung der in der Version 1.0 von XSLT vorhandenen Unterscheidung zwischen XML-Eingabeknoten und XML-Ergebnisknoten. Ab der Version 2.0 können Knoten des Ergebnisbaumes wie Eingabeknoten behandelt werden (näheres dazu in Kapitel 4.6), sodass die erwähnte Erweiterung nicht mehr notwendig ist. XSLT ist damit tatsächlich eine vollständige funktionale Programmiersprache.

- **XSLT ist deklarativ und arbeitet regelbasiert**
  Die Hauptbestandteile eines XSLT-Stylesheet sind Templates. Ein Template fungiert als Regel, die die Transformation von Knoten des gleichen Typs beschreibt. Welches Template für die aktuell zu bearbeitenden Knoten benutzt werden soll, bestimmt der XSLT-Prozessor. Das wichtigste Kriterium ist ein so genanntes Muster (*pattern*), über das der Typ, der Wert oder die Beziehung zu anderen Knoten beschrieben werden. Daneben lassen sich durch die Definition

von Modi Gruppen von Templates aktivieren bzw. deaktivieren. Mögliche Konflikte können durch die Vergabe von Prioritäten gelöst werden.

- **XSLT ist dokumentorientiert**
  XSLT verarbeitet immer vollständige Dokumente. Da XSLT auf XPath (siehe Kapitel 4.3) aufbaut, benutzt es insbesondere das durch XPath definierte XML-Datenmodell. XPath ermöglicht eine vollständige Sicht auf ein XML-Dokument. Alle aus einem Eingabedokument ablesbaren Informationen stehen in jedem Transformationsschritt zur Verfügung. Während der Transformation kann uneingeschränkt zu allen Knoten navigiert werden.

- **XSLT ist schwach typisiert**
  In XSLT existieren fünf Datentypen: Boolescher Wert, Zahl, Zeichenkette, Knotenmenge und Ergebnisbaumfragment.[1] Während der Transformation werden Werte immer automatisch in den jeweils verlangten Typ umgewandelt. Typfehler treten in XSLT nur dann auf, wenn ein Operand oder eine Funktion eine Knotenmenge verlangt, jedoch ein Wert eines anderen Typs übergeben wurde. Variablen besitzen keinen eigenen Typ und können beliebige Werte aufnehmen.

- **XSLT ist XML**
  Während DSSSL and CSS als Vorläufer von XSL jeweils eine eigene spezielle Syntax verwenden, sind XSLT-Stylesheets wohlgeformte XML-Dokumente. Dies unterscheidet XSLT insbesondere von üblichen Programmiersprachen, die in der Regel sehr kompakte Konstrukte ermöglichen. Ansätze wie XSLScript [XSLS], die eine kompaktere Notation auch für XSLT-Stylesheets vorschlagen, haben bisher keine weite Verbreitung gefunden. Eine eigene Syntax erfordert immer zusätzlichen Lernaufwand, während in XML dargestellte Daten in gewissem Umfang selbsterklärend sind.

  XSLT ist jedoch in erster Linie keine Programmiersprache, sondern eine XML-Transformationssprache. Die zu erzeugende XML-Struktur kann direkt in den Transformationscode eingebettet werden. XSLT benötigt damit keine speziellen Ausgabeanweisungen. Zu generierende XML-Strukturen werden literal als XML notiert, ohne dass eine zusätzliche Abstraktionsebene notwendig ist. Insbesondere steht der gesamte lexikalische Apparat von XML unmittelbar zur Verfügung, inklusive Entity-Referenzen und Zeichenkodierungen. Ein XSLT-Prozessor erkennt XSLT-Anweisungen anhand ihres Namensraumes und kann sie so von literalen XML-Elementen unterscheiden. Die Einbettung von Ausgabe-XML in den XSLT-Code hat darüber hinaus zur Folge, dass im Resultat immer wohlgeformtes XML generiert wird. Verletzungen der Wohlgeformtheit (z.B. ein fehlendes literales End-Tag) werden bereits statisch im Stylesheet erkannt.

  Schließlich sind XSLT-Stylesheets damit ebenfalls XML-Daten, die mit allen XML-Werkzeugen bearbeitet werden können, angefangen von XML-Editoren bis hin zu XSLT selbst. XSLT-Stylesheets können in andere XML-Dokumente eingebettet sein. Sie lassen sich generieren, transformieren, ändern oder analysieren. So ist beispielsweise die aspektorientierte Programmierung, bei der auf Quellcode-Ebene Aspekte in ein Programm hineingewebt werden, mit XSLT

[1] Der Typ Ergebnisbaumfragment (*result tree fragment*) wird in XSLT 2.0 gestrichen, siehe Kapitel 4.6. Teilergebnisse der Transformation sind dann wiederum Knotenmengen. Die in [Nov03] vorgestellte Methode für Funktionen höherer Ordnung beruht auf der Umwandlung eines Ergebnisbaumfragments in eine Knotenmenge durch eine Erweiterungsfunktion.

ohne spezielle Werkzeuge möglich. Ein weiteres sehr populäres Beispiel für die Generierung von XSLT-Code ist die Schemasprache Schematron [Schtrn], deren Implementierung auf der Transformation eines Schematron-Schemas in ein validierendes XSLT-Stylesheet beruht.

Turing-Vollständigkeit

Als funktionale Sprache ist XSLT Turing-vollständig. Dies lässt sich veranschaulichen, indem ein XSLT-Programm eine Turing-Maschine simuliert. Das Turing-Programm selbst (in Form von Zustandsübergangsregeln) wird als XML notiert und dient als Eingabe für XSLT. Das Ergebnis der Transformation ist dann das Ergebnis der Turing-Maschine. Von Bob Lyons stammen sowohl das XML-Vokabular TMML für Turing-Maschinen als auch ein XSLT-Stylesheet, das diese Maschinen »ausführt« [TMML]. In Kapitel 6.1 wird dieses Vokabular noch einmal aufgegriffen, um die Turing-Vollständigkeit von STX zu zeigen.

XSLT ist damit eine Transformationssprache, die theoretisch jede Art von algorithmisch beschreibbaren Transformationsregeln ausführen kann. So lassen sich Stylesheets erstellen, die zum Beispiel Primzahlen berechnen[2] oder das Problem des Springerzuges auf einem Schachbrett lösen (»Knight's Tour Stylesheet« in [Kay00], Seite 613 ff). Dies sind natürlich sehr untypische Anwendungsfälle für XSLT.

4.2 Grundaufbau und Verarbeitungsmodell

Der grundlegende Aufbau eines XSLT-Stylesheet soll anhand der in Beispiel 2 auf Seite 17 skizzierten Transformation vorgestellt werden. Das folgende Listing 3 enthält den dazugehörigen XSLT-Code:

Eine einfache XSLT-Transformation **Listing 3**

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

  <xsl:template match="faq">
    <faq>
      <source>
        <xsl:value-of select="@quelle" />
      </source>
      <xsl:apply-templates />
    </faq>
  </xsl:template>

  <xsl:template match="frage">
    <question>
      <xsl:apply-templates />
    </question>
  </xsl:template>

  <!-- gegebenenfalls weitere Templates für <antwort> etc -->

</xsl:stylesheet>
```

[2] Sieb des Eratosthenes, http://www.informatik.hu-berlin.de/~obecker/XSLT/#eratosthenes

Das Wurzelelement eines XSLT-Stylesheet heißt `xsl:stylesheet`, synonym dazu kann auch `xsl:transform` verwendet werden. Es gehört zum Namensraum *http://www.w3.org/1999/XSL/Transform*. Alle XML-Elemente aus diesem Namensraum (erkennbar an dem Präfix »`xsl:`«) werden als XSLT-Anweisungen interpretiert.

Templates

Die wichtigsten Bestandteile eines Stylesheet sind die Transformationsregeln (Templates). Dieses Stylesheet enthält nur zwei Templates. Das erste (Zeile 4) beschreibt eine Transformationsregel für `faq`-Elemente, das zweite (Zeile 13) eine Regel für `frage`-Elemente. Für welche Knoten ein Template verantwortlich ist, wird durch ein Muster über das `match`-Attribut festgelegt.

Der Inhalt eines Template kann aus weiteren XSLT-Anweisungen oder literalen Elementen bestehen. Letztere werden einfach in die Ausgabe kopiert. Ein Template fungiert damit im wörtlichen Sinne als Schablone, da es ein Stück der zu generierenden XML-Daten beschreibt.

In diesem Beispiel wird für ein Eingabe-`faq`-Element ebenfalls ein `faq`-Element in der Ausgabe erzeugt. Darunter wird ein Kindelement `source` generiert, dessen Inhalt sich aus dem Wert des `quelle`-Attributs von `faq` ergibt. Schließlich sorgt die Anweisung `xsl:apply-templates` dafür, dass die Verarbeitung der Eingabe mit den Kindknoten des `faq`-Elements fortgesetzt wird.

Der XSLT-Prozessor bestimmt anhand der Muster selbstständig, welches Template anzuwenden ist. Der Programmablauf erfolgt eingabegetrieben durch das zu verarbeitende XML-Dokument. Die Anweisung `xsl:apply-templates` wählt eine neue Menge von Knoten des Eingabedokuments aus, für die der XSLT-Prozessor anschließend geeignete Templates finden muss. Ist, wie in diesem Beispiel, kein `select`-Attribut angegeben, werden durch `xsl:apply-templates` die Kindknoten ausgewählt. Prinzipiell kann jedoch jeder beliebige Knoten des Eingabebaumes über einen XPath-Ausdruck ausgewählt und mit Hilfe des Attributs `select` der Anweisung `xsl:apply-templates` übergeben werden.

Prioritäten

Es kann der Fall eintreten, dass die Muster verschiedener Templates auf den selben zu verarbeitenden Knoten passen. In diesem Fall wird das Template mit der höheren Priorität ausgewählt. Prioritäten werden automatisch durch den XSLT-Prozessor bestimmt, wobei – vereinfacht gesagt – ein spezifischeres Muster eine höhere Priorität erhält.[3] Mit Hilfe des Attributs `priority` kann einem Template alternativ eine explizite Priorität zugewiesen werden.

Modi

Darüber hinaus kann jedes Template mit Hilfe eines `mode`-Attributs einem Modus zugeordnet werden. Alle Templates des gleichen Modus bilden damit implizit eine Gruppe. Über das gleichnamige Attribut `mode` in `xsl:apply-templates` wird der angegebene Modus ausgewählt. Nur die Templates des gleichen Modus stehen für die Verarbeitung der ausgewählten Knoten zur Verfügung. Die Verwendung von Modi ermöglicht das mehrfache Traversieren der Eingabe unter verschiedenen Gesichtspunkten. So kann beispielsweise ein Modus für die Erstellung eines Inhaltsverzeichnisses zuständig sein, während ein anderer Modus den Inhalt formatiert.

Vorgabe-Templates

Existiert kein passendes Template im Stylesheet, wendet der XSLT-Prozessor abhängig vom Knotentyp spezielle Vorgaberegeln an. So werden Textknoten automatisch in die Ausgabe kopiert. Diese Regel sorgt dafür, dass sich der gesamte Textinhalt des

[3] Die XSLT-Spezifikation legt vier Prioritätsgruppen fest und ordnet jedem Template automatisch einen der Prioritätswerte -0.5, -0.25, 0 und 0.5 zu. Insbesondere werden alle zusammengesetzten Muster mit dem gleichen Wert 0.5 bewertet, sodass für diese keine automatische Differenzierung stattfindet.

Beispiels ebenfalls in der Ausgabe wieder findet. Zur Vermeidung dieses Effekts muss man entweder die Verarbeitung der entsprechenden Textknoten verhindern oder für diese ein spezielles Template aufnehmen, das keine Ausgabe produziert. Die Vorgaberegel für Elementknoten setzt die Verarbeitung mit den Kindknoten fort, ohne dass das Element selbst kopiert wird. Auf diese Weise wird hier das Markup für `begriff` wie gewünscht aus der Eingabe entfernt. Die Vorgabe-Templates für Kommentare und Verarbeitungsanweisungen sind leer.

Diese etwas willkürlich anmutenden unterschiedlichen Regeln für Textknoten, Elemente und Kommentare bzw. Verarbeitungsanweisungen beruhen darauf, dass XSLT ursprünglich allein als Transformationsteil der Stilsprache XSL konzipiert war. Ein leeres Stylesheet ohne Regeln reproduziert allein den Textinhalt des Eingabedokuments und entfernt jegliches Markup.

4.3 XPath

XSLT bedient sich zur Navigation innerhalb des XML-Eingabedokuments der Sprache XPath. Diese beinhaltet zwar ebenfalls logische und arithmetische Ausdrücke, den wichtigsten Bestandteil stellen jedoch die Pfadausdrücke dar. Mit ihnen kann auf jeden beliebigen Knoten des Eingabedokumentes zugegriffen werden. Das Beispiel in Listing 3 auf Seite 43 enthält nur den Pfadausdruck `@quelle` (Zeile 7), der auf das `quelle`-Attribut zugreift. Ausgangspunkt eines solchen Pfades ist dabei der so genannte *Kontextknoten*, der in diesem Beispiel ein `faq`-Element ist. Dies ist der Knoten, für den das Template ausgeführt wird. Der Aufbau von Pfadausdrücken wird im Folgenden genauer vorgestellt.

Ein Pfad kann aus mehreren Schritten zusammengesetzt werden, die durch Schrägstriche (*slashes*) voneinander getrennt sind. Jeder Schritt wiederum besteht aus einer optionalen Achse, einem Knotentest und einer ebenfalls optionalen Liste von Prädikaten:

```
Achse::Knotentest Prädikate
```

Die *Achse* gibt an, in welche Richtung innerhalb des Eingabebaumes navigiert werden soll. Mit Hilfe des *Knotentests* werden diverse Knotentypen unterschieden. Die Liste der *Prädikate* dient schließlich dem Test weiterer Eigenschaften, denen die ausgewählten Knoten genügen müssen.

XPath definiert 13 verschiedene Achsen:

Achsen

Tabelle 1

XPath-Achsen

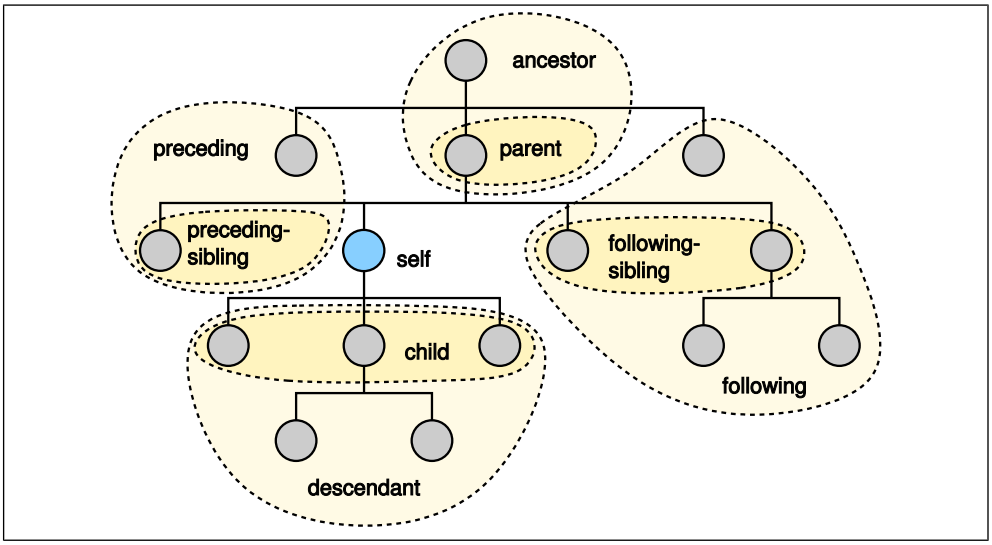
Achse	Bedeutung
`child`	Kindknoten
`parent`	Elternknoten
`descendant`	Nachkommen (Kinder, Enkel, usw.)
`ancestor`	Vorfahren (Eltern, Großeltern, usw.)
`preceding`	Knoten, die dem Kontextknoten vorangehen
`following`	Knoten, die dem Kontextknoten folgen
`preceding-sibling`	vorherige Geschwister
`following-sibling`	nachfolgende Geschwister
`descendant-or-self`	Nachkommen plus Kontextknoten
`ancestor-or-self`	Vorfahren plus Kontextknoten
`self`	der Kontextknoten selbst

`attribute`	Attribute
`namespace`	Namensräume

Die letzten beiden Achsen nehmen eine Sonderrolle ein, da über sie nur spezielle Knotentypen ausgewählt werden: Attributknoten bzw. Namensraumknoten. Alle anderen Achsen (von `self` abgesehen) enthalten niemals Knoten dieser beiden Knotentypen. Abbildung 2 zeigt 9 der 13 Achsen. Die jeweils ausgewählte Knotenmenge ist durch eine gestrichelte Linie umrandet. Der Kontextknoten ist im Bild durch die Achse `self` gekennzeichnet.

Abbildung 2

Einige XPath-Achsen



Die Achse `child` ist die Vorgabeachse. Sie kann daher inklusive der beiden Doppelpunkte weggelassen werden. Die Attributachse lässt sich durch das Zeichen `@` abkürzen. Der genannte Beispielpfad `@quelle` steht damit für `attribute::quelle`.

Knotentests

Ein Knotentest bestimmt den gewünschten Knotentyp. Neben den bereits genannten Attribut- und Namensraumknoten unterscheidet das XPath-Datenmodell fünf weitere Knotentypen, die im Folgenden aufgelistet werden:

Tabelle 2

XPath-Knotentests

Knotentyp	**Knotentest**
Element	`*` ein beliebiges Element `foo` ein `foo`-Element. `ns:*` ein Element aus dem durch `ns` bezeichneten Namensraum `ns:foo` ein `foo`-Element aus dem durch `ns` bezeichneten Namensraum
Textknoten	`text()`
Kommentarknoten	`comment()`
Verarbeitungsanweisung	`processing-instruction()` `processing-instruction(`*`String`*`)`
Dokumentknoten	kein spezieller Knotentest

Auf der Attribut- und Namensraumachse würden `*`, `foo`, `ns:*` und `ns:foo` Attribut- bzw. Namensraumknoten mit den entsprechenden Namen auswählen.[4] Den Dokumentknoten erreicht man über den Pfad `/` (ein einzelner Schrägstrich). Ein Pfad, der mit einem Schrägstrich beginnt, wird absoluter Pfad genannt und immer vom Dokumentknoten her ausgewertet. Der Knotentest `node()` passt schließlich auf alle Knoten unabhängig von ihrem Typ.

Möchte man beispielsweise im Template für `frage` auf das `quelle`-Attribut von `faq` zugreifen, so würde der Pfad `parent::faq/attribute::quelle` zum Ziel führen. Praktischerweise kennt XPath neben `@` weitere Abkürzungen, sodass sich als wesentlich kompakterer Ausdruck `../@quelle` notieren lässt. Die Abkürzung `..` steht dabei für `parent::node()`, also für den Elternknoten unabhängig von seinem Typ. Ein einzelner Punkt `.` bezeichnet den Kontextknoten selbst (`self::node()`). Die Nachkommen des Kontextknotens lassen sich über die Kurzschreibweise `//` erreichen, welche für `/descendant-or-self::node()/` steht.

Abkürzungen

In jedem Schritt kann schließlich eine Liste von Prädikaten angegeben werden, die die ausgewählten Knoten weiter selektieren. Ein einzelnes Prädikat besteht dabei aus einem in eckigen Klammern eingeschlossenen XPath-Ausdruck. Für jeden durch den Schritt als Kontextknoten ausgewählten Knoten werden die Ausdrücke in den Prädikaten ausgewertet. Liefert ein Ausdruck den Booleschen Wert *falsch*, wird der entsprechende Knoten aus der Menge entfernt.

Prädikate

Wenn das Dokument mit dem Wurzelelement `faq` in der Realität ein Fragment eines größeren Dokuments mit vielen Fragen und Antworten ist, so würde

```
//faq[antwort/name='Rick Jelliffe']
```

nur die `faq`-Elemente auswählen, die von Rick Jelliffe beantwortet wurden. Dieser Pfad beginnt mit einem Schrägstrich und untersucht daher das gesamte Dokument. Sollen außerdem nur seine Antworten aus dem August des Jahres 2000 betrachtet werden, würde folgender Ausdruck zum Ziel führen

```
//faq[antwort/name='Rick Jelliffe']
     [contains(@quelle,'/200008/')]
```

Die Funktion `contains` liefert den Wert *wahr*, wenn die Zeichenkette des zweiten Parameters im ersten Parameter enthalten ist. Da hier beide Prädikate unabhängig voneinander erfüllt sein müssen, können deren Bedingungen auch innerhalb eines Prädikats miteinander verknüpft werden. Ein äquivalenter Pfadausdruck lautet daher

```
//faq[antwort/name='Rick Jelliffe' and
      contains(@quelle,'/200008/')]
```

Jeder Knoten besitzt eine Position innerhalb der Liste der ausgewählten Knoten. In Schritten, die eine vorwärts gerichtete Achse enthalten, werden diese Knoten entsprechend der Originalreihenfolge im Dokument nummeriert. Eine vorwärts gerichtete Achse enthält dabei nur Knoten, die dem Kontextknoten im Dokument folgen. Enthält der Schritt dagegen eine rückwärts gerichtete Achse, dreht sich diese Reihenfolge um. Der Elternknoten ist somit der erste Knoten auf der Achse `ancestor`; der direkte Vorgänger ist der erste Knoten auf der Achse `preceding-sibling` usw.

Positionen

Die Position kann mit Hilfe der Funktion `position` abgefragt werden:

```
antwort/absatz[position()=2]
```

[4] Dabei ist anzumerken, dass Namensraumknoten selbst keinen qualifizierten Namen besitzen. Das bedeutet, dass `namespace::ns:*` und `namespace::ns:foo` niemals einen Knoten auswählen.

würde den zweiten Absatz der Antwort auswählen. Dagegen würde

```
//absatz[position()=2]
```

alle Absätze auswählen, die an zweiter Stelle innerhalb ihres Elternelements vorkommen. Ergibt die Berechnung des Ausdrucks in einem Prädikat eine Zahl, so ist das Prädikat dann erfüllt, wenn diese Zahl mit der aktuellen Position übereinstimmt. Der letzte Ausdruck kann daher kürzer als

```
//absatz[2]
```

notiert werden. Durch vorangehende Prädikate ändern sich in der Regel auch die Positionen der betrachteten Knoten. Dazu ebenfalls zwei Beispiele:

```
//faq[antwort/name='Rick Jelliffe'][2]
```

liefert das zweite der durch Rick Jelliffe beantworteten `faq`-Elementen. Dagegen wählt

```
//faq[2][antwort/name='Rick Jelliffe']
```

zunächst das zweite `faq`-Element aus und testet anschließend, ob dieses ebenfalls die folgende Bedingung erfüllt.

Es ist leicht einzusehen, dass eine Liste von Prädikaten, in der höchstens im ersten Prädikat auf die Position zugegriffen wird, äquivalent durch ein einziges Prädikat ausgedrückt werden kann, in dem die einzelnen Ausdrücke mit einem logischen *und* verbunden werden. Das letzte Beispiel wird damit zu

```
//faq[position()=2 and antwort/name='Rick Jelliffe']
```

Muster

Muster (*patterns*) sind spezielle XPath-Pfadausdrücke. Sie werden in XSLT für die Auswahl eines geeigneten Template mit Hilfe des so genannten *Pattern-Matching* benutzt. Muster sind deshalb in der XSLT-Spezifikation [W3C99c] und nicht in XPath [W3C99b] beschrieben.

Muster unterscheiden sich syntaktisch von vollen Pfaden dadurch, dass in ihnen explizit nur die beiden Achsen `child` und `attribute` verwendet werden können. Daneben sind jedoch alle Abkürzungen inklusive des doppelten Schrägstriches (`//`) erlaubt. Innerhalb von Prädikaten können volle XPath-Ausdrücke benutzt werden, mit der Ausnahme, dass diese keine Variablen enthalten dürfen.

Ein Muster passt dann auf einen Knoten, wenn es einen anderen Knoten im Dokument gibt, von dem als gedachter Kontextknoten aus das Muster den betrachteten Knoten auswählen würde. Diese etwas komplizierte Definition beschreibt nur die Semantik, nicht jedoch die Implementation. Tatsächlich kann man ein Muster immer von rechts nach links lesen und im Eingabebaum geeignete Knoten auf der Vorfahrenachse suchen.

4.4 Speicheranforderungen

Die volle Unterstützung des XPath-Datenmodells bedingt, dass alle gängigen XSLT-Prozessoren das gesamte Eingabedokument vor Beginn der eigentlichen Transformation in den Speicher einlesen und intern als Baum repräsentieren. Dagegen kann der im Ergebnis der Transformation entstehende Baum unmittelbar serialisiert werden, sodass eine interne Baum-Repräsentation des Ergebnisses nicht notwendig ist. Das Ziel, XSLT-Transformationen allein durch statische Analyse des Stylesheet möglichst seriell auszuführen, gilt innerhalb der XSLT-Gemeinde als »heiliger Gral« [Kay01].

Allerdings sind die theoretischen Möglichkeiten der Analyse begrenzt, sodass das Hauptaugenmerk bei der Entwicklung von XSLT-Prozessoren auf der Verbesserung der Transformationsgeschwindigkeit und der effizienten Speicherverwaltung liegt.

Im Test: Saxon und Xalan

Exemplarisch wurden im Folgenden die beiden sehr weit verbreiteten XSLT-Prozessoren Saxon 7.9.1 [Saxon] und Xalan 2.6.0 [ASFb] untersucht. Beide sind in Java implementiert. Dies ist insofern von Bedeutung, als in Java der zur Verfügung stehende Speicher nicht allein durch das Betriebssystem und den von ihm verwalteten virtuellen Speicher begrenzt wird, sondern bereits durch die Java-Laufzeitumgebung (die virtuelle Maschine JVM). In der Standardeinstellung liegt die maximale Speichernutzung bei 64 MB. Der folgende Test ist dahingehend plattform- und hardwareunabhängig.

Testeingabe

Als Eingabedokument wurden generierte XML-Dateien verwendet, deren Baumtiefe vier Ebenen und deren maximale Textknotengröße 132 Zeichen beträgt. Die einzelnen XML-Dateien unterscheiden sich nur durch die Anzahl der Unterelemente in der zweiten Ebene (d.h. direkt unterhalb des Wurzelelements). Diese Unterelemente besitzen den gleichen Aufbau. Die Testdaten enthalten leere Elemente, Elemente mit gemischtem Inhalt, Attribute, einen zusätzlichen Namensraum sowie XML-Kommentare.

Test-Stylesheets

Für die Ermittlung der Speichergrenzen kamen zwei sehr einfache Stylesheets zum Einsatz. Das erste Stylesheet T_1 enthält allein das folgende leere Template

```
<xsl:template match="text()" />
```

Dieses Template wird für Textknoten aufgerufen und führt keine Operation aus. Für alle anderen Knoten werden die eingebauten Vorgabe-Templates verwendet. Im Ergebnis der Transformation wird also keine Ausgabe erzeugt. Das zweite Stylesheet T_2 bestimmt mit Hilfe der beiden XPath-Ausdrücke

```
count(//node())
string-length(/)
```

die Gesamtzahl der Knoten im Dokument (ohne Attribute) sowie die Gesamtlänge des enthaltenen Textes. Beide Stylesheets könnten prinzipiell seriell ausgeführt werden.

Testergebnis

Unter Verwendung der Standardeinstellungen von Java 1.4.1 konnten die folgenden Maximalwerte ermittelt werden:

		Dokumentgröße	Anzahl Knoten	Textlänge
Saxon	T_1	ca. 11,4 MB	ca. 690.000	ca. 8.190.000
	T_2	ca. 9,4 MB	ca. 570.000	ca. 6.750.000
Xalan	$T_{1/2}$	ca. 16,8 MB	ca. 1.020.000	ca. 12.500.000

Tabelle 3

Maximale Dokumentgrößen in XSLT

Bei Xalan ergeben sich für beide Stylesheets die gleichen Werte. Zwar ist Xalan in der Lage, größere XML-Dokumente zu bewältigen, allerdings benötigt es für gleiche Dokumentgrößen etwa doppelt soviel Zeit wie Saxon. Die konkreten Zeitwerte wurden hier nicht angegeben. Da Java in der Grundeinstellung jeweils 64 MB zur Verfügung standen, ergibt sich ein Faktor von etwa 4 bis 6 für die Größe des notwendigen Hauptspeichers im Verhältnis zur Größe des XML-Dokuments. Der Speicherbedarf eines XSLT-Prozessors verhält sich somit immer proportional zur Größe der zu transformierenden Eingabedaten. Dies kann nicht nur bei einzelnen sehr großen Dokumenten die verfügbaren Ressourcen überschreiten, sondern ebenso bei mehreren parallelen XSLT-Prozessen zu massiven Speicheranforderungen führen.

Wie das Beispiel belegt, sind weder Saxon noch Xalan in der Lage zu erkennen, dass selbst für das sehr einfache Stylesheet T_1 der Aufbau einer Baumstruktur nicht notwendig ist. Sollten zukünftige XSLT-Implementierungen Fortschritte in diese Richtung machen, bleibt eine konkrete Zusicherung immer an die jeweilige Implementation gebunden. Für den Autor eines XSLT-Stylesheet gibt es demnach keine Gewähr, dass eine konkrete Transformation implementationsunabhängig seriell ausgeführt werden kann.

`saxon:preview`

Ein weiterer möglicher Ausweg sind implementationsspezifische Erweiterungen, die ebenfalls an das benutzte Produkt gebunden sind. Solche Erweiterungselemente müssen zu einem separaten Namensraum gehören und können gegebenenfalls von anderen XSLT-Prozessoren ignoriert werden. So unterstützt beispielsweise Saxon [Saxon] bis zur Version 6.5.3[5] ein spezielles Element `saxon:preview`, das es ermöglicht, die dort spezifizierten Elemente bereits während des Baumaufbaus zu transformieren. Nachdem ein solches Element verarbeitet wurde, wird sein gesamter Inhalt inklusive aller Unterelemente verworfen. Die Anweisung `saxon:preview` erzwingt damit eine serielle Verarbeitung für bestimmte Teile der Eingabe, unabhängig davon, ob die gegebene XSLT-Transformation dafür geeignet ist. Im ungünstigen Fall entspricht das berechnete Ergebnis nicht dem tatsächlich zu erwartenden Transformationsergebnis. Die Erweiterung `saxon:preview` kann daher nicht als zuverlässige Lösung angesehen werden. In der derzeit aktuellen Entwicklungslinie 7 ist `saxon:preview` unter anderem deshalb nicht mehr enthalten.

4.5 Probleme mit XSLT

Neben der besprochenen Beschränkung in der Größe der zu transformierenden Eingabe gibt es weitere Anwendungsfälle, für die XSLT kein geeignetes Mittel darstellt.

Datenströme

Als erstes ist hier die Transformation eines kontinuierlichen Datenstroms zu nennen. In diesem Fall sollen die ersten Teile des Ergebnisses möglichst unmittelbar nach dem Einlesen der ersten Daten bereitgestellt werden. Vorstellbar ist beispielsweise, dass ein umfangreiches Ergebnis auf eine Anfrage von einem entfernten Server in XML ausgeliefert wird und für die Anzeige nach HTML transformiert werden soll. Die Übermittlung könnte einige Zeit in Anspruch nehmen. Der Nutzer soll jedoch den Beginn der HTML-Seite bereits lesen können, obwohl das Ende des XML-Datenstroms noch nicht eingetroffen ist. In einem anderen Szenario könnte ein theoretisch unendlicher Datenstrom aus XML-Fragmenten[6] zur Übermittlung von Messdaten etc. benutzt werden, welche dann auf dem Weg der Transformation in andere XML-Vokabulare umgewandelt werden müssen.

Schleifen

XSLT als funktionale Sprache enthält keine Schleifenkonstrukte. Die für viele Anwender vertrauten Iterationen müssen in XSLT durch rekursive Aufrufe realisiert werden. Dies ist an sich kein Nachteil von XSLT, kann jedoch eine kleine Hürde für weniger erfahrene Anwender darstellen. Im Folgenden werden zwei mögliche Vereinfachungen kurz vorgestellt.

[5] Kurz zu Saxons Versionsnummern: Die Version 6.5.3 ist die letzte und stabilste Version der 6er-Linie, welche XSLT 1.0 implementiert. Mit Beginn der 7er-Linie implementiert Saxon den Entwurf der XSLT-2.0-Spezifikation. Für den obigen Speichertest wurde die letzte Version 7.9.1 verwendet. Aufgrund ihrer Instabilität rät der Autor derzeit jedoch davon ab, diese Version in Produktionsumgebungen einzusetzen.

[6] Die Baumstruktur von XML sieht kontinuierliche Datenströme nicht vor. Zwar lassen sich mit XSLT die einzelnen Fragmente transformieren, jedoch muss die Zerlegung in diese Fragmente außerhalb von XSLT geschehen.

Die erste Variante greift wieder auf Erweiterungen des bereits vorgestellten XSLT-Prozessors Saxon zurück. Dieser bietet mit `saxon:while` eine Schleifenanweisung und mit `saxon:assign` eine Anweisung zum Ändern von Variablen an. Eine solche Variable muss zuvor als änderbar deklariert worden sein, siehe Listing 4.

Schleifenkonstrukte in Saxon **Listing 4**

```
<xsl:variable name="i" select="0" saxon:assignable="yes" />
<saxon:while test="$i &lt; 10">
  Der Wert von i ist <xsl:value-of select="$i"/>
  <saxon:assign name="i" select="$i+1" />
</saxon:while>
```

Ein solches Stylesheet, das Erweiterungselemente benutzt, ist jedoch Saxon-spezifisch und daher nicht portabel.

Die zweite Möglichkeit führt typische Schleifenkonstrukte auf Rekursionen in XSLT zurück, indem der dafür notwendige Code automatisch generiert wird. Das eigentliche XSLT-Stylesheet entsteht damit selbst auf dem Weg einer Transformation aus einem »angereicherten« Stylesheet. Im Unterschied zu Saxons Erweiterungselementen werden die zusätzlichen Anweisungen jedoch nicht durch den XSLT-Prozessor selbst interpretiert, sondern durch reine XSLT-Konstrukte ersetzt. Der vom Verfasser entwickelte »Loop-Compiler«[7] ist ein XSLT-Stylesheet, das eine solche Transformation vornimmt. Den vom Anwender zu erstellenden Code demonstriert Listing 5.

Erweiterter XSLT-Code für den Loop-Compiler **Listing 5**

```
<xsl:variable name="i" select="0" />
<loop:while test="$i &lt; 10">
  <loop:do>
    Der Wert von i ist <xsl:value-of select="$i" />
  </loop:do>
  <loop:update name="i" select="$i+1" />
</loop:while>
```

Der Einsatz dieser Methode bedeutet jedoch einen zusätzlichen Verarbeitungsschritt vor der Verwendung des resultierenden XSLT-Stylesheet.

Änderbare Variablen

Eine grundlegende Eigenschaft vieler funktionaler Sprachen und auch von XSLT ist, dass Variablen nach ihrer Initialisierung nicht geändert werden können. Daraus ergibt sich, dass Zwischenergebnisse nicht mitgeführt und aktualisiert werden können, sondern der benötigte Wert an Ort und Stelle aus den zur Verfügung stehenden Eingabedaten (neu) berechnet werden muss. Clevere Optimierungsstrategien im XSLT-Prozessor können jedoch dafür sorgen, dass gleiche Ausdrücke nicht tatsächlich mehrfach berechnet werden.

Berechnungskomplexität

Prinzipiell führt das mehrfache Berechnen der gleichen Werte jedoch zu einer höheren Komplexitätsstufe. Das folgende reale Beispiel aus dem DDD-Projekt[8] benötigt beispielsweise eine zusätzliche Annotierung eines in XML ausgezeichneten Textes, in

[7] Siehe http://www.informatik.hu-berlin.de/~obecker/XSLT/loop-compiler/

[8] Siehe http://www.linguistik.hu-berlin.de/ddd/

der für jeden Text- und Elementknoten die Anfangsposition bezüglich des reinen Textinhalts bestimmt und hinzugefügt wird. Die Eingabe

```
<a>Das ist ein <b>Beispiel</b> dafür.</a>
```

würde zum Beispiel folgendermaßen annotiert werden

```
<a start="0"><text start="0">Das ist ein </text><b start="12"><text
start="12">Beispiel</text></b><text start="20"> dafür.</text></a>
```

Eine nahe liegende Lösung für dieses Problem besteht darin, das Ursprungsdokument einmal durchzugehen und die Länge des bereits überlesenen Textes in einer Variablen mitzuführen. Der Berechnungsaufwand steigt hier offenbar linear mit der Länge der Eingabe.

Ein solcher Algorithmus lässt sich allerdings nicht ohne weiteres in XSLT realisieren. Ein Mitführen der Länge des bereits gelesenen Textes ist in XSLT nicht möglich (von Erweiterungen wie `saxon:assign` einmal abgesehen). Der XSLT-typische Ansatz besteht darin, die Anfangsposition direkt aus den vorangehenden Knoten zu bestimmen. In diesem Fall muss die Summe der Längen aller Textknoten auf der Achse `preceding` bestimmt werden. Da hier für jede neue Position alle vorangehenden Knoten erneut besucht werden müssen, verhält sich dieser Algorithmus in seiner Laufzeit quadratisch zur Länge der Eingabe.

Ein geeigneter Algorithmus mit linearer Laufzeit müsste in XSLT wieder rekursiv programmiert werden, wobei die jeweiligen aktuellen Positionswerte über Parameter mitgegeben werden. Allerdings bedeutet dies, das Traversieren der Eingabe, das schrittweise Durchgehen des Baumes von Knoten zu Knoten in der richtigen Reihenfolge, ebenfalls durch rekursive Aufrufe zu realisieren. Der durch den XSLT-Prozessor bereitgestellte Mechanismus mittels `xsl:apply-templates` kann hier nicht genutzt werden.

4.6 Ausblick auf XSLT 2.0

Seit Dezember 2001 arbeitet das W3C an den Nachfolgeversionen 2.0 für XSLT [W3C03b] und XPath [W3C03a]. Beide Spezifikationen werden in Gemeinschaft mit der XML-Anfragesprache XQuery [W3C03c] (siehe Kapitel 3.3.3) entwickelt und besaßen Anfang 2004 den Status eines finalen Arbeitsentwurfs.[9] Einige der neuen Konzepte aus XSLT/XPath 2.0 werden sich auch in STX wieder finden. In diesem Kapitel werden die wesentlichen Aspekte beider Spezifikationen kurz vorgestellt.

XPath 2.0

Sequenztyp

Das Datenmodell von XPath 2.0 führt die *Sequenz* als neuen grundlegenden Datentyp ein. Eine Sequenz ist eine Folge von einfachen Werten bzw. Knoten, die durch Kommata voneinander getrennt notiert werden. Dabei besteht kein Unterschied zwischen einem einzelnen Wert und einer einelementigen Sequenz. Beide können nicht

[9] engl.: »*Working Draft in Last Call*«. Dies ist die letzte Stufe vor einem Empfehlungskandidaten (*Candidate Recommendation*). Die mit der Entwicklung einer Spezifikation beauftragte W3C-Arbeitsgruppe zeigt mit der Bezeichnung als finaler Arbeitsentwurf an, dass aus ihrer Sicht alle Anforderungen erfüllt sind. Sie erwartet nun Kommentare der Öffentlichkeit und anderer W3C-Arbeitsgruppen. Auf dieser Stufe können letztmalig inhaltliche Änderungen vorgenommen werden.

voneinander unterschieden werden. Verschachtelte Sequenzen gibt es nicht; solche Konstruktionen werden immer in flache Sequenzen aufgelöst. Sequenzen verallgemeinern damit den aus XPath 1.0 bekannten Typ der Knotenmenge auf die einfachen Datentypen. Eine leere Sequenz entspricht der leeren Knotenmenge.

Sequenzen sind ein grundlegender strukturierter Datentyp für atomare Werte. In XPath 1.0 gibt es keine Möglichkeit, eine »Sammlung« einfacher Werte als einzelnen Wert zu behandeln, ihn beispielsweise einer Variablen zuzuweisen. Diese Schwäche wird in XPath 2.0 beseitigt. Der Sequenztyp wird sich in der in dieser Arbeit entwickelten Transformationssprache STX wieder finden.

Für Sequenzen gibt es in XPath 2.0 einige neue Operatoren: den Bereichsoperator `to`, den Schleifenoperator `for $i in` *`Seq`* `return` *`Wert`*, der aus der Sequenz *`Seq`* eine neue Sequenz konstruiert, sowie die Quantifizierungsoperatoren `some` (entspricht dem mathematischen ∃) und `every` (entspricht ∀).

Typisierung

Die zweite wesentliche Änderung neben der Einführung von Sequenzen betrifft die strenge Typisierung. In XPath 1.0 gibt es neben dem Knotentyp nur drei einfache Datentypen: Boolesche Werte, Zahlen und Zeichenketten. Werte dieser Typen werden bei Bedarf automatisch in den erforderlichen Typ umgewandelt. Typfehler können somit nicht auftreten (von der nicht möglichen Umwandlung eines einfachen Wertes in einen Knoten einmal abgesehen).

Das Typsystem von XPath 2.0 basiert stattdessen auf dem Typsystem von XML Schema [W3C01b] und ergänzt dieses um fünf weitere Datentypen. Automatische Typumwandlungen werden nun nur noch zwischen kompatiblen Typen vorgenommen, d.h. wenn der bereitgestellte Typ ein Subtyp des benötigten Typs ist. Damit sind insbesondere die sehr häufig in XPath 1.0 auftretenden impliziten Umwandlungen zwischen Zahlen und Zeichenketten in XPath 2.0 nicht mehr möglich. Stattdessen muss nun eine explizite Umwandlung mit Hilfe der entsprechenden Konvertierungsfunktionen vorgenommen werden. XPath 2.0 ist damit auf der einen Seite typsicherer, verlangt jedoch auf der anderen Seite viele explizite Typumwandlungen und somit häufig kompliziertere Ausdrücke.

Bedingte Ausdrücke

Ein weiteres neues XPath-Konstrukt sind bedingte Ausdrücke. Diese lassen sich in XPath 1.0 nur über komplexe Konstruktionen mit Zeichenkettenfunktionen realisieren.[10] In XPath 2.0 können bedingte Ausdrücke nun direkt als `if (`*`Bedingung`*`) then` *`Wert`*$_1$ `else` *`Wert`*$_2$ notiert werden.

Namensraum-Achse

Schließlich gilt die Achse `namespace` in XPath 2.0 als veraltet (*deprecated*). Stattdessen bieten zusätzliche Funktionen den Zugriff auf die Informationen über aktive Namensräume. Dies soll effizientere XPath-Implementierungen ermöglichen, da für Namensräume nun keine eigenen Knoten mehr erzeugt werden müssen.

XSLT 2.0

Temporäre Bäume

Die wichtigste Neuerung in XSLT betrifft die Entfernung des Typs Ergebnisbaumfragment (*result tree fragment*). Solche im Transformationsprozess erzeugten XML-Fragmente können in XSLT 1.0 ausschließlich in die Ausgabe kopiert oder in eine Zeichenkette umgewandelt werden. Eine Weiterverarbeitung in Form einer weiteren Transformation ist nicht möglich. In XSLT 2.0 wird ein solches Fragment nun durch die Wurzel (den Dokumentknoten) eines temporären Baumes repräsentiert. Auf diesen

[10] Diese vom Autor entwickelte Vorgehensweise wird in [Ten01] unter dem Namen *Becker Method* dargestellt.

Baum kann mit Hilfe normaler XPath-Ausdrücke zugegriffen werden. Insbesondere können diese Knoten ebenfalls durch `xsl:apply-templates` ausgewählt und so erneut transformiert werden.

Mehrere Ausgabedokumente

In XSLT 1.0 gibt es kein Konzept, mehrere unterschiedliche Ausgabedokumente zu erzeugen. Dies liegt wieder darin begründet, dass XSLT ursprünglich allein als Bestandteil einer Stilsprache entworfen wurde, in der nur ein primäres Ausgabedokument sinnvoll ist. XSLT 2.0 sieht nun eine Anweisung `xsl:result-document` vor, mit deren Hilfe separate Ausgabedokumente erzeugt werden können. Darüber hinaus lässt sich die Serialisierung in einen XML-Text detaillierter steuern. Da die verschiedenen Serialisierungsoptionen nicht nur in XSLT benötigt werden, hat das W3C diese in eine eigene Spezifikation ausgelagert [W3C03f].

Gruppieren

Die Lösung von Gruppierungsproblemen gehört in XSLT 1.0 zu den anspruchsvollen Aufgaben. Zwar gibt es eine unter dem Namen »Muench'sche Methode« (nach deren Entwickler Steve Muench) bekannt gewordene Methode, die jedoch für mehrstufige Gruppierungsaufgaben sehr komplex wird. Da die Bildung von Gruppen jedoch sehr häufig innerhalb von XML-Transformationen auftritt, sieht XSLT 2.0 hierfür eine spezielle Anweisung vor.

Stylesheet-Funktionen

Die Definition eigener Funktionen ist in XSLT 1.0 nur über den Umweg benannter Templates möglich. Innerhalb eines solchen Template muss der Rückgabewert als XML-Fragment (d.h. als Textknoten) erzeugt werden. Um diesen Wert anschließend innerhalb von XPath-Ausdrücken verwenden zu können, muss der Aufruf `xsl:call-template` immer in eine zusätzliche Variablendefinition eingebettet werden, sodass diese Variable den zurückgegebenen Wert aufnimmt. In XSLT 2.0 können nun Funktionen mit der Anweisung `xsl:function` direkt definiert und innerhalb von XPath-Ausdrücken aufgerufen werden.

Textanalyse

XSLT 2.0 bietet erweiterte Möglichkeiten zur Auswertung von Texten mit Hilfe regulärer Ausdrücke. Das Kapitel 5.6.6 wird auf dieses Problem im Zusammenhang mit der in STX dafür vorgesehenen Anweisung genauer eingehen.

Typen und Validierung

Das strenge Typsystem von XPath 2.0 setzt sich in XSLT 2.0 fort. So können nun Variablen bei ihrer Deklaration typisiert werden. Die Zuweisung eines unverträglichen Werts führt in diesem Fall zu einem Typfehler.

Darüber hinaus ermöglicht XSLT 2.0 das Importieren von Schema-Definitionen. Auf diese Weise können beispielsweise Templates programmiert werden, die alle Knoten eines bestimmten Typs oder seiner Subtypen bearbeiten. Auf die gleiche Weise lassen sich Templates für alle Elemente einer Schema-Ersetzungsgruppe erstellen. Des Weiteren ermöglicht XSLT 2.0 wahlweise eine lockere oder strenge Validierung des erzeugten Ergebnis-Dokuments oder auch den gänzlichen Verzicht auf die Validierung.

Kapitel 5
Streaming Transformations for XML

Das Design der Transformationssprache XSLT bedingt, dass diese nicht für die Verarbeitung beliebig großer Dokumente und XML-Datenströme eingesetzt werden kann. Allerdings sind auch keine alternativen Transformationsmethoden bekannt, die sowohl dieses Kriterium erfüllen, als auch hinsichtlich Mächtigkeit und Einfachheit mit XSLT vergleichbar sind (siehe Kapitel 3).

Ziel der vorliegenden Arbeit ist es, eine Transformationssprache für XML zu entwerfen, die auf der einen Seite auf einem Datenstrom operiert und auf der anderen Seite aus XSLT bekannte Konstrukte wieder verwendet. Diese Sprache soll **Streaming Transformations for XML** genannt werden, abgekürzt **STX**.

Dieses Kapitel beschreibt zunächst die Anforderungen an STX und davon ausgehend den Entwurf dieser Sprache.

5.1 Anforderungen

STX ist als spezielle Transformationssprache und nicht als API konzipiert. Ein API kann immer nur einen Zusatz zu einem durch die jeweilige Programmiersprache festgelegten Daten- und Typsystem darstellen. XML-Daten müssen hier stets durch die in der Sprache vorhandenen Konstrukte repräsentiert werden. Eine eigene Transformationssprache kann dagegen von vornherein XML als nativen Datentyp vorsehen und die vertraute XML-Syntax zur lexikalischen Repräsentation nutzen. Zudem lässt sich eine speziell entworfene Sprache auf den Aspekt der Transformation zuschneiden, indem sie einen leicht verständlichen Mechanismus für die Abbildung des Eingabe-XML auf das angestrebte Ausgabe-XML bereitstellt.

Die weiteren Anforderungen an die zu entwerfende Sprache beinhalten:

- **Serielle Verarbeitung eines Datenstroms**
  STX muss einen XML-Datenstrom transformieren können, ohne dass dazu die Repräsentation des gesamten Dokuments im Speicher erforderlich ist. Die ersten Ergebnisdaten sind so unmittelbar nach dem Beginn der Transformation verfügbar. Beliebig große XML-Dokumente können transformiert werden, indem sie sequentiell als Strom verarbeitet werden. Die serielle Verarbeitung ist eine Zusicherung durch die Sprache selbst und nicht eine Eigenschaft einer speziellen Implementation.
- **Ähnlichkeit zu XSLT**
  Die Sprache XSLT hat seit ihrer Spezifizierung vor viereinhalb Jahren eine sehr große Verbreitung innerhalb der XML-Gemeinde gefunden. Es bietet sich daher an, die Vorteile von XSLT in STX weiter zu nutzen. Dazu zählen insbesondere die XML-basierte Syntax von XSLT, der Template-Mechanismus, die direkte Angabe von literalen Ausgabeelementen, die Möglichkeit von Attributwert-Templates sowie die implizite Serialisierung des Ergebnisses als XML-Text. Die Pfadsprache muss so abgewandelt werden, dass eine interne Baumdarstellung nicht aufgebaut werden muss.

Eine weitgehende Ähnlichkeit mit XSLT fördert die Akzeptanz von STX und verringert den erforderlichen Lernaufwand. Darüber hinaus lassen sich vom Wesen her sequentielle XSLT-Transformationen mit geringem Aufwand nach STX portieren.

Die Übernahme syntaktischer Konstrukte aus XSLT bedarf einer sorgfältigen Vorgehensweise. Gleiche Notationen mit unterschiedlichen Bedeutungen in XSLT und STX sind zu vermeiden.

- **Kompatibilität mit den Spezifikationen des W3C**
  Die Transformationssprache STX muss sich in die Familie der W3C-Spezifikationen zu XML einordnen. Existierende Spezifikationen sollten – soweit sinnvoll – in STX verwendet werden. Dies betrifft neben der XML-Spezifikation selbst insbesondere die Namensräume [W3C99a], das Infoset [W3C04c], das XPath-Datenmodell [W3C03d] oder auch XML-Base [W3C01c].
- **Möglichkeit komplexer Transformationen**
  Serielle Transformationssprachen erscheinen von vornherein in ihrer Mächtigkeit beschränkt. STX sollte die Möglichkeit bieten, gegebenenfalls auf Kosten des benötigten Speichers, beliebige XML-Transformationen auszuführen. Im Unterschied zu XSLT hat der STX-Anwender jedoch selbst Einfluss darauf, in welchem Umfang die Transformation Speicherplatz erfordert.
- **Plattformunabhängigkeit und Integrierbarkeit**
  STX darf keine Vorgaben für die Art der Implementierung des STX-Prozessors machen. So dürfen keine Abhängigkeiten von einer konkreten universellen Programmiersprache enthalten sein. Darüber hinaus sollte STX implementationsunabhängige Schnittstellen zu anderen Transformationsmethoden bieten. Nur so kann STX prinzipiell in jeder Programmiersprache implementiert und folglich problemlos in existierende Applikationen integriert werden.

  Idealerweise sollte sich STX ohne Schwierigkeiten auf der Basis verbreiteter Stream-APIs wie SAX oder XmlPull implementieren lassen.
- **Validierung und Schema-Unterstützung**
  STX sollte potenziell in der Lage sein, eine XML-Schemasprache zu unterstützen und das durch die Transformation erzeugte XML entsprechend zu validieren.

Letztgenannter Punkt wird in der vorliegenden Arbeit nicht umgesetzt. STX orientiert sich an XSLT 1.0, das weder das in XPath 2.0 eingeführte umfangreiche Typsystem noch eine Validierung des Ergebnisses unterstützt. Wie der derzeitige Entwurf von XSLT 2.0 zeigt, ist eine solche Erweiterung auf Ebene der Transformationssprache leicht möglich. Allerdings belegt die Diskussion um XSLT 2.0 auch, dass die damit einhergehende strenge Typisierung nicht unumstritten ist.

Es existieren mittlerweile eine Reihe von Schema-Implementierungen, die XML-Datenströme anhand eines XML-Schemas validieren. Eine solche Validierungskomponente kann somit den Ergebnisdatenstrom direkt verarbeiten. Bei einer zukünftigen schema-unterstützenden STX-Version würde eine solche Komponente direkter Bestandteil des STX-Prozessors sein.

Für viele einfache Anwendungen ist jedoch die Formulierung eines Schemas für die mit XML ausgezeichneten Daten nicht erforderlich. Da XML immer auch ohne Typinformationen verarbeitet werden können muss, darf eine Validierung nur als optionaler Bestandteil vorgesehen werden.

STX als XSLT-Teilmenge?

Auf der Suche nach Lösungen für die XSLT-Beschränkung bei großen Dokumenten tauchen auch immer wieder Vorschläge auf, eine sequentielle Teilmenge von XSLT zu definieren. Durch Einschränkungen im Sprachumfang sollte ein XSLT-Prozessor in der Lage sein, ein XML-Dokument sequentiell zu transformieren. Beispielsweise existiert mit dem als Teilmenge von XPath definierten *Sequential XPath* [Des01] bereits eine Lösung für die Pfadsprache, die direkt auf einem XML-Datenstrom operiert. Für XSLT gibt es jedoch bisher keine vergleichbaren Lösungen.

Das liegt vor allem daran, dass der funktionale Charakter von XSLT ein Mitführen von Zustandsinformationen in Variablen verbietet. Eine sequentielle XSLT-Teilmenge könnte nur sehr einfache Transformationen ausführen, die während der Bearbeitung eines Knotens keine Informationen aus anderen Teilen des Dokuments benötigen. Ein Zustand könnte allein über rekursive Template-Aufrufe mitgeführt werden. Dies verhindert jedoch eine natürliche Tiefe-zuerst-Traversierung des Dokuments und reduziert erheblich die Verwendbarkeit der resultierenden Sprache.

5.2 Verarbeitungsmodell

Die Transformationssprache STX führt eine sequentielle Verarbeitung der XML-Daten durch. Unter *sequentiell* ist dabei zu verstehen, dass

- die Transformation unmittelbar nach dem Einlesen der ersten XML-Daten beginnen kann und
- während der Transformation stets nur ein begrenzter Umfang an Informationen aus den Eingabedaten automatisch mitgeführt wird.

Diese zweite Eigenschaft ermöglicht die Transformation beliebig großer XML-Dokumente mit STX. Dem STX-Anwender steht es natürlich frei, nach eigenem Ermessen zusätzliche Informationen aus dem Eingabestrom in Variablen zwischenzuspeichern und mitzuführen.

Für jeden besuchten Knoten innerhalb des XML-Baumes sind keinerlei Informationen über die nachfolgenden Knoten (`following`) vorhanden, da diese erst später im Eingabestrom übertragen werden. Informationen über die Nachkommen (`descendant`) können erst nach der vollständigen Verarbeitung der Kindknoten vorliegen. Informationen über vorangegangene Knoten (`preceding`) sowie die Vorfahren (`ancestor`) sind dagegen prinzipiell verfügbar.

Zur Entscheidung der Frage, welche dieser Informationen bereits durch die Sprache STX selbst bereitgestellt werden sollten, werden die folgenden Kriterien herangezogen:

- *Umfang der Informationen*

  Erfahrungen zeigen, dass sehr große Dokumente in der Regel sehr breiten Bäumen entsprechen. Zusätzliche Daten spiegeln sich nicht in einer größeren Tiefe des XML-Baumes wider, sondern in zusätzlichen Geschwisterknoten. Dies hängt damit zusammen, dass Datenstrukturen lokal betrachtet meist als Liste dargestellt

werden[1] und viele Dokumenttypen nur eine endliche Tiefe der modellierten Daten erlauben.

- *Relevanz der Informationen, Abhängigkeit der Transformation eines Elements von seinen Vorgängern und seinen Vorfahren*

  In der Regel spielen die Vorfahren eine große Rolle. Die Formatierung von Listenelementen (`<li>` genannt) hängt davon ab, ob sie innerhalb einer einfachen Liste (`<ul>`) oder einer nummerierten Liste (`<ol>`) vorkommen. Die Darstellung einer Überschrift hängt davon ab, welche Gliederungsstufe durch das übergeordnete Kapitelelement festgelegt wird. In XSLT wird dieser Anwendungsfall durch Muster unterstützt, die solche Beziehungen zu Vorfahren ausdrücken können (z.B. `match="ul/li"`).

  Demgegenüber spielen die Vorgänger häufig keine Rolle für eine Transformation. Lediglich das Wissen um die Position eines Elementes kann nützlich sein, zum Beispiel, wenn der erste Abschnitt eines Textes anders formatiert wird als die folgenden.

STX stellt daher allein den Zugriff auf die Vorfahren sowie Positionszähler zur Verfügung. Daten aus vorangegangenen Knoten müssen explizit durch den STX-Anwender in Variablen abgespeichert werden.

Zur Veranschaulichung einer STX-Anwendung dient das folgende Beispiel. Die in Listing 3 auf Seite 43 dargestellte XSLT-Transformation kann in STX folgendermaßen programmiert werden (Listing 6):

Listing 6 *STX-Transformations-Sheet*

```
<?xml version="1.0" encoding="iso-8859-1"?>
<stx:transform xmlns:stx="http://stx.sourceforge.net/2002/ns"
               version="1.0" pass-through="text">

  <stx:template match="faq">
    <faq>
      <source>
        <stx:value-of select="@quelle" />
      </source>
      <stx:process-children />
    </faq>
  </stx:template>

  <stx:template match="frage">
    <question>
      <stx:process-children />
    </question>
  </stx:template>

  <!-- gegebenenfalls weitere Templates für <antwort> etc. -->

</stx:transform>
```

[1] Betrachtet man zum Beispiel die Gliederung eines Buches, so besitzen umfangreiche Werke (große Dokumente) im Allgemeinen mehr Kapitel als kurze Abhandlungen und nicht etwa eine tiefer verschachtelte Struktur in Form von Unterunterunterkapiteln.

Ein solches STX-»Programm« soll im Folgenden *STX-Transformations-Sheet*[2] genannt werden. Zwar weist STX viele (nicht nur syntaktische) Gemeinsamkeiten mit XSLT auf, jedoch hat STX selbst keinen speziellen Stilbezug. Der aus der XSL-Welt stammende Begriff *Stylesheet* erscheint für STX unpassend.

Wie Listing 6 zeigt, ist STX wie XSLT deklarativ und arbeitet ebenfalls regelbasiert. Darüber hinaus bestehen die folgenden wesentlichen Unterschiede zu XSLT:

- Der Namensraum für STX-Elemente lautet *http://stx.sourceforge.net/2002/ns*. Üblicherweise wird das Präfix »`stx:`« für diesen Namensraum deklariert. XSLT verwendet stattdessen den Namensraum *http://www.w3.org/1999/XSL/Transform*.
- Das Wurzelelement heißt `stx:transform`. In XSLT können stattdessen die Elemente `xsl:stylesheet` und `xsl:transform` synonym verwendet werden.
- Dieses Wurzelelement besitzt das zusätzliche Attribut `pass-through`. Dieses ermöglicht eine Konfigurierung der Vorgabe-Templates. Der Wert `"text"` bewirkt ein XSLT-kompatibles Verhalten, indem nur die Textknoten automatisch in die Ausgabe kopiert werden.
- Anstelle der Anweisung `xsl:apply-templates` kommt hier zweimal die STX-Anweisung `stx:process-children` zum Einsatz. Auf diese und die verwandten Anweisungen wird in Kapitel 5.6.3 im Detail eingegangen.

Templates

Die wichtigsten Bestandteile eines solchen STX-Transformations-Sheet sind wie in XSLT die Templates. Auch hier bestimmt der STX-Prozessor anhand eines Musters, welches Template für den aktuell betrachteten Knoten anzuwenden ist. Im Gegensatz zu XSLT werden hier jedoch die Knoten stets in der Reihenfolge verarbeitet, in der sie im Eingabedokument auftreten. Dies entspricht einer Preorder-Traversierung des XML-Baumes (siehe Abbildung 3).

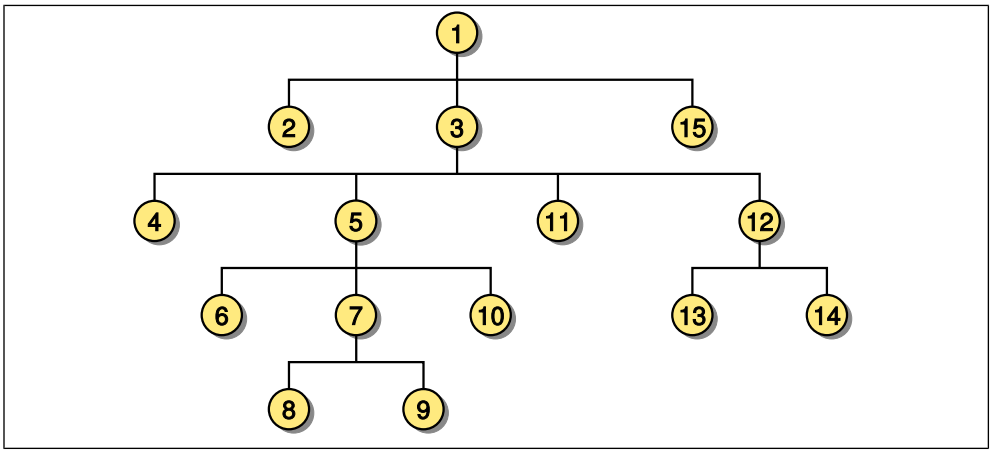


Abbildung 3

Reihenfolge der Knoten in einer Preorder-Traversierung

Daraus ergibt sich unmittelbar, dass in einem STX-Transformations-Sheet der Kontextknoten nicht beliebig geändert werden kann. Der durch ein Template bearbeitete

[2]Die Verknüpfung deutscher (*Transformation*) und englischer (*sheet*) Begriffe ist sicher nicht optimal. Der englische Fachbegriff *transformation sheet* lässt sich jedoch nur schwer vollständig ins Deutsche übertragen. Insbesondere erscheint eine Übersetzung von *sheet* hier nicht hilfreich. Alternativ dazu werden im Folgenden auch *STX-Transformation* (wenn eher der Prozess gemeint ist) oder *STX-Code* (wenn die einzelnen STX-Anweisungen im Vordergrund stehen) verwendet.

Kontextknoten ist in STX immer der aktuelle Knoten aus dem Eingabestrom. XSLT-Anweisungen wie `xsl:apply-templates` und `xsl:for-each`, die beliebige Knoten zu neuen Kontextknoten bestimmen, sind in STX in dieser Form nicht realisierbar. Für sie gibt es daher abgewandelte Varianten, die in Kapitel 5.6.1 bzw. Kapitel 5.6.3 vorgestellt werden.

Die einzige Möglichkeit, die Transformation der Eingabedaten innerhalb eines Template fortzusetzen, besteht in der Verarbeitung der im Datenstrom folgenden Knoten, etwa der Kindknoten. Der aktuelle Kontextknoten wird in diesem Fall durch den STX-Prozessor zwischengespeichert, bis die Verarbeitung der Kindknoten beendet ist. Im Listing 6 wird beispielsweise in Zeile 10 das aktuelle Template durch die Anweisung `stx:process-children` unterbrochen, bis alle Kindknoten verarbeitet worden sind. Wird dieses Template anschließend fortgesetzt, sind alle Eigenschaften des dazugehörigen Kontextknotens wieder verfügbar.

Ein STX-Prozessor verwaltet zu diesem Zweck einen so genannten Vorfahren-Stack (Kellerspeicher) in dem die Knoten für alle geöffneten Elemente abgelegt werden. Der Kontextknoten ist immer der oberste Knoten auf diesem Stack. Erreicht der STX-Prozessor das Ende eines Elements (im XML-Text dargestellt durch das End-Tag), wird der dazugehörige Knoten vom Vorfahren-Stack entfernt.

Der STX-Prozessor ermittelt nun für jeden neu auf den Stack gelegten Knoten ein passendes Template und führt dieses aus. Kommt es zu einer Unterbrechung der Template-Ausführung (etwa für Kindknoten durch die Anweisung `stx:process-children`), wird das Template zurückgestellt. Der erste Kindknoten wird auf den Vorfahren-Stack gelegt, und es wird ein geeignetes neues Template ermittelt. Nach dem Ende der Kindknoten, d.h. beim Erreichen des Elementendes, wird das zuvor zurückgestellte Template fortgesetzt. Die Unterbrechung von Templates für die Verarbeitung anderer Knoten ist Inhalt von Kapitel 5.6.3.

Änderbare Variablen

Die strikt definierte Bearbeitungsreihenfolge in einer STX-Transformation ermöglicht und erfordert änderbare Variablen. Anders als in XSLT gestattet es STX nicht, Templates in einer selbst gewählten Reihenfolge oder sogar parallel auszuführen. Die Bearbeitungsreihenfolge wird durch das Eingabedokument, d.h. durch die Reihenfolge der XML-Daten im Eingabestrom bestimmt. STX besitzt in dieser Hinsicht einen imperativen Charakter. Da sich zudem vorangegangene Knoten nicht mehr auf dem Vorfahren-Stack befinden, kann der STX-Prozessor keine Informationen über diese Knoten automatisch zur Verfügung stellen. Vielmehr müssen in STX Variablen zur Speicherung dieser Informationen benutzt werden.

Vorgabe-Templates

STX definiert ebenso wie XSLT, wie standardmäßig mit Knoten verfahren werden soll, für die kein passendes Template gefunden wurde. Im Gegensatz zu XSLT besitzt STX jedoch keinen festen Satz an Vorgabe-Templates, sondern erlaubt eine Konfigurierung des gewünschten Verhaltens. STX unterscheidet drei Verhaltensweisen, die über ein Attribut `pass-through` festgelegt werden können:

`"none"`
: Es wird standardmäßig nichts kopiert. Alle Knoten, für die kein Template gefunden wird, erscheinen auch nicht in der Ausgabe. Der Wert `"none"` ist auch der Vorgabewert für das Attribut `pass-through`.

`"all"`
: Alle Knoten, für die kein Template gefunden wird, werden in die Ausgabe kopiert.

`"text"`
: Es wird standardmäßig nur Text kopiert. Diese Variante entspricht dem Verhalten von XSLT.

Bei Elementknoten werden standardmäßig in allen Fällen die Kindknoten verarbeitet. Selbst wenn nichts kopiert wird, werden für die Kindknoten erneut Templates gesucht.

5.3 Datenmodell

Das Datenmodell definiert zum einen die zur Verfügung stehenden Datentypen und beschreibt zum anderen die für STX relevanten Arten und Eigenschaften der in einem XML-Eingabedokument enthaltenen Knoten. Im Interesse der Ähnlichkeit zu XSLT wird STX sich an dessen Datenmodell orientieren. Das in XSLT 1.0 benutzte Modell ist in der XPath-1.0-Spezifikation [W3C99b] beschrieben. Mit der Weiterentwicklung zu XSLT 2.0 und der parallelen Entstehung von XQuery 1.0 hat das W3C das gemeinsame Datenmodell in eine eigene Spezifikation namens *XQuery 1.0 and XPath 2.0 Data Model* [W3C03d], hier kurz XPath2-Datenmodell genannt, ausgelagert. Dieses Datenmodell kann für wohlgeformte und den Namensraumregeln folgende XML-Dokumente angewendet werden. Die Gültigkeit bezüglich einer DTD oder eines Schemas ist nicht erforderlich.

STX übernimmt das XPath2-Datenmodell im Wesentlichen und modifiziert es nur teilweise. Durch diese prinzipielle Übernahme werden jedoch eine Reihe von Knoteneigenschaften beschrieben, die für STX irrelevant sind. Ein speziell für STX entworfenes Datenmodell könnte einfacher gestaltet werden. Da das W3C jedoch derzeit mit dem XPath2-Datenmodell bereits ein weitgehend sprachunabhängiges Modell entwickelt, bietet es sich an, dieses ebenfalls für STX zu nutzen.

5.3.1 Sequenzen

Eine der wesentlichen Neuerungen im XPath2-Datenmodell gegenüber XPath 1.0 ist die Einführung von Sequenzen. Deren Eigenschaften wurden bereits in Kapitel 4.6 erörtert.

STX unterstützt ebenfalls Sequenzen. Jeder Wert in STX ist eine Sequenz; gegebenenfalls eine einelementige. Es gelten die gleichen Eigenschaften wie im XPath2-Datenmodell. Wie das folgende Kapitel zeigt, existieren jedoch Unterschiede bei den atomaren Werten, die in Sequenzen enthalten sind.

5.3.2 Einfache Datentypen und atomare Werte

Basis: XPath 1.0

Die einfachen Datentypen in STX entsprechen den drei durch XPath 1.0 definierten einfachen Typen. STX übernimmt damit nicht das im XPath2-Datenmodell eingeführte komplexe Typsystem, welches 21 einfache Typen unterscheidet. Dafür gibt es die folgenden Gründe:

- Das Typsystem von XPath 1.0 ist einfach, leicht verständlich und hat seine Eignung für XML-Transformationen innerhalb von XSLT 1.0 unter Beweis gestellt.
- XML-Dokumente sind häufig ohne Typinformationen ausgestattet. In diesem Fall kann von den zahlreichen zur Verfügung stehenden Datentypen nicht profi-

tiert werden, da alle Werte eines solchen XML-Dokuments im XPath2-Datenmodell zunächst den unspezifischen Basistyp `xdt:untypedAtomic` besitzen.

- Die mit dem neuen Typsystem einhergehenden strengeren Konvertierungsregeln haben bisher zu heftigen Debatten in den entsprechenden Diskussionsforen des W3C geführt. Hauptkritikpunkt ist dabei, dass das neue Typsystem mehr »künstliche« Typfehler produziere als tatsächliche Konvertierungsfehler aufdecke (siehe auch Tennison in [Ten03]). Das W3C reagierte auf diese Probleme mit der Zusage, die derzeitigen Regeln einer Revision zu unterziehen. Dieser Bereich des neuen XPath2-Datenmodells ist noch sehr instabil und somit für eine Übernahme in STX im aktuellen Stadium nicht empfehlenswert.
- Diese Entscheidung wirkt sich nicht auf die grundlegenden Eigenschaften von STX als Transformationssprache für XML-Datenströme aus. Eine spätere Revision im Sinne strengerer Regeln wie in XPath 2.0 ist ohne weiteres möglich.

Die durch STX unterstützten einfachen Typen werden im Folgenden genau definiert:

Boolesche Werte (*boolean*)
Der Wertebereich dieses Typs enthält allein die beiden logischen Werte *wahr* und *falsch*. Es existieren in STX keine Literale für diese Werte.

Zahlen (*number*)
Der Wertebereich von *number* umfasst 64-Bit-Fließkommazahlen mit doppelter Genauigkeit gemäß IEEE 754 [IEEE754]. Zahlen-Literale in STX genügen der Produktion 58 der STXPath-Grammatik im Anhang B.2.

Zeichenkette (*string*)
Zeichenketten enthalten eine beliebige Folge von Unicode-Zeichen. In STX werden Zeichenketten-Literale durch einfache oder doppelte Anführungszeichen gekennzeichnet. Sie genügen der Produktion 6 der STXPath-Grammatik.

5.3.3 Knoten

XML-Bäume

Das XPath2-Datenmodell beschreibt ein XML-Dokument zunächst grundsätzlich als einen Baum von Knoten, die sich wiederum auf Informationseinheiten des XML-Infoset [W3C04c] beziehen. Diese Baumansicht steht nicht im Widerspruch zum seriellen Verarbeitungsmodell, da STX keine Sprachmittel zur Verfügung stellt, die einen wahlfreien Zugriff auf die Knoten des Baumes ermöglichen. So sind allein die Vorfahren des Kontextknotens tatsächlich über Pfadausdrücke erreichbar. Die STX-Pfadsprache wird in Kapitel 5.4 vorgestellt.

Die im XPath2-Datenmodell definierte Dokumentordnung für die Knoten eines Baumes korrespondiert zur Reihenfolge, in der die Knoten durch den STX-Prozessor verarbeitet werden. Dies gilt nicht für Attributknoten, da die Attribute des Kontextknotens zum einen nur bei Bedarf und zum anderen beliebig oft verarbeitet werden können. Darüber hinaus besitzen die Attribute untereinander keine definierte Ordnung. Die Ordnung zwischen Knoten aus verschiedenen Dokumenten kann ebenfalls nicht auf die Verarbeitungsreihenfolge abgebildet werden, da die Verarbeitung eines XML-Datenstroms zeitweilig für einen anderen XML-Datenstrom unterbrochen werden kann.

Knotentypen

Das XPath2-Datenmodell definiert die folgenden sieben verschiedenen Knotentypen (sie wurden bereits im Kapitel 4.3 erwähnt):

- Dokumentknoten
- Elementknoten
- Attributknoten
- Namensraumknoten
- Verarbeitungsanweisungsknoten
- Kommentarknoten
- Textknoten

STX fügt diesen zwei weitere Knotentypen hinzu:

- CDATA-Knoten
- DOCTYPE-Knoten

Knoten-eigenschaften

Die Eigenschaften der Knoten werden im XPath2-Datenmodell über so genannte Zugriffsfunktionen verfügbar gemacht. Diese werden hier durch das Präfix »`dm:`« gekennzeichnet. Es handelt sich dabei ausschließlich um Funktionen, die die Datenmodell-Schnittstelle beschreiben. Es sind keine Funktionen im literalen Sinne, die von Anwendern aufgerufen werden können.

- `dm:base-uri`
- `dm:node-kind`
- `dm:node-name`
- `dm:parent`
- `dm:string-value`
- `dm:typed-value`
- `dm:type`
- `dm:children`
- `dm:attributes`
- `dm:namespaces`
- `dm:nilled`

Die Funktionen `dm:typed-value`, `dm:type` und `dm:nilled` greifen auf Schemainformationen zurück und werden durch STX nicht unterstützt. Der Wert der Funktion `dm:children` kann aufgrund des seriellen Charakters von STX nicht direkt zugänglich gemacht werden. Stattdessen werden die durch diese Funktion gelieferten Knoten (d.h. die Kindknoten) durch die Anweisung `stx:process-children` verarbeitet. Die Funktion `dm:parent` schließlich wird nur für Knoten auf dem Vorfahren-Stack benutzt.

Ansonsten gelten die Eigenschaften der Typen Attributknoten, Namensraumknoten, Verarbeitungsanweisungsknoten und Kommentarknoten unverändert ebenso in STX. Dokumentknoten unterscheiden sich allein im Rückgabewert der Zugriffsfunktion `dm:string-value`, die in STX die leere Zeichenkette liefert. Darüber hinaus sind

Namensraumknoten in STX nicht direkt erreichbar. Die in ihnen repräsentierten relevanten Informationen (der Namensraum-URI und das dazugehörige Präfix) können über Funktionen abgerufen werden.

Im Folgenden werden die in STX vorgenommenen Änderungen für Element- und Textknoten sowie die beiden neuen Typen CDATA-Knoten und DOCTYPE-Knoten ausführlicher besprochen.

Elementknoten

Der durch die Funktion `dm:string-value` zurückgegebene Zeichenkettenwert eines Elementknotens berechnet sich im XPath2-Datenmodell aus der Verkettung aller Textknoten-Nachkommen dieses Elements. Der Zeichenkettenwert eines Elements ist damit dessen Textinhalt, aus dem jegliches Markup entfernt wurde. Für einen XML-Datenstrom würde das bedeuten, dass der Zeichenkettenwert erst dann bekannt ist, wenn das Ende des Elements erreicht wurde. Gemäß der Preorder-Traversierung werden Elemente jedoch an ihrem Beginn (beim Erreichen des Start-Tags) verarbeitet. Sie besitzen daher zu diesem Zeitpunkt noch keinen in STX verfügbaren Zeichenkettenwert.

Abkürzung für Elemente mit reinem Textinhalt

Mit Hilfe einer kleinen Erweiterung des Verarbeitungsmodells lässt sich jedoch auch für Elemente ein geeigneter Zeichenkettenwert definieren. Häufig tritt in XML-Daten der Fall auf, dass Elemente ausschließlich Text, d.h. weder Kindelemente noch andere Arten von Markup im Inneren enthalten. Soll der Text solcher Elemente verarbeitet (zum Beispiel für eine spätere Verwendung in einer Variablen gespeichert) werden, müsste das in einem Template erfolgen, das den Textknoten unterhalb des Elementknotens verarbeitet:

```
<stx:template match="begriff/text()">
  <stx:assign name="term" select="." />
</stx:template>
```

Hier wird der Variablen `term` der im Element `begriff` enthaltene Text zugewiesen. Diese Vorgehensweise funktioniert dann nicht, wenn das Element `begriff` leer ist. In diesem Fall existiert kein Textknoten, für den das Template aufgerufen werden könnte. Die Variable `term` würde ihren alten Wert behalten.

Um diesen Anwendungsfall zu vereinfachen und die aus XSLT gewohnten intuitiven Templates für Elementknoten ebenfalls in STX anwenden zu können, definiert STX den Zeichenkettenwert und damit den Rückgabewert der Funktion `dm:string-value` als den Zeichenkettenwert des ersten im Datenstrom folgenden Kindknotens, vorausgesetzt, es handelt sich dabei um einen Textknoten. Ein solcher Textknoten besitzt also keinerlei vorangehende Geschwisterknoten. Ansonsten ist der Zeichenkettenwert die leere Zeichenkette. Auf diese Weise lässt sich das obige Beispiel kompakter und XSLT-typischer als

```
<stx:template match="begriff">
  <stx:assign name="term" select="." />
</stx:template>
```

notieren. Bei Elementen mit weiteren Kindknoten unterscheidet sich jedoch dieser Zeichenkettenwert von dem im XPath2-Datenmodell beschriebenen Wert. Die folgende Tabelle enthält dafür einige Beispiele.

XML-Markup	p-Zeichenkettenwert
`<p>Ein <em>warnendes</em> Beispiel</p>`	»Ein «
`<p>Zwei <!-- oder mehr --> Worte</p>`	»Zwei «
`<p><strong>STX</strong></p>`	»«

Dieser als *Look-Ahead* dem Prozessor bekannte Textknoten hat keinerlei Auswirkung auf das Pattern-Matching oder andere Transformationsabläufe. So ist der im Voraus gelesene Knoten weder bereits auf dem Vorfahren-Stack sichtbar, noch kann durch den STX-Anwender auf unmittelbar folgende Kindknoten anderer Typen zugegriffen werden. Der Look-Ahead-Textknoten dient allein zur Bestimmung des Zeichenkettenwerts des Elternelements. Aus diesem Grund ist auch ein darüber hinaus gehendes *Im-Voraus-Lesen* weiterer Knoten nicht sinnvoll.

Textknoten

Textknoten repräsentieren im XPath2-Datenmodell zusammenhängende Textabschnitte des XML-Dokuments. Das bedeutet, dass zwei Textknoten hier niemals als unmittelbare Geschwisterknoten auftreten.

In einem seriellen Verarbeitungsprozess würde dies bei sehr langen Textpassagen problematisch werden. Ein solcher Textabschnitt könnte erst dann verarbeitet werden, wenn er vollständig in den Speicher eingelesen worden ist. Für Textknoten gilt in XPath im Kleinen, was für XML-Dokumente im Großen gilt. Einerseits lässt sich zwar davon ausgehen, dass ein »vernünftiges« XML-Dokument keine langen Textpassagen enthält, hieße das doch, dass dieser Text keine Unterstruktur besitzt. Andererseits können jedoch z.B. in XML verpackte und als Base64-kodierte Binärdaten ohne weiteres zu solch großen Textblöcken führen.

In STX wird daher diese Eigenschaft für Textknoten aufgehoben. Ein zusammenhängender Textabschnitt darf durch mehrere Textknoten repräsentiert werden. Ob eine solche Zerlegung erfolgen soll, wird durch den Anwender im STX-Transformations-Sheet konfiguriert. Standardmäßig verhält sich STX in dieser Hinsicht wie XSLT bzw. XPath, indem Textknoten nicht zerlegt werden. Alternativ dazu kann vereinbart werden, dass Textknoten jeweils einzelne Zeilen repräsentieren, indem sie an Zeilenenden[3] voneinander getrennt werden.

`text-by-lines`

Die STX-Anweisungen `stx:transform` und `stx:group` (siehe Kapitel 5.6.4) besitzen zu diesem Zweck das Attribut `text-by-lines` mit den möglichen Werten `"yes"` und `"no"`. Der Vorgabewert ist `"no"`.

CDATA-Knoten

CDATA-Abschnitte ermöglichen eine alternative lexikalische Repräsentation für Textdaten, die die in XML reservierten Zeichen < und & enthalten. Sie tragen jedoch keine eigene semantische Bedeutung. Insbesondere sollte eine Applikation, die XML-Daten verarbeitet, Textdaten in CDATA-Abschnitten niemals speziell interpretieren.

[3] Die XML-Spezifikation [W3C04a] definiert, welche Zeichen in einem XML-Text als Zeilenendezeichen gelten. Durch den XML-Parser werden diese jedoch normalisiert, sodass in den XML-Daten nur noch das Zeichen #xA am Zeilenende auftritt.

Aus diesem Grund werden CDATA-Abschnitte nicht im XML-Infoset und also auch nicht im XPath2-Datenmodell repräsentiert.[4]

Das hat zur Folge, dass in XSLT zwar über eine Ausgabeoption gesteuert werden kann, für welche Elemente des erzeugten XML der Textinhalt in CDATA-Abschnitten ausgegeben werden soll, es jedoch keine Möglichkeit gibt, auf CDATA-Abschnitte der Eingabe zuzugreifen. Insbesondere können CDATA-Abschnitte auch nicht einfach in die Ausgabe kopiert werden. Dies bedeutet eine nicht zu unterschätzende Einschränkung, da durch eine XSLT-Transformation ungewollt die lexikalische Repräsentation des im XML-Dokument enthaltenen Textes verändert wird.

In STX sind deshalb CDATA-Abschnitte während der Transformation sichtbar und durch einen eigenen Knotentyp repräsentiert. Dies setzt allerdings voraus, dass der Eingabestrom Informationen über CDATA-Grenzen enthält. Ist das nicht der Fall (das ist gemäß Infoset zulässig), können auch in STX CDATA-Abschnitte nicht von normalen Textdaten unterschieden werden.

`recognize-cdata`

Die Behandlung von CDATA-Abschnitten ist in STX optional und kann mit Hilfe des Attributs `recognize-cdata` ein- (Wert "`yes`") und ausgeschaltet ("`no`") werden. Standardmäßig werden CDATA-Abschnitte erkannt. Textknoten des Datenmodells repräsentieren dann nur Text außerhalb von CDATA-Abschnitten. Es ist nicht möglich, den in CDATA-Abschnitten enthaltenen Text zeilenweise zu verarbeiten (`text-by-lines="yes"`), da dann die ursprünglichen CDATA-Grenzen nicht mehr von den durch den STX-Prozessor eingefügten Grenzen am Zeilenende unterschieden werden können. In diesem Fall wären jegliche Informationen über CDATA-Grenzen wertlos. Aus diesem Grund dürfen die Attribute `recognize-cdata` und `text-by-lines` nicht gleichzeitig den Wert "`yes`" besitzen.

CDATA-Knoten besitzen die gleichen Eigenschaften wie Textknoten. Es gibt jedoch zwei Ausnahmen: Die im XPath2-Datenmodell vorgesehene Zugriffsfunktion `dm:node-kind` liefert den Wert »cdata«, und CDATA-Knoten können leer sein.

DOCTYPE-Knoten

Ein DOCTYPE-Knoten repräsentiert eine Dokumenttyp-Deklaration. Zwar sind diese im XPath2-Datenmodell nicht enthalten, jedoch beschreibt das Infoset für diese eine eigene Informationseinheit. Dokumenttyp-Deklarationen können in XSLT nicht transformiert werden.

Das STX-Datenmodell ergänzt das XPath2-Datenmodell um diesen neuen Knotentyp, um auch Dokumenttyp-Deklarationen in die Transformation einbeziehen zu können. Ein XML-Dokument darf höchstens einen DOCTYPE-Knoten besitzen.

Im Folgenden wird eine Übersicht über die für STX relevanten Zugriffsfunktionen des neuen Knotentyps gegeben:

`dm:base-uri`
: liefert die Eigenschaft *base-uri* des DOCTYPE-Knotens, d.h. den Basis-URI des Dokuments, das diesen Knoten enthält.

`dm:node-kind`
: liefert die Zeichenkette »doctype«.

[4] Allerdings beschreibt das Document Object Modell [W3C00] CDATA-Abschnitte mit Hilfe eines speziellen Interface `CDATASection`. Diese Inkonsistenz ist dem Evolutionsprozess der einzelnen XML-Spezifikationen geschuldet.

`dm:node-name`
liefert den Namen des Dokumenttyps. Dies ist gleichzeitig der Name des Wurzelelements.

`dm:parent`
liefert den Dokumentknoten des Dokuments, das diesen Knoten enthält.

`dm:string-value`
liefert die leere Zeichenkette. Insbesondere kann in STX auch nicht auf DTD-Deklarationen zugegriffen werden.

`dm:children`, **`dm:attributes`**, **`dm:namespaces`**
liefern jeweils die leere Sequenz.

Zusätzlich zu diesen allgemeinen Zugriffsfunktionen werden für DOCTYPE-Knoten die beiden Funktionen `dm:system-id` und `dm:public-id` zur Verfügung gestellt. Ihre Definition erfolgt auf Grundlage der entsprechenden Informationseinheit des Infoset:

`dm:system-id`
liefert den Wert der Infoset-Eigenschaft *system identifier*.

`dm:public-id`
liefert den Wert der Infoset-Eigenschaft *public identifier*.

Für alle anderen Knotentypen liefern beide Funktionen die leere Sequenz.

5.4 Pfadsprache STXPath

Neben dem Datenmodell benötigt STX eine Pfadsprache, die den Zugriff auf die in diesem Datenmodell beschriebenen Daten ermöglicht. Dabei ist zu beachten, dass nur die sichtbaren Knoten (genauer: die Knoten auf dem Vorfahren-Stack) tatsächlich über Pfadausdrücke erreichbar sind. Insbesondere kann ein Pfad generell nicht auf die Kinder des Kontextknotens zugreifen. Der Elternknoten ist nur für Knoten auf dem Vorfahren-Stack verfügbar, jedoch nicht für Knoten, die in Variablen zwischengespeichert wurden.

Basis: XPath 2.0

Die Pfadsprache für XSLT ist, wie bereits genannt, XPath. Sie ist kompakt, leicht zu erlernen und allgemein anerkannt.[5] Die in STX benutzte Pfadsprache *STXPath* wird daher ebenfalls an XPath angelehnt sein, speziell an die seit Ende 2001 entwickelte Nachfolgeversion XPath 2.0 [W3C03a]. Wenn im Folgenden nur noch kurz XPath genannt wird, bezieht sich dies immer auf XPath 2.0.

Eine vollständige XPath-Integration in STX ist aufgrund der eingeschränkten Sicht auf die XML-Daten während einer seriellen Transformation nicht möglich. Die Unterstützung einer XPath-Teilmenge erscheint dagegen sehr sinnvoll. Diese Teilmenge, hier $XPath_S$ genannt,[6] sollte den Zugriff auf die verfügbaren XML-Daten in gewohnter Weise ermöglichen. Es sollte sich um eine syntaktische Teilmenge mit weitgehend identischer Semantik handeln. Anwender von STX sollten nicht mit unerwarteten

[5] Für einige andere W3C-Spezifikationen wurden hingegen Alternativen entwickelt, da sich die spezifizierten Techniken als zu schwerfällig oder zu kompliziert erwiesen. Als Beispiele wären hier alternative Schemasprachen als Gegenstück zu XML-Schema oder spezielle baumorientierte APIs als Alternative zum W3C DOM zu nennen.

[6] $XPath_S$ = XPath-Subset. Dieser Name wird nur in diesem Kapitel benutzt und dient allein der Definition der Pfadsprache STXPath.

Ergebnissen konfrontiert werden, wenn syntaktisch gleiche Ausdrücke in XSLT und STX verschiedene Bedeutungen besitzen. Dieses Ziel lässt sich leider nicht uneingeschränkt erreichen. Die komplette Pfadsprache STXPath ergibt sich durch eine Erweiterung von $XPath_S$ um zusätzliche Knotentests, sodass auf die in STX vorhandenen CDATA- und DOCTYPE-Knoten zugriffen werden kann.

Im folgenden Kapitel wird die Pfadsprache $XPath_S$ aus den Produktionsregeln von XPath abgeleitet. Diese Regeln sind dem XPath-2.0-Entwurf vom 12. November 2003 entnommen. Es kann nicht ausgeschlossen werden, dass die hier verwendeten Produktionsregeln von der endgültigen XPath-2.0-Empfehlung leicht abweichen. Die für die Definition von $XPath_S$ vorgenommenen Änderungen stellen sicher, dass die resultierende Sprache eine Teilmenge von XPath bildet. Alle $XPath_S$-Ausdrücke sind immer auch gültige XPath-Ausdrücke. Analog werden die in STX verwendeten Muster (*patterns*) aus den Regeln für XSLT-Muster abgeleitet. Die Original-Produktionsregeln sind im Entwurf von XSLT 2.0 enthalten.

Im Anhang B.1 wird erläutert, in welcher Form allgemein Regeln einer Grammatik modifiziert werden können, um die gewünschte Teilmengeneigenschaft zu erreichen. Anhang B.2 enthält dann die vollständige STXPath-Grammatik.

5.4.1 $XPath_S$ als Teilmenge von XPath

Entfernung aller Bezüge zu XML Schema

Wie in Kapitel 5.3.2 bereits diskutiert wurde, unterstützt STX keine Schema-Typen. Damit sind alle Schema-Bezüge in XPath für STXPath gegenstandslos.

Aus der Grammatik für XPath 2.0 wurden aus diesem Grund die Produktionsregeln 7–9, 61–64, 66–73, 75, sowie 79–80 gestrichen. In den Produktionsregeln 24–27 wurde jeweils der optionale rechte Teil entfernt. Die Operatoren `instance of`, `treat as`, `castable as` und `cast as` werden damit durch STXPath nicht unterstützt. Schließlich wurden in der Produktionsregel 65 die Alternativen *DocumentTest*, *ElementTest* und *AttributeTest* entfernt, da diese in XPath 2.0 aufgenommenen Knotentests in erster Linie für den Test auf einen bestimmten Schematyp gedacht sind.

Entfernung expliziter Achsen

Die in XPath vorhandenen Achsen (siehe auch Kapitel 4.3) dienen der Navigation im XML-Baum. In STX steht jedoch kein vollständiger Baum zur Verfügung, sondern nur die Knoten auf dem Vorfahren-Stack. Aus diesem Grund erscheinen zunächst allein solche Achsen sinnvoll, die auf den Kontextknoten, dessen Vorfahren und dessen Attribute zugreifen, speziell `self`, `parent`, `ancestor`, `ancestor-or-self` und `attribute`.

Allerdings ist für die Vorfahrenachsen `ancestor` bzw. `ancestor-or-self` keine abkürzende Schreibweise in XPath definiert. Die Einführung einer neuen STX-spezifischen Abkürzung kommt aufgrund der angestrebten XPath-Kompatibilität nicht in Betracht. Daher wurde zugunsten der Leserlichkeit und Kompaktheit von STXPath als Kompromiss auf die »entgegengesetzte« Achse `descendant` zurück-

gegriffen, für die als Abkürzung die Schreibweise `//`[7] definiert ist. Somit können bei der Verwendung von `descendant` als Ersatz für `ancestor` alle Pfadausdrücke in STXPath allein mit der abgekürzten Syntax dargestellt werden. Explizite Achsen können also vollständig aus STXPath gestrichen werden. Die noch in XPath 1.0 vorhandene Achse `namespace` gibt es in XPath 2.0 nicht mehr.

In der Grammatik bedeutet dies, dass die beiden Produktionsregeln 52 und 53 komplett entfallen. Die Produktionsregeln 48 und 49 wurden auf die jeweils zweite Alternative verkürzt, d.h. auf *AbbrevForwardStep* bzw. *AbbrevReverseStep*.

Der Preis für diese kompakte Syntax ist die geänderte Semantik der Notationen `//` und `/`. Es wird nur auf Knoten des Vorfahren-Stack zugegriffen, d.h. auf die aktuellen Vorfahren des Kontextknotens. Hierbei handelt es sich um eine Teilmenge der Knoten, die der gleiche Pfad in XPath in einem vollständigen Baum auswählen würde.

Beispiel

Diese Änderung soll anhand des Beispiels in Abbildung 1 auf Seite 19 verdeutlicht werden. Der Pfad `/faq/antwort/absatz` greift in XPath auf alle `absatz`-Elemente dieses Dokuments zu. In STX hingegen hängt das ausgewählte Element von dem aktuellen Kontextknoten ab. Handelt es sich beispielsweise um den Textknoten, der mit dem Wort »Middleware« beginnt, wählt der angegebene Pfad nur dessen `absatz`-Elternelement aus, nicht jedoch die anderen `absatz`-Elemente. Abbildung 4 veranschaulicht dieses Szenario. Sowohl der Kontextknoten als auch der zum ausgewählten `absatz`-Elementknoten führende Pfad sind hier hervorgehoben.

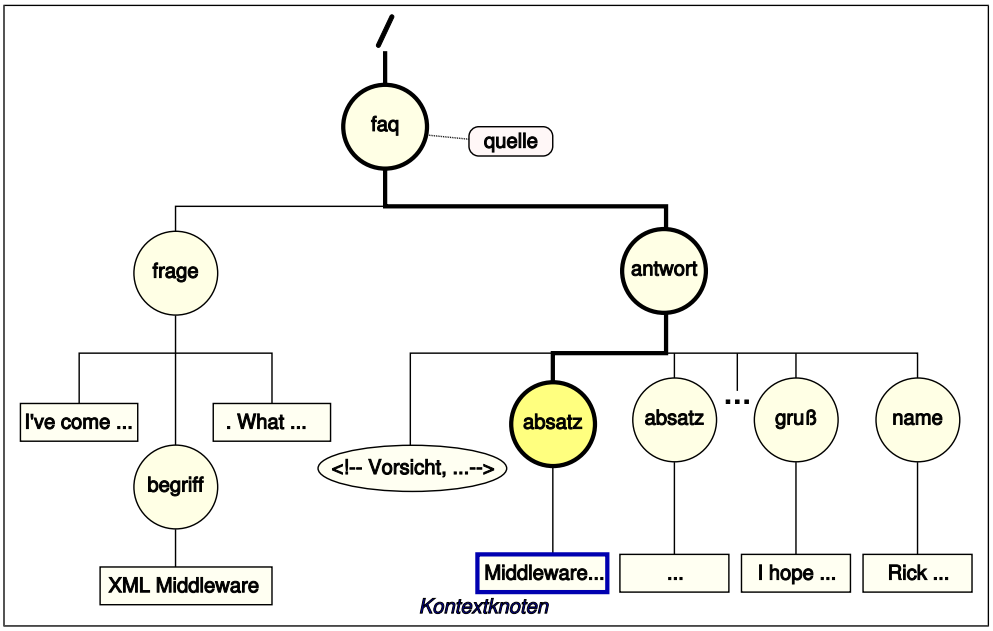


Abbildung 4

Vorfahren-Stack und Pfade in STX

Im Allgemeinen wählt ein solcher Pfad – anders als in XPath – damit maximal einen Knoten aus, und zwar denjenigen, der sich unter den Vorfahren des Kontextknotens befindet. Ein Pfad `//absatz` enthält nur dann mehrere `absatz`-Elemente, wenn diese sich zusammen auf der Vorfahrenachse befinden. Dies ist nur dann der Fall, wenn `absatz`-Elemente ineinander verschachtelt auftreten.

[7] Diese Darstellung ist so nicht ganz exakt. Genau genommen ist `//` die Abkürzung für `/descendant-or-self::node()/`. Allerdings verhält sich diese Abkürzung in den meisten Fällen so, als würde direkt `descendant` verwendet worden sein. Einzig bei der Nummerierung der ausgewählten Knoten (wichtig für positionale Prädikate) gibt es Unterschiede.

Entfernung erweiterter Pfade

In XPath 2.0 sind neben den bekannten Schritten entlang der vorhandenen Achsen auch weitere Ausdrücke in jedem Schritt möglich, beispielsweise Literale, Variablenreferenzen oder Funktionsaufrufe. Innerhalb von Klammern kann letztendlich jeglicher gültige XPath-Ausdruck stehen. Semantisch verbindet sich damit, dass der diesen Schritt repräsentierende Teilausdruck auf der bis dahin ausgewählten Knotenmenge berechnet wird. Liefert das Ergebnis wieder eine Knotenmenge, kann die Auswertung des Pfades wie bekannt fortgesetzt werden, ansonsten bricht die Berechnung mit einem Fehler ab. Bereits in XPath 1.0 konnten Pfade mit einer Variablenreferenz oder einem Funktionsaufruf beginnen, durften dann aber nur mit Schritten entlang der Achsen fortgesetzt werden.

In STX ist es nicht möglich, ausgehend von einem willkürlich gewählten Knoten zu anderen Knoten zu gelangen. Die vorhandenen (eingeschränkten) Achsen können ausschließlich auf die Liste der aktuellen Vorfahren zugreifen. Es ist nicht möglich, den Kontextknoten explizit für die Berechnung von Ausdrücken zu ändern.

Damit ist klar, dass Schritte, die beliebige Knoten berechnen, innerhalb eines Pfades in STX nicht realisierbar sind. Zwar können einzelne Knoten zum Beispiel als Werte von Variablen gespeichert werden, jedoch kann von diesen Knoten nicht mehr zu ihren Vorfahren navigiert werden.

Um die XPath-Grammatik entsprechend zu ändern, wird zunächst eine redundante Erweiterung vorgenommen. Offenbar lässt sich das Symbol *ValueExpr* unter Anwendung der (originalen) Produktionsregeln 35, 36, 37 und 38 in das Symbol *FilterStep* überführen. Die Erweiterung der Produktionsregel 35 um eine zusätzliche Alternative auf der rechten Seite, durch die eine direkte Ersetzung durch das Symbol *FilterStep* ermöglicht wird, ist damit redundant und ändert die definierte Sprache nicht.

Das anschließende Entfernen der Alternative *FilterStep* in der Produktionsregel 38 lässt nun die durch *FilterStep* definierten Ausdrücke nicht mehr als Pfadbestandteile zu. Sie sind jedoch durch die vorherige Erweiterung der Produktionsregel 35 weiterhin als allein stehende Ausdrücke möglich. Pfade in STXPath enthalten damit ausschließlich (abgekürzte) Schritte entlang der verfügbaren Achsen.

Einschränkung der Anzahl der Prädikate

Innerhalb von STXPath-Pfaden kann in jedem Schritt maximal ein Prädikat angegeben werden. In Kapitel 4.3 wurde bereits dargestellt, dass mehrfache Prädikate allein bei der Auswertung von Positionsinformationen von Bedeutung sind. In einem seriellen Transformationsprozess sind jedoch Informationen über bereits verarbeitete Knoten für einen Vergleich nicht mehr verfügbar. Demzufolge kann der STX-Prozessor im Nachhinein nicht mehr bestimmen, für wieviele Knoten der Eingabe ein konkretes Prädikat zutraf, um eine Positionsangabe bezüglich dieses Prädikates zu ermitteln.[8] Positionen bezüglich des Elternknotens (für eine Abfrage innerhalb des ersten Prädikats) werden mittels einfacher Zähler für die jeweiligen Knotentests realisiert.

[8]Korrekterweise muss hier gesagt werden, dass es auch in einer seriellen Verarbeitung möglich ist, die sich aus mehrfachen Prädikaten ergebenden Positionsangaben zu bestimmen. Für jedes solche Prädikat muss ein Zähler verwaltet werden, der bei Erfüllung dieses Prädikats um eins erhöht wird und somit die Position für das folgende Prädikat enthält. Dies hätte jedoch einen nicht unerheblichen Performanceverlust zur Folge, weil für jeden Knoten der Eingabe im gesamten Transformations-Sheet alle Pfade mit mehrfachen Prädikaten ausgewertet werden müssen, unabhängig davon, welches konkrete Template jeweils benutzt wird.

Aus diesem Grund wird die Regel 44 so abgewandelt, dass sie maximal ein Prädikat erzeugt.

Prädikate sind jedoch nicht auf Schritte in einem Pfad beschränkt, sondern können ebenso auf Sequenzen angewendet werden. Hier sind mehrere Prädikate jedoch unkritisch, da alle Werte der Sequenz für eine Auswertung zur Verfügung stehen. In der Grammatik ist das Symbol *PrimaryExpr* das Startsymbol für alle in STXPath möglichen Ausdrücke, die primär keine Pfade sind und somit eine beliebige Sequenz liefern. In der Regel 40 wird daher auf der rechten Seite dem Symbol *Predicate* der Wiederholungsoperator nachgestellt, sodass auf diese Weise der vorherige Zustand wieder hergestellt wird.

Entfernung spezieller Operationen auf Knoten

Einige Operationen in XPath 2.0 operieren auf Knotenmengen. In STX werden aber maximal die aktuellen Vorfahrenknoten in einem Schritt ausgewählt. Operationen, die Knotenmengen vereinigen oder den Durchschnitt bzw. die Differenz bilden, sind für die eingeschränkten STX-Knotenmengen nur von geringem praktischen Wert. Sie werden deshalb in STXPath nicht unterstützt. Das gleiche gilt für die Knotenoperationen, die zwei Knoten auf Identität prüfen oder für zwei Knoten ihre relative Ordnung im Dokument bestimmen.

In der Grammatik entfallen damit die mehrfach optionalen Bestandteile (*-Operator) der Produktionsregeln 33 und 34 (`union`, `intersect` und `except`). Darüber hinaus werden die Produktionsregeln 46 und 47 vollständig gestrichen und die Produktionsregel 28 entsprechend modifiziert.

5.4.2 STX-Muster

Die Grammatik für Muster in XSLT ist auf den ersten Blick wesentlich kürzer als die für XPath. Tatsächlich verwendet jedoch die Muster-Grammatik die XPath-Grammatik, indem sie auf einige ihrer Nichtterminalsymbole zurückgreift. Die Produktionsregeln für Muster in STX sind als Bestandteil der STXPath-Grammatik ebenfalls im Anhang B.2 beginnend bei Regel 81 (*Pattern*) enthalten.

Muster in STX gleichen bis auf wenige Einschränkungen den aus XSLT bekannten Mustern. Von Prädikaten abgesehen enthält ein solches Muster nur Schritte, die entlang der Vorfahren-Achse ausgewertet werden können; es wird weder auf vorangehende (im Sinne der Achse `preceding`) noch auf folgende (im Sinne von `following` und `descendant`) Knoten zugegriffen. Dies ist darauf zurückzuführen, dass Muster allein die Achsen `child` und `attribute` sowie alle definierten Abkürzungen einschließlich des Doppelschrägstrichs (`//`) enthalten dürfen.

Die vorgenommenen Änderungen lassen sich wie folgt zusammenfassen:

- Muster in STX benutzen ausschließlich die abgekürzte Syntax. Da diese semantisch der ausführlichen Syntax völlig gleichwertig ist, kann auf die explizite Angabe der Achsen `child` und `attribute` hier gänzlich verzichtet werden (Regel 85). Dies ist konsistent mit den Pfaden in STXPath, die ebenfalls keine expliziten Achsen besitzen.
- In jedem Schritt kann maximal ein Prädikat angegeben werden. Diese Einschränkung wurde bereits im Zusammenhang mit Pfaden diskutiert und in der STXPath-Regel 44 realisiert.

- Ein Muster darf in STX nicht mit einem Aufruf der Funktionen `id` oder `key` beginnen. Diese Funktionen liefern in XSLT diejenigen Knoten des Dokumentes, die zu einem bestimmten Wert (einem eindeutigen Bezeichner oder einem selbst definierten Schlüssel) gehören. Beide Funktionen setzen jedoch eine Gesamtsicht auf das Eingabedokument voraus, die in STX nicht verfügbar ist.

  Aus diesem Grund wurden die Produktionsregeln 86–88 entfernt und die rechte Seite der Regel 82 entsprechend modifiziert.

- Schließlich wurden Muster, die mit einem doppeltem Schrägstrich beginnen, aus STX entfernt. Solche Muster sind redundant, da jeder Knoten ein Nachkomme des Dokumentknotens ist. Ein Knoten, der auf das Muster `//foo` passt, passt auch immer auf das Muster `foo`. Dies gilt im Übrigen auch in XSLT 1.0.

  In der Regel 82 wurde deshalb die dritte Alternative auf der rechten Seite gestrichen.

Genauso wie XPath_S eine Teilmenge von XPath ist, sind die hier beschriebenen Muster für STX damit eine Teilmenge der Muster von XSLT. Mehr noch: sie gleichen sogar den für XMLTK (siehe Kapitel 3.3.5) definierten XPath-Mustern. Wie XMLTK demonstriert hat, lassen sich solche Muster effizient, d.h. in konstanter, nicht von der Anzahl der Muster abhängenden Zeit, mit Hilfe endlicher Automaten auswerten (so genanntes *Pattern Matching*).

5.4.3 Erweiterte Knotentests

Im Kapitel 5.3.3 wurden als spezielle Erweiterung des XPath-Datenmodells die zusätzlichen Knotentypen CDATA (für CDATA-Abschnitte) und DOCTYPE (für die Dokumenttyp-Deklaration) beschrieben. Die Sprache STXPath benötigt daher ein passendes syntaktisches Konstrukt, um auf diese Knoten zuzugreifen.

CDATA-Knoten

CDATA-Knoten sind ein Spezialfall von Textknoten. Der Knotentest `text()` ist somit auch für CDATA-Knoten erfüllt. Darüber hinaus stellt STXPath den speziellen Knotentest `cdata()` zur Verfügung, der nur für CDATA-Knoten erfüllt ist. Die Produktionsregel 65 wird dazu um die Alternative *CdataTest* erweitert. Dieses neue Symbol wird anschließend in der zusätzlichen Regel 78a definiert.

DOCTYPE-Knoten

DOCTYPE-Knoten lassen sich über den Knotentest `doctype()` auswählen. Analog zur Erweiterung um CDATA-Knoten wird daher die Regel 65 auf der rechten Seite um die Alternative *DoctypeTest* ergänzt und die Regel 78b als zusätzliche Regel in die Grammatik aufgenommen. Da DOCTYPE-Knoten nur als Kindknoten des Dokumentknotens auftreten können, sind Pfade, die diesen neuen Knotentest innerhalb einer längeren Schrittfolge enthalten, wenig sinnvoll.

Die Sprache STXPath ist nun zwar keine echte XPath-Teilmenge mehr, allerdings wurden nur sehr geringfügige Erweiterungen vorgenommen. Alle Pfadausdrücke bzw. STX-Muster, die keinen der Knotentests `cdata()` und `doctype()` enthalten, sind ebenfalls korrekte XPath-Ausdrücke.

5.5 Speicheranforderungen

Aus den bisherigen Ausführungen lassen sich die folgenden Faktoren für das Speicherverhalten eines STX-Prozessors ableiten:

1. **Tiefe des XML-Baumes**
   Während der Verarbeitung des Baumes werden alle geöffneten Elemente auf einem Vorfahren-Stack abgelegt und sind dann über Pfadausdrücke erreichbar. Die maximale Tiefe des Baumes bestimmt die maximale Größe dieses Stack. Wie in Kapitel 5.2 bereits dargestellt wurde, wachsen XML-Dokumente jedoch typischerweise in der Breite und nicht in der Tiefe.

2. **Größe der Knoten des Baumes**
   Wenn man von theoretisch denkbaren beliebig langen Element- und Attributnamen absieht, wird der für einen Elementknoten benötigte Speicherplatz in erster Linie vom Inhalt seiner Attribute bestimmt. Zwar muss für jedes Element außerdem Platz für dessen Zeichenkettenwert bereitgestellt werden, dieser ergibt sich jedoch direkt aus der Größe eines folgenden Textknotens und wird daher hier nicht gesondert betrachtet. Die Attribute eines Elements dienen in der Regel der Aufnahme von kurzen Meta-Informationen über den Elementinhalt. Obwohl es in XML prinzipiell möglich ist, sehr lange Attributwerte zu verwenden, kommt dies in der Praxis kaum vor. Der zweite in diesem Zusammenhang wichtige Faktor, die Anzahl der Attribute, ist ebenfalls beschränkt, da sich Attribute nicht wiederholen dürfen und ein Schema immer nur eine endliche (und überschaubare) Anzahl erlaubter Attribute definiert.

   Textknoten treten in STX in zwei Varianten auf: zum einen können sie (wie in XPath) einen zusammenhängenden Textabschnitt repräsentieren, zum anderen können sie jedoch auch zeilenweise gebildet werden. Diese zweite Variante bewirkt, dass die Größe der Textknoten von der Länge der Zeilen im Eingabedokument abhängt.

   Der für CDATA-, Kommentar- und Verarbeitungsanweisungsknoten benötigte Speicherplatz entspricht der Anzahl der in diesen Knoten enthaltenen Zeichendaten. Für Verarbeitungsanweisungen ist das in der Praxis unproblematisch, bei sehr langen Kommentaren oder CDATA-Abschnitten könnte jedoch der zur Verfügung stehende Speicher erschöpft werden.

   Die Knotentypen Dokumentknoten und DOCTYPE-Knoten besitzen eine feste Größe und sind daher unkritisch.

3. **Anzahl unterschiedlicher Geschwister-Elemente**
   Für jeden Elementtyp der Eingabe wird innerhalb des jeweiligen Elternelements ein Positionszähler verwaltet. Die Anzahl solcher Zähler korreliert folglich mit der Anzahl unterschiedlicher Elemente unterhalb des gleichen Elternelements. In der Praxis ist diese Anzahl jedoch beschränkt. Insbesondere gilt, wie bereits für Attribute, dass ein Schema immer nur eine endliche und überschaubare Anzahl von Elementtypen definiert. Darüber hinaus kann ein STX-Prozessor eine STX-Transformation dahingehend optimieren, dass nur die abgefragten Zähler tatsächlich verwaltet werden.

4. **Art und Inhalt der verwendeten Variablen**
   Der in STX unterstützte Sequenztyp ermöglicht dem Anwender, beliebig lange Sequenzen zu konstruieren und auf diese Weise jede erdenkliche Speichergrenze

zu überschreiten. Gleiches gilt für den Spezialfall der Variablen, die XML-Fragmente aufnehmen können (genannt *Puffer*, siehe Kapitel 5.6.5). Hier wird das Speicherverhalten jedoch nicht von der XML-Eingabe, sondern von der Art der STX-Transformation bestimmt. Es liegt damit in der Verantwortung des STX-Autors, solche Konstruktionen ausschließlich sinnvoll einzusetzen bzw. ganz zu vermeiden.

Für jede feste Größe an verfügbarem Arbeitsspeicher lässt sich damit künstlich ein XML-Dokument konstruieren, das mit STX nicht mehr verarbeitet werden kann. Solche Dokumente stellen jedoch Extremfälle dar, die in der Praxis nicht vorkommen:

- extrem tiefe Verschachtelung (der XML-Baum entspricht einem langen Pfad),
- sehr lange Attributwerte,
- sehr große Kommentarbereiche,
- sehr lange Textpassagen ohne Zeilenumbruch.

Mit »sehr lang« ist hier jeweils gemeint, dass der gesamte zur Verfügung stehende Speicher durch die entsprechenden Zeichendaten (in einem Attributwert, Kommentar oder einer Textzeile) aufgebraucht wird.

Die in der Praxis vorkommenden XML-Dokumente besitzen diese Eigenschaften in der Regel nicht und können daher mit STX transformiert werden.

5.6 STX-Konzepte

Da die Ähnlichkeit zu XSLT ein wesentlicher Aspekt bei der Entwicklung der Transformationssprache STX war, finden sich viele XSLT-Elemente ebenfalls in STX wieder. Diese Elemente besitzen den gleichen lokalen Namen und die gleichen Attribute, sie befinden sich jedoch im STX-Namensraum (z.B. `stx:value-of`). Mit XSLT vertraute Anwender werden somit in die Lage versetzt, ohne großen Lernaufwand STX-Transformations-Sheets zu erstellen.

Bei einigen aus XSLT bekannten Elementen wurde die Menge der möglichen Attribute leicht eingeschränkt oder erweitert (z.B. bei `stx:transform`). Alle anderen STX-Elemente bieten entweder eine vollständig neue Funktionalität oder mussten aufgrund des für STX geltenden Verarbeitungsmodells abweichend definiert werden (z.B. `stx:process-children`). Bei STX-Elementen mit einer abweichenden Semantik gegenüber ihrem XSLT-Äquivalent wurde dies durch die Wahl eines anderen Namens explizit gekennzeichnet (`stx:for-each-item`).

Im Folgenden werden alle Elemente aus dem STX-Namensraum vorgestellt. Für die aus XSLT bekannten Anweisungen genügt in der Regel deren Nennung. Der Schwerpunkt der anschließenden Diskussion liegt auf den von XSLT abweichenden Anweisungen und deren Eigenschaften.

Für die weiteren Details ist es wichtig, die XSLT-Version anzugeben. STX und XSLT 2.0 haben sich zum Teil parallel entwickelt. Da XSLT 2.0 derzeit noch nicht fertig gestellt ist, beziehen sich die folgenden Aussagen in der Regel auf XSLT 1.0. Da teilweise bereits Eigenschaften aus XSLT 2.0 in STX übernommen wurden, wird an entsprechender Stelle darauf hingewiesen.

5.6.1 Aus XSLT übernommene Elemente

Dieses Kapitel stellt die aus XSLT übernommenen Elemente kurz vor. Eigenschaften, die ebenfalls für die jeweiligen XSLT-Pendants zutreffen, werden an dieser Stelle nicht genauer besprochen. Stattdessen liegt der Schwerpunkt auf den in STX zu findenden Abweichungen.

`stx:transform`
ist das Wurzelelement eines STX-Transformations-Sheet. Es besitzt – ebenso wie in XSLT – die Attribute `version` und `exclude-result-prefixes`. Für das Attribut `stxpath-default-namespace` existiert ein ähnlich lautendes Vorbild in XSLT 2.0.

Die beiden Attribute `output-method` (mit den Werten "`xml`" und "`text`") und `output-encoding` legen Ausgabeeigenschaften fest, die in XSLT über ein separates Element `xsl:output` definiert werden müssen.

Über die darüber hinaus vorhandenen Attribute `pass-through`, `recognize-cdata`, `strip-space` und `text-by-lines` lassen sich grundlegende Transformationseigenschaften auf Gruppenebene vereinbaren. Sie werden im Kapitel 5.6.4 vorgestellt.

Da STX als Transformationssprache keinen speziellen *Style*-Bezug besitzt, existiert – anders als in XSLT – auch kein `stylesheet`-Element als Synonym zu `stx:transform`. Nichtsdestoweniger können mit STX ebenso Daten in ein Präsentationsvokabular wie XHTML oder XSLFO umgewandelt werden.

`stx:template`
definiert ein Template in STX. Templates arbeiten genauso wie in XSLT. Mit Hilfe eines Musters im `match`-Attribut werden die zu behandelnden Knoten beschrieben. Durch ein optionales `priority`-Attribut kann jedem Template eine spezielle Priorität zugewiesen werden.

Modes für Templates (Attribut `mode`) sind in STX nicht mehr notwendig, da diese Funktionalität durch benannte Gruppen abgedeckt wird (siehe Kapitel 5.6.4).

Eine neue Anweisung `stx:procedure` übernimmt die Funktion benannter Templates; das Attribut `name` gibt es für `stx:template` daher nicht. Diese neue Anweisung wurde eingeführt, um eine klare Trennung zwischen Templates (die über Muster automatisch ausgewählt werden) und Prozeduren (die explizit über ihren Namen aufgerufen werden) zu erreichen. Damit wird Missverständnissen über die Funktionsweise von Templates vorgebeugt, die insbesondere bei Anfängern auftreten können.

`stx:call-procedure`
gibt es mit diesem Namen in XSLT nicht. Es handelt sich hier um die STX-Entsprechung des Elements `xsl:call-template`. Da benannte Templates in STX durch ein eigenes Element `stx:procedure` repräsentiert werden, wurde auch die dazugehörige Aufrufanweisung entsprechend umbenannt.

`stx:namespace-alias`
ermöglicht die Festlegung eines speziellen Alias-Namensraums für literale Elemente oder Attribute aus dem Transformations-Sheet. Diese werden während der Transformation wie Elemente bzw. Attribute aus dem durch `stx:namespace-alias` zugewiesenen Namensraum behandelt. Das in der XSLT-Vari-

ante vorhandene Attribut `stylesheet-prefix` wurde in STX aus nahe liegenden Gründen in `sheet-prefix` umbenannt.

Diese Anweisung ermöglicht beispielsweise die Erzeugung von STX-Code durch ein STX-Transformations-Sheet.

`stx:value-of`
wertet den im `select`-Attribut angegebenen Ausdruck aus und kopiert das Ergebnis als Text in die Ausgabe.

Diese Anweisung unterstützt das in XSLT 2.0 eingeführte Attribut `separator`, über das ein spezielles Trennsymbol für die Ausgabe von Sequenzen angegeben werden kann. Fehlt dieses Attribut, werden alle Elemente der Sequenz ohne Trennsymbol ausgegeben. Damit verhält sich `stx:value-of` für Sequenzen (bzw. Knotenmengen) anders als `xsl:value-of` in XSLT 1.0.

`stx:text`
kopiert den Inhalt als Text in die Ausgabe.

Das zusätzliche Attribut `markup` definiert, wie Nicht-Textknoten innerhalb von `stx:text` behandelt werden sollen. Der Wert "`ignore`" ignoriert alle anderen Knoten, der Wert "`error`" führt in diesem Fall zu einem dynamischen Fehler. Mittels "`serialize`" wird die textuelle Repräsentation des Inhalts erzeugt, d.h. seine Serialisierung als XML-Text.

`stx:element`
erzeugt dynamisch ein XML-Element. Diese Anweisung wurde originalgetreu aus XSLT übernommen.

`stx:attribute`
erzeugt dynamisch ein XML-Attribut. Diese Anweisung wurde originalgetreu aus XSLT übernommen.

`stx:comment`
erzeugt dynamisch einen Kommentar. Diese Anweisung wurde originalgetreu aus XSLT übernommen.

`stx:processing-instruction`
erzeugt dynamisch eine Verarbeitungsanweisung. Auch diese Anweisung entspricht dem Original aus XSLT.

`stx:copy`
kopiert wie in XSLT den aktuellen Knoten, den Kontextknoten, in die Ausgabe.

Da STX nicht auf Bäumen operiert, besitzt es auch keine Anweisung `copy-of`. Das Kopieren eines vollständigen Unterbaumes kann jedoch durch die Anweisung `stx:process-children` (siehe Kapitel 5.6.3) und einer leeren "copy"-Gruppe (siehe Listing 7 auf Seite 85 im Kapitel 5.6.4) erfolgen. Auf die gleiche Art und Weise müssten Attribute per `stx:process-attributes` kopiert werden.

Um jedoch die Lösung dieses häufigen Anwendungsfalls zu vereinfachen, verfügt `stx:copy` über ein zusätzliches Attribut `attributes`. Sein Wert ist ein STX-Muster, auf das alle zu kopierenden Knoten passen müssen.

Beispielsweise kopiert `<stx:copy>` den Kontextknoten ohne Attribute und `<stx:copy attributes="@*">` den Kontextknoten einschließlich aller Attribute. `<stx:copy attributes="@src | @alt">` würde neben dem

Element selbst nur dessen `src` und `alt`-Attribute kopieren und alle anderen übergehen.

`stx:variable`
deklariert eine Variable. Falls dieses Element mit Inhalt verwendet wird (und damit ohne `select`-Attribut), wird die Zeichenkette, die sich aus diesem Inhalt ergibt, als Wert der Variablen verwendet. Der in XSLT an dieser Stelle verwendete Typ Ergebnisbaum-Fragment (*result tree fragment*)[9] ist in STX nicht vorhanden.

Darüber hinaus ist zu beachten, dass Variablen in STX mit Hilfe der zusätzlichen Anweisung `stx:assign` geändert werden können (siehe Kapitel 5.6.2).

`stx:param`
deklariert einen Parameter wie in XSLT. Parameter sind spezielle Variablen, für die bei einem Aufruf mittels `stx:with-param` ein Wert übergeben werden kann.

Das Element `stx:param` darf als Kind von `stx:template` (als Template-Parameter), `stx:transform` (als globaler Parameter) und darüber hinaus von `stx:procedure` (siehe Erläuterung bei `stx:template`), `stx:group` (siehe Kapitel 5.6.4) und `stx:recover` (siehe Kapitel 5.6.8) angegeben werden.

Enthält ein `stx:param`-Element Inhalt, gelten die gleichen Regeln wie für `stx:variable`.

Parameter können über das Attribut `required` und die möglichen Werte "`yes`" und "`no`" als obligatorisch oder optional gekennzeichnet werden. Dieses Attribut wurde aus dem Entwurf für XSLT 2.0 übernommen.

`stx:with-param`
übergibt einen Parameter an ein Template oder eine Prozedur. Enthält dieses Element Inhalt, gelten die gleichen Regeln wie für `stx:variable`.

`stx:if`
testet die Bedingung in ihrem `test`-Attribut und führt bei Erfüllung dieser Bedingung die Anweisungen aus dem Inhalt aus.

Diese Anweisung wurde originalgetreu aus XSLT übernommen. In STX kann jedoch zusätzlich ein nachfolgendes Element `stx:else` angegeben werden.

`stx:choose`, **`stx:when`**, **`stx:otherwise`**
dienen zur Notation einer Fallunterscheidung. Diese Anweisungen wurden originalgetreu aus XSLT übernommen.

`stx:for-each-item`
iteriert über eine im `select`-Attribut anzugebende Sequenz von Werten. Diese Anweisung wurde der XSLT-Variante `xsl:for-each` nachempfunden. In STX kommt jedoch als grundlegender Unterschied gegenüber XSLT zum Tragen, dass der Kontextknoten innerhalb von `stx:for-each-item` nicht geändert wird. Dieser bleibt immer der durch das aktuelle Template bearbeitete Knoten (siehe Kapitel 5.2). Aus diesem Grund wurde zur Abgrenzung ein abgewandelter Name gewählt.

[9] Ab der Version 2.0 von XSLT wird es diesen Typ nicht mehr geben. Stattdessen wird der Wert einer auf diese Weise deklarierten Variablen eine Knotenmenge sein, die die Wurzel eines temporären Baumes enthält. Da STX keine XML-Bäume verarbeitet, ist diese neue Semantik in STX jedoch ebenfalls nicht anwendbar. Ein gewissen Ersatz bieten Puffer, die in Kapitel 5.6.5 vorgestellt werden.

Der Zugriff auf die einzelnen Elemente in der Sequenz erfolgt hier nicht über den Ausdruck ".", sondern über eine im `name`-Attribut anzugebende Laufvariable. Der folgende Ausschnitt demonstriert dies:

```
<stx:for-each-item name="i" select="(1,2,3)">
  <nr><stx:value-of select="$i" /></nr>
  ...
</stx:for-each-item>
```

`stx:message`
produziert eine Nachricht. Diese Anweisung wurde um Logging-Fähigkeiten erweitert, siehe Kapitel 5.6.8.

`stx:include`
schließt ein anderes STX-Transformations-Sheet in das aktuelle Transformations-Sheet ein. Diese Anweisung darf in STX auf Gruppenebene verwendet werden und wird wie in XSLT zur Compile-Zeit ausgewertet. Das Gruppenkonzept von STX bewirkt jedoch eine leicht veränderte Semantik, sodass das eingeschlossene Transformations-Sheet eine neue Gruppe bildet, siehe Kapitel 5.6.4.

`stx:result-document`
erzeugt ein neues Ausgabedokument. Die innerhalb dieser Anweisung erzeugten Knoten werden in dieses neue Dokument geschrieben. Die Anweisung `stx:result-document` existiert in XSLT 1.0 noch nicht, wird jedoch ab der Version 2.0 Bestandteil von XSLT sein.

Im Vergleich zur im derzeitigen Entwurf von XSLT 2.0 enthaltenen Definition gibt es in STX jedoch kein `format`-Attribut, das auf ein `xsl:output`-Element verweist. Stattdessen werden Ausgabeeigenschaften direkt in den Attributen `output-method` und `output-encoding` angegeben. Sie besitzen damit für das neue Dokument die gleiche Bedeutung wie die gleichnamigen Attribute in `stx:transform` für die Hauptausgabe.

Die in XSLT 2.0 vorgesehenen Attribute `type` und `validation` existieren in STX ebenfalls nicht, da STX keine Kenntnisse über ein eventuelles Schema besitzt und daher die produzierte Ausgabe nicht validieren kann.

Nicht übernommene XSLT-Elemente

STX bietet nicht für alle Anweisungen aus XSLT 1.0 eine Entsprechung. Einige Elemente lassen sich aufgrund des seriellen Charakters der durch STX beschriebenen Transformation nicht realisieren. Andere Elemente wurden nicht berücksichtigt, um den Umfang der neuen Sprache nicht zu groß werden zu lassen. Solche Elemente können bei Bedarf von Seiten der Anwender in den Sprachumfang aufgenommen werden.

Es folgt eine kurze Liste der unberücksichtigten XSLT-Elemente:

`xsl:apply-templates`
Diese Anweisung kann in ihrer Originalform nicht in STX eingesetzt werden, da über ihr `select`-Attribut beliebig im Baum navigiert werden kann. Eine Variante für STX ohne dieses Attribut wäre zu restriktiv.

In Kapitel 5.6.3 wird ausführlich diskutiert, durch welche speziellen STX-Elemente diese Anweisung ersetzt wurde.

`xsl:import`
Das Importieren von Transformations-Sheets (im Gegensatz zum einfachen Einschließen per `stx:include`) wird in STX derzeit nicht unterstützt. Ein analoger Mechanismus wäre jedoch auch hier leicht realisierbar.

`xsl:attribute-set`, **`xsl:decimal-format`**
Auch diese Anweisungen könnten ohne Probleme in STX spezifiziert werden. In XSLT erleichtern sie jedoch in erster Linie die Erzeugung von Präsentationsformaten. Ihre Funktionalität kann bereits vollständig mit den in STX vorhandenen Mitteln erreicht werden.

`xsl:output`, **`xsl:strip-space`**, **`xsl:preserve-space`**
STX enthält vereinfachte Versionen der durch diese Elemente erbrachten Funktionalität. Ausgabeeigenschaften von `xsl:output` werden direkt in `stx:transform` angegeben; das Entfernen von Leerraum ist als Gruppeneigenschaft über das Attribut `strip-space` (siehe Kapitel 5.6.4) möglich.

`xsl:key`, **`xsl:number`**, **`xsl:sort`**
Für diese Elemente gibt es aufgrund des seriellen Charakters von STX keine Entsprechungen.

Das Element `xsl:key` erstellt einen Index über dem gesamten Eingabedokument, welches in STX jedoch nicht zur Verfügung steht.

`xsl:number` generiert eine laufende Nummer für ein Element. Diese Aufgabe kann in STX jedoch durch die Benutzung einer eigenen Zählvariablen gelöst werden. Es wäre sehr ineffizient, den STX-Prozessor alle potenziellen Zähler mitführen zu lassen.

Die Anweisung `xsl:sort` ist aus nahe liegenden Gründen ebenfalls nicht in STX vorhanden, da keine Gesamtsicht auf das Dokument vorhanden ist und somit Knoten des Eingabedokuments nicht sortiert werden können.

`xsl:fallback`
Dieses Element dient dazu, ein Ausweichverhalten für implementationsspezifische Erweiterungselemente zu definieren, falls der verwendete XSLT-Prozessor diese nicht unterstützt. Prinzipiell sind solche Erweiterungselemente ebenfalls in STX denkbar, sie wurden aber im aktuellen Sprachentwurf nicht berücksichtigt.

Wie in Kapitel 5.6.8 dargestellt wird, enthält STX einen allgemeinen Fehlerbehandlungsmechanismus, der auch solche Fehler abfangen kann. Auf ein `fallback`-Element kann daher in STX gänzlich verzichtet werden.

5.6.2 STX als prozedurale Sprache

Änderbare Variablen

STX verarbeitet sequentiell einen XML-Datenstrom und muss es daher dem STX-Programmierer erlauben, einen Zustand mitzuführen. Der funktionale Charakter von XSLT ist für STX nicht möglich, da die XML-Daten nicht als Gesamtheit den einzelnen Schritten in der Transformation zur Verfügung stehen. Der Ablauf einer Transformation wird durch den Eingabestrom gesteuert und ist dadurch klar bestimmt.[10]

[10] In XSLT können dagegen einzelne Schritte der Transformation auch parallel ausgeführt werden, da diese sich nicht gegenseitig beeinflussen (XSLT ist frei von Seiteneffekten).

Infolgedessen sind in STX änderbare Variablen notwendig, die den aktuellen Zustand der Transformation repräsentieren. Variablen werden wie aus XSLT bekannt durch die Anweisungen `stx:variable` bzw. `stx:param` (als Parameter) deklariert. Ihr Wert kann anschließend jedoch durch die neue Anweisung `stx:assign` geändert werden. Ihre Syntax orientiert sich an `stx:variable`. Der Name der zu ändernden Variablen wird im `name`-Attribut angegeben. Der neue Wert kann wahlweise im Attribut `select` oder im Inhalt des Elements angegeben werden. Das Heraufsetzen eines Zählers `i` wird z.B. in STX folgendermaßen notiert:

```
<stx:assign name="i" select="$i + 1" />
```

Schleifen

STX enthält eine Schleifenanweisung `stx:while`, mit der zyklisch eine Folge von STX-Anweisungen mehrfach nacheinander ausgeführt werden kann. Ihre Syntax orientiert sich an `stx:if`. Der Inhalt von `stx:while` wird solange ausgeführt, wie die Berechnung des Ausdrucks im Attribut `test` den Wert *wahr* ergibt. Demzufolge sollte der getestete Ausdruck auf eine Variable zugreifen, die innerhalb der `stx:while`-Anweisung geändert wird.

Die folgenden Zeilen generieren beispielsweise HTML-Spaltenüberschriften mit den Zahlenwerten von 1 bis 9

```
<stx:variable name="i" select="1" />
<stx:while test="$i &lt; 10">
  <th><stx:value-of select="$i" /></th>
  <stx:assign name="i" select="$i + 1" />
</stx:while>
```

Getrennte Start- und Endtags

Der serielle Charakter der Transformation kann es erforderlich machen, Beginn und Ende eines Elements getrennt voneinander zu erzeugen. Vor allem bei Gruppierungsproblemen, wie sie in Kapitel 5.7.6 dargestellt werden, tritt dieser Fall auf. An der Stelle, an der der Beginn des neuen Elements erzeugt werden muss, kann noch nicht bestimmt werden, wieviele Daten aus dem Eingabestrom verarbeitet werden müssen, bevor das Element wieder geschlossen werden kann.

Aus diesem Grund stellt STX die beiden Anweisungen `stx:start-element` und `stx:end-element` zur Verfügung. Ihre Namen lehnen sich an die entsprechenden Events aus dem SAX-API [SAX] an. Beide Elemente besitzen das Attribut `name` zur Angabe des Elementnamens sowie das optionale Attribut `namespace` für die Angabe eines Namensraumes. Ihre Syntax orientiert sich damit an der bereits bekannten Anweisung `stx:element`.

In einer STX-Transformation ist damit folgendes Vorgehen möglich:

```
<stx:template match="start">
  <stx:start-element name="foo" />
</stx:template>

<stx:template match="end">
  <stx:end-element name="foo" />
</stx:template>
```

Innerhalb des ersten Template wird das Element `foo` geöffnet, innerhalb des zweiten Elements wird es wieder geschlossen. Das Kapitel 5.7.6 enthält zwei ausführlichere Beispiele für diese Anweisungen.

Erhöhte Fehleranfälligkeit

Die Verwendung von `stx:start-element` und `stx:end-element` birgt jedoch die Gefahr schwer wartbaren und fehleranfälligen STX-Codes. Der Zusammenhang zwischen Beginn und Ende eines Ergebnis-Elementes ist nicht klar ersichtlich. Häufig ist es sehr aufwändig, die Ursache für ein fehlplatziertes Elementende herauszufinden.

Durch den STX-Prozessor wird allerdings sichergestellt, dass das Ergebnis in jedem Fall wohlgeformtem XML entspricht. Der Versuch, ein Element zu schließen, für das es kein öffnendes Tag gibt, wird als dynamischer Fehler gemeldet. Dies würde auch in folgendem Fall geschehen:

```
<foo>
  <stx:end-element name="foo" />
</foo>
```

5.6.3 Traversieren der XML-Eingabe

Sowohl XSLT als auch STX benutzen den Template-Mechanismus, um einzelne Knoten zu transformieren. Ein Template bestimmt über ein Muster, welche Knoten durch dieses Template bearbeitet werden können. Innerhalb des Template ist dann eine Anweisung erforderlich, die dem Prozessor mitteilt, dass die Verarbeitung der XML-Eingabe fortgesetzt werden soll. Üblicherweise werden hier alle Kindknoten abgearbeitet, bevor das Template selbst beendet wird. Diese Vorgehensweise entspricht der typischen Rekursion, wobei die Auswahl des rekursiv aufgerufenen Template durch den Prozessor erfolgt.

Ersatz für `xsl:apply-templates`

In XSLT heißt diese Anweisung `xsl:apply-templates`. Diese besitzt ein zusätzliches `select`-Attribut, über das beliebige Knotenmengen ausgewählt werden können, für die der XSLT-Prozessor passende Templates suchen soll. `xsl:apply-templates` kann beliebig oft innerhalb eines Template verwendet werden.

Da STX nicht frei auf den gesamten XML-Baum zugreifen kann, ist es nicht möglich, beliebig ausgewählte Knoten zu bearbeiten. Vielmehr müssen die Knoten in der Reihenfolge ihres Auftretens im XML-Dokument bearbeitet werden. Ein explizites Ändern des Kontextknotens in einen beliebigen anderen Knoten ist nicht möglich.

Ein einfaches Entfernen des `select`-Attributes für STX würde dieses Problem beseitigen. Allerdings wäre die Verarbeitung anderer Knoten dann auf die Kindknoten beschränkt. Das STX-Verarbeitungsmodell erlaubt jedoch ebenfalls die Verarbeitung der Attribute, da diese beim Beginn eines Elements bekannt sind. Es ermöglicht ebenso die Verarbeitung folgender Geschwisterknoten oder auch die Verarbeitung anderer XML-Dokumente. Darüber hinaus erweist es sich als nützlich, den aktuellen Knoten selbst durch das nächste passende Template bearbeiten zu lassen. In XSLT werden diese Fälle durch geeignete Ausdrücke im `select`-Attribut der Anweisung `xsl:apply-templates` gelöst, für externe Dokumente beispielsweise durch den Aufruf der Funktion `document`. STX benötigt an dieser Stelle ein anderes Sprachmittel.

Zu diesem Zweck existieren in STX mehrere Anweisungen, die jeweils eine bestimmte Gruppe von Knoten bearbeiten.

`stx:process-children`
verarbeitet die Kindknoten.

Innerhalb eines Template darf diese Anweisung nur einmal ausgeführt werden.

`stx:process-attributes`
verarbeitet die Attribute.

Da die Attribute eines Elements bei dessen Verarbeitung durch STX vollständig bekannt sind, kann diese Anweisung ohne weiteres mehrmals aufgerufen werden.

`stx:process-siblings`
verarbeitet nachfolgende Geschwisterknoten.

Diese Anweisung ermöglicht beispielsweise das Gruppieren benachbarter Knoten in einem gemeinsamen Elternelement (siehe auch Kapitel 5.7.6). Dies wäre ohne diese Anweisung und ohne die Möglichkeit, Start und Ende eines Elementes getrennt zu erzeugen, nicht möglich. Nach einem `stx:process-siblings` darf kein `stx:process-children` aufgerufen werden, da die Kindknoten in diesem Fall bereits abgearbeitet worden sind.

Mehrere `stx:process-siblings`-Anweisungen sind jedoch ohne weiteres gestattet. Über die Attribute `while` und `until` kann festgelegt werden, welche Geschwisterknoten verarbeitet werden sollen. Mit einem Muster im `while`-Attribut werden alle Knoten verarbeitet, die auf dieses Muster passen; mit einem Muster im `until`-Attribut werden solange die Geschwister verarbeitet, bis ein Knoten auf das Muster passt. Die Verarbeitung endet in jedem Fall, wenn das Ende des Elternelements erreicht ist und damit alle Geschwister abgearbeitet worden sind.

`stx:process-self`
verarbeitet den aktuellen Knoten erneut mit dem nächsten passenden Template.

Die Semantik entspricht der in XSLT 2.0 eingeführten Anweisung `xsl:next-match` und lässt sich kurz so beschreiben: Es wird das Template ausgeführt, das ausgewählt worden wäre, wenn das aktuelle Template (und alle durch vorherige Aufrufe von `stx:process-self` virtuell entfernten Templates) nicht existieren würden.

`stx:process-document`
verarbeitet ein externes Dokument.

In XSLT stellt die Funktion `document` ein externes Dokument als Ganzes, d.h. in Form eines Baumes bereit. Sie kann daher kein Bestandteil von STX sein. Stattdessen soll ein solches Dokument ebenfalls seriell verarbeitet werden. Dies wird durch diese spezielle Anweisung ermöglicht.

In XSLT 2.0 wird eine neue Funktion `unparsed-text` verfügbar sein, die ein Dokument ungeparst in Form einer langen Zeichenkette zurückliefert. Auch hier wird das Dokument als Ganzes in den Speicher eingelesen. Die in STX angemessene Verarbeitung als Strom von Zeichendaten lässt sich ebenfalls über die Anweisung `stx:process-document` mit einem entsprechenden Attribut `input-method` und dem Wert `"text"` erreichen.

`stx:process-buffer`
verarbeitet den Inhalt eines STX-Puffers.

Der Puffermechanismus von STX wird im folgenden Kapitel 5.6.5 vorgestellt. Analog zu `stx:process-document` wird der Inhalt des Puffers seriell verarbeitet, d.h. er ersetzt den aktuellen Eingabestrom. Der Inhalt eines Puffers lässt sich nur über diese Anweisung verarbeiten.

Bis auf `stx:process-attributes` wird durch jede der hier genannten Anweisungen ein XML-Fragment verarbeitet. Diese Eigenschaft wird für die Zusammenarbeit mit anderen Transformationsprozessen (siehe Kapitel 5.6.7) wichtig sein.

5.6.4 Gruppen

Ein STX-Transformations-Sheet kann in eigenständige Module, *Gruppen* genannt, unterteilt werden. Eine Gruppe wirkt wie ein eigenständiges Transformations-Sheet. Der aktuelle Transformationsschritt wird nur durch die Templates der aktuellen Gruppe bestimmt. Darüber hinaus kann eine Anzahl von Grundeinstellungen auf Gruppenebene vorgenommen werden.

Eine Gruppe wird durch das Element `stx:group` gebildet. Gruppen können rekursiv ineinander verschachtelt werden. Das Wurzelelement `stx:transform` eines Transformations-Sheet bildet selbst die äußerste bzw. Standard-Gruppe. Eine Gruppe muss immer ein Unterelement einer anderen Gruppe sein und kann selbst weitere Untergruppen sowie die auf Gruppenebene erlaubten STX-Elemente enthalten, wie beispielsweise `stx:template`, `stx:procedure`, `stx:variable` oder `stx:recover`.

Die folgenden drei Grundeinstellungen sind auf Gruppenebene möglich: Gruppenoptionen

`pass-through`
Dieses Attribut ermöglicht die Angabe eines Standardverhaltens für unbehandelte Knoten. In XSLT wird die pragmatische (und im Präsentationskontext sinnvolle) Regel angewendet, dass Textknoten standardmäßig in die Ausgabe kopiert werden, dies bei allen anderen Knotentypen jedoch unterbleibt. In STX ist es möglich, zwischen den Werten "`all`" (alles kopieren), "`text`" (nur Text kopieren) oder "`none`" (nichts kopieren) zu wählen. Der Vorgabewert für `stx:transform` ist "`none`".

`recognize-cdata`
bestimmt, ob CDATA-Abschnitte als eigenständige Knoten während der Transformation behandelt werden sollen. Dies setzt voraus, dass Informationen über CDATA-Grenzen im STX-Datenmodell vorhanden sind. Da es sich hier um einen optionalen Bestandteil des Datenmodells handelt (siehe Kapitel 5.3.3), kann es vorkommen, dass in der zu transformierenden Dateninstanz diese Informationen fehlen. Der Wert dieses Attributes hat in diesem Fall keine Bedeutung. Der Vorgabewert für `stx:transform` ist "`yes`". Durch den Wert "`no`" wird ein XSLT-konformes Verhalten erreicht, sodass Text, der CDATA-Abschnitte enthält, als ein einzelner Textknoten repräsentiert wird.

`strip-space`
Das Attribut `strip-space` bestimmt, ob Leerraumknoten (d.h. Textknoten, die ausschließlich aus Leerraumzeichen bestehen) verarbeitet werden sollen. Ist dieses Attribut auf den Wert "`yes`" gesetzt, werden solche Textknoten ignoriert. Der Vorgabewert für `stx:transform` ist "`no`".

`text-by-lines`
: Falls dieses Attribut auf den Wert "`yes`" gesetzt ist, werden Textknoten der Eingabe an den Zeilenenden voneinander getrennt. Auf diese Weise lassen sich auch sehr lange Textblöcke durch STX seriell verarbeiten. Der Vorgabewert für `stx:transform` ist "`no`".

Der Vorgabewert dieser Attribute für `stx:group`-Elemente ist "`inherit`". In diesem Fall erbt eine Gruppe den Wert ihrer Elterngruppe.

Wechsel zwischen Gruppen

Nur die in der aktuellen Gruppe *sichtbaren* Templates werden für den nächsten Transformationsschritt, d.h. für die Verarbeitung des aktuellen Knotens, herangezogen. Zu den sichtbaren Templates gehören neben den direkt enthaltenen Templates meist noch einige andere, die dann einen Wechsel der aktuellen Gruppe bewirken. Daneben können Gruppen benannt werden, sodass sich zwei voneinander unabhängige Varianten für den Wechsel in eine andere Gruppe ergeben:

1. **direkt**: über den Namen der Gruppe
2. **indirekt**: durch über Gruppengrenzen hinweg sichtbare Templates

Bevor der Begriff der *Sichtbarkeit* im Detail erläutert wird, sei hier jedoch zunächst die erste Variante besprochen, da sie gewisse Ähnlichkeiten mit den aus XSLT bekannten Modi besitzt.

Benannte Gruppen

Jeder Gruppe mit Ausnahme der Standard-Gruppe kann mit Hilfe eines `name`-Attributs ein Name gegeben werden. Dieser Name muss im gesamten STX-Transformations-Sheet eindeutig sein. Der Wechsel in eine solche benannte Gruppe erfolgt durch die Angabe eines `group`-Attributs in einer `stx:process-`*`xxx`*-Anweisung. Die durch diese Anweisung adressierten Knoten werden durch die Templates der gewählten Gruppe verarbeitet. Es spielt an dieser Stelle keine Rolle, wie die einzelnen Gruppen ineinander verschachtelt worden sind.

Ersatz für `mode`

Die Benutzung benannter Gruppen entspricht der Wirkungsweise des `mode`-Attributes in XSLT. Alle XSLT-Templates, die zu einem Modus gehören, bilden implizit eine Gruppe. Wenn in der Anweisung `xsl:apply-templates` ein entsprechender Modus angegeben wird, kommen nur die Templates des gleichen Modus für den nächsten Verarbeitungsschritt in Frage.

In XSLT werden alle Templates auf globaler Ebene definiert und dürfen in einer beliebigen Reihenfolge im Stylesheet stehen. Die Gruppierung in STX bewirkt, dass zusammengehörende Templates (solche, die zum gleichen Modus gehören) auch im Transformations-Sheet nebeneinander als Unterelemente der gleichen Gruppe notiert werden müssen. Gruppen in STX wirken damit ebenfalls als strukturierendes Mittel beim Entwurf von STX-Transformations-Sheets.

Beispiel

Der Einsatz benannter Gruppen ist insbesondere dann hilfreich, wenn für die Verarbeitung der folgenden Knoten andere Gruppeneigenschaften gelten sollen. Ein sehr häufiger Anwendungsfall betrifft das Attribut `pass-through`. Angenommen, für das gesamte Transformations-Sheet gilt, dass unbehandelte Knoten nicht kopiert werden (`pass-through="none"`). Wenn nun bestimmte Unterbäume (XML-Fragmente) vollständig in die Ausgabe kopiert werden sollen, kann dies mit Hilfe einer leeren Gruppe und der Angabe `pass-through="all"` erreicht werden.

Listing 7 demonstriert einen solchen Fall, in dem nur die `info`-Elemente in das Ergebnis kopiert werden, bei denen der Text "`XML`" im `keyword`-Attribut auftritt.

Anwendung einer leeren "copy"-Gruppe **Listing 7**

```
<stx:template match="info">
  <stx:if test="contains(@keyword, 'XML')">
    <stx:process-self group="copy" />
  </stx:if>
  <stx:else>
    <stx:process-children />
  </stx:else>
</stx:template>

<!-- copy-Gruppe -->
<stx:group name="copy" pass-through="all" />
```

Anonyme Gruppen und Sichtbarkeiten von Templates

Im Gegensatz zu benannten Gruppen erfolgt der Wechsel in eine anonyme Gruppe ausschließlich automatisch über die Auswahl von Templates. Jedes Template gehört zu einer Sichtbarkeitsstufe, aus der sich ergibt, in welchen anderen Gruppen dieses Template ebenfalls sichtbar ist. Ein Gruppenwechsel erfolgt damit nicht gezielt durch die Angabe des Namens der Zielgruppe, sondern wird durch die zu transformierenden XML-Eingabedaten gesteuert.

Vorrangkategorien

In jeder Gruppe werden drei *Vorrangkategorien* bestimmt, die die jeweils dort sichtbaren Templates enthalten. Die Zugehörigkeit eines Template zu einer dieser Vorrangkategorien ergibt sich aus dessen Attributen `public` und `visibility`.

`public`

Mit Hilfe des Attributs `public` werden öffentliche (`public="yes"`) und private (`public="no"`) Templates unterschieden. Ohne Angabe dieses Attributs sind Templates der Standard-Gruppe öffentlich, alle Templates in Untergruppen jedoch privat. Öffentliche Templates besitzen die Eigenschaft, auch in ihrer Elterngruppe gleichberechtigt mit den anderen Templates sichtbar zu sein. Ein öffentliches Template fungiert damit als Einstiegs-Template in eine Gruppe. Die nahe liegende Analogie zu objektorientierten Sprachen, in der STX-Gruppen mit Klassen vergleichbar sind, und deren öffentliche Member den öffentlichen Templates der Gruppe entsprechen, trifft jedoch nur begrenzt zu.[11]

`visibility`

Das zweite Attribut `visibility` legt eine großräumigere Sichtbarkeit fest. Die möglichen Werte sind "`local`" (dies ist gleichzeitig der Vorgabewert), "`group`" und "`global`". Lokale Templates sind nur in ihrer eigenen Gruppe sichtbar, gegebenenfalls noch in ihrer Elterngruppe, wenn es sich um ein öffentliches Template handelt. Gruppen-Templates (`visibility="group"`) können aus allen Untergruppen heraus benutzt werden. Globale Templates sind schließlich im gesamten Transformations-Sheet sichtbar.

[11] Über die Angabe ihres Namens kann direkt in eine Gruppe gesprungen werden. Öffentliche und private Templates dieser Gruppe spielen dann keine Rolle. Aus objektorientierter Sicht würde man erwarten, dass von außen immer nur die öffentlichen Templates benutzt werden können, aber dieser Eindruck trügt.

Diese Sichtbarkeitsstufen formen die drei folgenden Vorrangkategorien, in denen nacheinander nach einem passenden Template für den jeweils aktuellen Knoten gesucht wird:

1. Die erste Kategorie enthält alle Templates der aktuellen Gruppe sowie zusätzlich die öffentlichen Templates aus allen direkten Untergruppen (den Kindgruppen).

   Auf diese Weise lassen sich Transformations-Sheets erstellen, deren Gruppen-Struktur die Struktur der Eingabedaten widerspiegelt. Templates für innere Elemente dienen als Einstiegs-Template in eine neue Gruppe. Innerhalb dieser Gruppe müssen vom STX-Prozessor dann weniger Templates mit dem aktuellen Knoten verglichen werden. Diese Vorgehensweise eignet sich vorzugsweise für streng strukturierte Dokumenttypen ohne rekursive Elemente.

2. Die zweite Kategorie enthält alle Templates aus Gruppen, die Vorfahren der aktuellen Gruppe sind, und die die Werte "`group`" oder "`global`" für das Attribut `visibility` besitzen.

   Solche Templates gelten damit für ihre eigene Gruppe und für alle rekursiv enthaltenen Untergruppen. Sie ermöglichen das »Zurückspringen« aus einer inneren Gruppe.

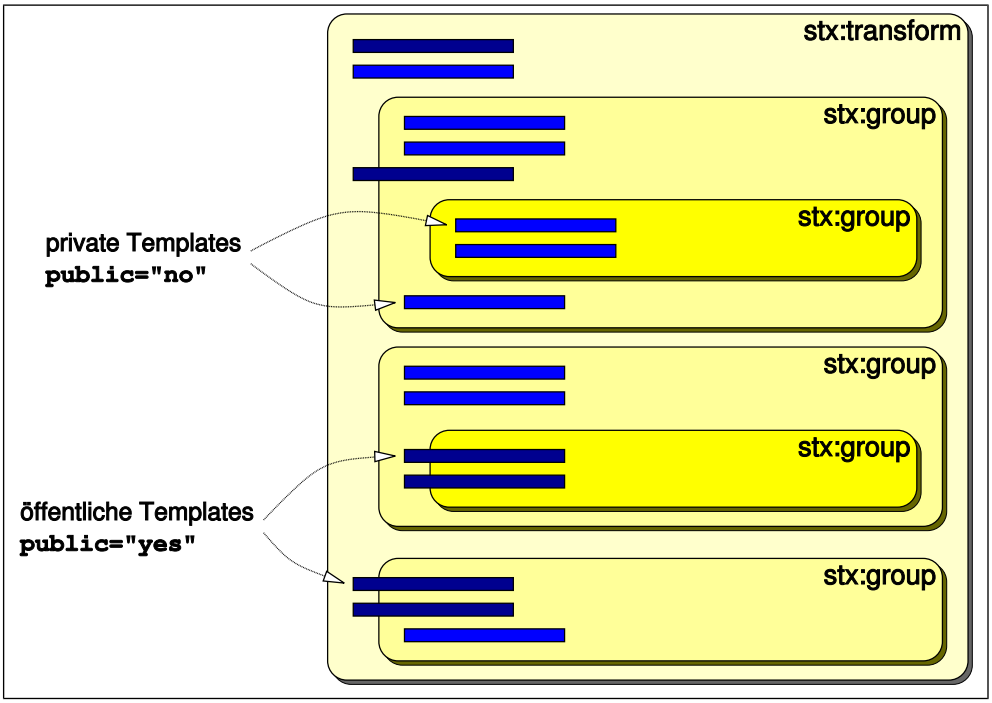
3. Die dritte Kategorie enthält alle globalen Templates im Transformations-Sheet.

   Diese Templates sind in jeder Gruppe sichtbar. Ein Template, das zum Beispiel als *catch-all*-Template alle unerwarteten Elemente behandelt und mit einer Fehlerausgabe (`stx:message`) darauf reagiert, muss typischerweise global sichtbar sein.

Nur wenn in einer Kategorie kein passendes Template gefunden wird, wird die Suche in der nächsten Kategorie fortgesetzt. Die Festlegung von Prioritäten für Templates durch das Attribut `priority` dient ausschließlich der Gewichtung innerhalb der gleichen Kategorie.

Abbildung 5

Gruppen und Template-Sichtbarkeiten



In Abbildung 5 werden diese drei Kategorien schematisch dargestellt. Jedes gelbe Rechteck symbolisiert eine Gruppe, jedes blaue Rechteck symbolisiert ein Template.

Die genannten Regeln über Vorrangkategorien gelten für alle Gruppen, nicht nur für die anonymen. Bei der expliziten Angabe einer Zielgruppe muss man sich deshalb darüber bewusst sein, dass diese Zielgruppe anschließend als neue aktuelle Gruppe (Basisgruppe) für die Suche nach einem passenden Template dient. Damit stehen nicht nur die direkt enthaltenen Templates der Gruppe zur Verfügung, sondern ebenfalls alle in den drei Vorrangkategorien der Zielgruppe enthaltenen Templates.

Prozeduren

Außer für Templates gelten die drei vorgestellten Vorrangkategorien ebenfalls für Prozeduren (`stx:procedure`). So ist es möglich, Prozeduren mit gleichem Namen innerhalb desselben Transformations-Sheet zu definieren, solange in jeder Vorrangkategorie alle Prozedurnamen eindeutig sind. Während bei der Erstellung eines einzelnen Transformations-Sheet aus Gründen der Wartbarkeit alle Prozeduren mit einem eindeutigen Namen versehen werden sollten, können beim Einschließen anderer Transformations-Sheets per `stx:include` durchaus Prozeduren mit gleichem Namen auftreten. Mit Hilfe der Vorrangkategorien werden mögliche Konflikte vermieden. Ein Suchprozess wie bei Templates findet hier zur Laufzeit jedoch nicht statt. Die durch eine `stx:call-procedure`-Anweisung aufgerufene Prozedur kann statisch, d.h. zur Compile-Zeit, bestimmt werden.

Einschließen anderer STX-Dateien

Die Verwendung von Templates anderer Transformations-Sheets geschieht mit Hilfe der Anweisung `stx:include`. Diese darf auf Gruppenebene, d.h. als Kind von `stx:group` frei im Transformations-Sheet auftreten. Zur Compile-Zeit wird dann die jeweilige `stx:include`-Anweisung durch eine neue Gruppe ersetzt. Das Element `stx:transform` des einzuschließenden Transformations-Sheet wird zu einem `stx:group`-Element des einschließenden Transformations-Sheet. Die Attribute `pass-through`, `strip-space` und `recognize-cdata` des eingeschlossenen `stx:transform`-Elements übertragen sich auf die neue Gruppe und bleiben so für die darin beschriebene Transformation erhalten.

Diese Vorgehensweise stellt somit sicher, dass die Semantik des eingeschlossenen Transformations-Sheet weitgehend erhalten bleibt. Gruppeneigenschaften der einschließenden Gruppe haben keine Auswirkung auf die neue Gruppe. Ebenso bleibt durch den Einschluss die erste Vorrangkategorie der neuen Gruppe unverändert. Eine Transformation, die niemals Templates der zweiten oder dritten Vorrangkategorie benutzt, wird auch nicht durch Templates des einschließenden Transformations-Sheet beeinflusst.

Da alle Templates der Standard-Gruppe per Voreinstellung öffentlich sind, werden diese der ersten Vorrangkategorie der einschließenden Gruppe hinzugefügt. Dies erleichtert die Verwendung der Anweisung `stx:include` so wie aus XSLT bekannt.

Gruppen-Variablen

Das Konzept der Gruppen wirkt sich auch auf die Sichtbarkeit von Variablen aus. Aus XSLT sind bereits globale Variablen bekannt. Diese werden außerhalb von Templates als so genannte Top-Level-Elemente deklariert. In STX entsprechen die Variablen der Standard-Gruppe diesen globalen Variablen. Daneben ist es nun ebenfalls möglich, Variablen in Untergruppen zu deklarieren. Diese sind dann nur innerhalb ihrer Gruppe (inklusive aller weiteren Untergruppen) sichtbar. Solche

Gruppenvariablen verhalten sich wie statische Variablen, d.h. es existiert genau eine Instanz, die vor Beginn der Transformation initialisiert wird.

5.6.5 Temporäre XML-Fragmente

STX wurde als Transformationssprache entworfen, die ein XML-Dokument in einem einzigen Durchlauf transformiert, d.h. eine *one-pass*-Transformation durchführt. Der Inhalt des XML-Dokuments wird dabei einmal sequentiell an den STX-Prozessor gemeldet, der wiederum die erhaltenen Daten unmittelbar in ein XML-Ergebnis umwandelt.

Puffer

Es ist leicht einzusehen, dass ohne zusätzliche Speichermechanismen eine Vielzahl von Transformationen nicht ausgeführt werden könnten. Beispielhaft sei hier das Generieren eines Abbildungsverzeichnisses am Ende eines zu formatierenden Artikels genannt. Mit den zur Verfügung stehenden Sequenzen können einfache Werte oder auch einzelne Knoten, jedoch keine XML-Strukturen repräsentiert werden. Es wird damit ein Konzept benötigt, mit dem sich XML-Fragmente abspeichern lassen. STX stellt für diesen Anwendungsfall so genannte *Puffer* zur Verfügung. Solche Puffer können als besondere Variablen angesehen werden, die jedoch nur mit speziellen Anweisungen verarbeitet werden können. Insbesondere gibt es in STXPath kein Konstrukt für den Zugriff auf Puffer.

Die einzige Möglichkeit, den Inhalt eines Puffers zu verarbeiten, besteht darin, ihn als Quelle der Transformation zu verwenden. Der Inhalt des Puffers wird in diesem Fall ebenso sequentiell verarbeitet wie der normale XML-Eingabestrom. Ein direkter Zugriff auf die Knoten des gespeicherten Fragments ist nicht möglich, da dies einer vollen XPath-Unterstützung für den Inhalt von STX-Puffern entspräche. Da STX eine Schnittstelle bietet, XML-Fragmente durch externe Applikationen verarbeiten zu lassen (siehe Kapitel 5.6.7), ist eine weitergehende direkte XPath-Unterstützung an dieser Stelle nicht notwendig.

Die für die Verarbeitung von Puffern in STX bereitgestellten Anweisungen folgen den bekannten Mustern ähnlicher Anweisungen:

`stx:buffer`
: deklariert einen Puffer.

  Dies entspricht dem Muster der Variablendeklaration. Der Inhalt dieser Anweisung wird zur initialen Belegung des Puffers verwendet.

`stx:result-buffer`
: aktiviert einen Puffer als Ziel des folgenden Transformationsschrittes.

  Der Inhalt dieses Elementes füllt den Puffer. Dies entspricht dem Muster der `stx:result-document`-Anweisung, die auch in XSLT 2.0 in ähnlicher Form enthalten sein wird.

`stx:process-buffer`
: verarbeitet den Inhalt eines Puffers.

  Der aktuelle Eingabestrom wird durch diese Anweisung unterbrochen, der Inhalt des Puffers eingeschoben und der Eingabestrom anschließend fortgesetzt. Diese Anweisung folgt dem Muster der anderen `stx:process-`*`xxx`*-Anweisungen, die im Kapitel 5.6.3 vorgestellt wurden.

Mehrfache Verarbeitung von Puffern

Mit Hilfe von Puffern lässt sich der strenge *one-pass*-Charakter von STX abmildern. So kann der Inhalt eines Puffers mehrfach verarbeitet werden, u.a. durch verschiedene Templates in verschiedenen Gruppen. Darüber hinaus kann derselbe Puffer gleichzeitig als Quelle und als Ziel eines Transformationsschrittes dienen. Auf diese Weise lässt sich der Inhalt eines Puffers in mehreren Iterationsschritten ändern. Der Grundaufbau einer solchen Iteration ist in Listing 8 skizziert.

Iteration über dem Inhalt eines STX-Puffers **Listing 8**

```
1  <stx:buffer name="store">
2    <!-- Anfangsbelegung des Puffers -->
3    ...
4  </stx:buffer>
5
6  <stx:while test="Bedingung">
7    <stx:result-buffer name="store" clear="yes">
8      <stx:process-buffer name="store" />
9    </stx:result-buffer>
10 </stx:while>
```

Das `clear`-Attribut in Zeile 7 mit dem Wert "`yes`" sorgt dafür, dass der vorherige Inhalt des Puffers vor dem Füllen gelöscht wird. Der neue Inhalt wird erst am Ende der `stx:result-buffer`-Anweisung tatsächlich in den Puffer geschrieben. Auf diese Weise wird die dargestellte Vorgehensweise erst ermöglicht, da durch die `stx:process-buffer`-Anweisung in Zeile 8 der vorherige Inhalt verarbeitet wird, obwohl dieser in Zeile 7 scheinbar schon überschrieben wurde.

Beispiel: Sortieren

Mit Hilfe eines Puffers kann auch sortiert werden.[12] Beispielsweise lässt sich ein einfacher Bubble-Sort-Algorithmus in STX implementieren, indem die zu sortierenden Elemente zunächst in einen Puffer kopiert werden. Während der Verarbeitung des Puffers werden benachbarte Elemente miteinander verglichen und gegebenenfalls miteinander vertauscht. Diese Iteration läuft solange, bis keine Änderungen mehr vorgenommen werden müssen.

Dieses Beispiel verdeutlicht zugleich, dass mit Hilfe von Puffern zwar komplexere Transformationen möglich sind, diese aufgrund des sequentiellen Charakters allerdings sehr ineffizient ablaufen. Darüber hinaus lässt sich durch die extensive Verwendung von Puffern der Ansatz von STX als speicherfreundliche XML-Transformationssprache ohne weiteres aushebeln.[13]

5.6.6 Verarbeitung von Zeichendaten

XSLT in der Version 1.0 konzentriert sich auf die Transformation von XML-Daten auf der Ebene von Knoten. Die durch das XPath-Datenmodell definierten Knoten des XML-Baumes bilden die atomaren Einheiten, die durch XSLT-Templates verarbeitet werden können. Für die Verarbeitung von Zeichendaten, z.B. des Inhalts eines Text-

[12] Das Sortieren von n Elementen benötigt mindestes $O(n \log n)$ Berechnungsschritte. In einem einmaligen Durchlauf der n Elemente ist kein Sortieren möglich.

[13] Allerdings könnten Puffer durchaus über externe Warteschlangen (FIFOs) implementiert werden, da auf sie ausschließlich sequentiell zugegriffen wird. Ein STX-Prozessor muss den Inhalt eines Puffers in diesem Fall nicht im Hauptspeicher verwalten. Er würde jedoch an Performance verlieren.

knotens, stehen allein XPath-Zeichenkettenfunktionen zur Verfügung. Der funktionale Charakter von XSLT bedingt jedoch, dass die Ersetzung einzelner Textabschnitte durch XML-Markup nur aufwändig über rekursive Templates erreicht werden kann.

Beispiele für solche Ersetzungen sind

- das Ersetzen aller Newline-Zeichen durch das leere Element `<br/>`,
- die Überführung von CSV[14]-Daten in eine Folge von Elementen, die die einzelnen Werte enthalten,
- die Ersetzung spezieller Textformate (z.B. ein Datum als 2004-03-12) durch explizites Markup.

Die Umwandlung von XML-Dokumenten mit implizit strukturiertem Textinhalt in solche mit explizitem Markup gehört zu den typischen Transformationsaufgaben. Trotzdem bietet XSLT 1.0 hier kaum Unterstützung.

Vorbild: *lex*

Das in STX für diesen Zweck vorgesehene Sprachmittel orientiert sich am Einsatz regulärer Ausdrücke, wie er aus dem Scanner-Generator *lex* [LS75] bekannt ist. In *lex* werden zeilenweise reguläre Ausdrücke als Muster für zu suchende Zeichenketten notiert. Dazu wird jeweils eine Aktion angegeben, die dann ausgeführt wird, wenn eine passende Zeichenkette gefunden wurde. Das folgende Beispiel zeigt ein solches *lex*-Programm, das einige Ersetzungen vornimmt.

```
%%
MegaTool      { printf("<b>MegaTool</b>"); }
DM            { printf("Euro"); }
[a-zA-Z-]+    { ECHO; }
```

Die auf der linken Seite stehenden regulären Ausdrücke werden hier nach folgender Regel ausgewählt:

1. Es wird der Ausdruck gesucht, der auf das erste Zeichen der Eingabe passt.
2. Von allen Ausdrücken, die an der gleichen Position beginnen, wird derjenige gewählt, der die längste Teilzeichenkette abdeckt.
3. Sollte es dabei mehrere Möglichkeiten geben, wird derjenige gewählt, der im Programm am weitesten vorn steht.

Dieser hier beschriebene Algorithmus lässt sich direkt auf STX übertragen.

`stx:analyze-text`

Das Element `stx:analyze-text` bestimmt die Zeichenkette, die mit Hilfe von regulären Ausdrücken untersucht werden soll. Dessen Kindelemente `stx:match` enthalten jeweils einen regulären Ausdruck in ihrem `regex`-Attribut und in ihrem Inhalt die einzusetzende XML-Struktur. Ein optionales Element `stx:no-match` kann angegeben werden für Teilzeichenketten, die auf keinen der spezifizierten regulären Ausdrücke passen. Die gesamte Struktur ähnelt damit dem Aufbau von `stx:choose` mit seinen `stx:when`- und `stx:otherwise`-Zweigen.

Die Auswertung der einzelnen `stx:match`-Zweige erfolgt jedoch nicht streng nacheinander bis ein passender Ausdruck gefunden wurde, sondern gemäß den oben formulierten Regeln. Es wird der am besten passende reguläre Ausdruck gesucht und dann der im dazugehörigen `stx:match`-Element stehende Inhalt abgearbeitet. Ist ein `stx:no-match`-Element vorhanden, wird dieses für die längste Folge aufein-

[14] CSV = Comma Separated Values

ander folgender Zeichen aufgerufen, die auf keinen der angegebenen regulären Ausdrücke passen. Mit anderen Worten: es wird niemals ein `stx:no-match` direkt nach einem vorangehenden `stx:no-match` ausgeführt. Genauso wie es in XPath keine benachbarten Textknoten gibt, gibt es hier keine aufeinander folgenden Zeichenketten, die auf keines der `stx:match` passen.

Template-Analogie

Die hier beschriebene Vorgehensweise für die Verarbeitung von Zeichendaten entspricht darüber hinaus dem bekannten Template-Modell von XSLT und STX. Jedes `stx:match`-Element fungiert als ein Template für Zeichendaten, das im Attribut `regex` ein Muster für die zu verarbeitenden Zeichen enthält. Anstelle eines Stroms von XML-Daten verarbeitet `stx:analyze-text` eine Folge von Zeichendaten. Während ein Template jedoch immer für genau einen XML-Knoten aufgerufen wird, kann ein `stx:match` für beliebig viele zusammenhängende Zeichen ausgeführt werden. Analog wird für nichtbehandelte XML-Knoten immer einzeln die festgelegte Standardregel ausgeführt, während nichtbehandelte Zeichendaten als Zeichenkette betrachtet werden. Damit gibt es für das Element `stx:no-match` auch keine Entsprechung auf Template-Ebene.

Das folgende in Listing 9 abgedruckte Beispiel ersetzt in einem Text alle Leerzeichen durch das Zeichen *NO-BREAK SPACE* (Code `0xA0`) und alle Zeilenwechsel durch das leere Element `br`.

Textersetzung mit stx:analyze-text **Listing 9**

```
1  <stx:template match="pre">
2    <stx:analyze-text select=".">
3      <stx:match regex="\n">
4        <br />
5      </stx:match>
6      <stx:match regex="' '"> </stx:match>
7      <stx:no-match>
8        <stx:value-of select="regex-group(0)" />
9      </stx:no-match>
10   </stx:analyze-text>
11 </stx:template>
```

Die in Zeile 8 aufgerufene Funktion `regex-group` liefert im Normalfall die Teilzeichenkette, die durch den zum übergebenen Argument gehörenden geklammerten Bereich im regulären Ausdruck ausgewählt wird. Das Argument `0` adressiert die gesamte Teilzeichenkette, im Falle des Elements `stx:no-match` also die gesamte durch keinen regulären Ausdruck abgedeckte Zeichenkette. Da in STX der Punkt in einem Ausdruck »`.`« immer den aktuellen Knoten bezeichnet, kann dieser nicht zum Zugriff auf Teilzeichenketten in `stx:analyze-text` herangezogen werden. Mit der Funktion `regex-group` steht aber ein vollwertiger Ersatz zur Verfügung.

Vergleich: XSLT 2.0

Ab der Version 2.0 von XSLT wird es auch hier eine Anweisung für die Verarbeitung von Text mit Hilfe regulärer Ausdrücke geben. Diese wird `xsl:analyze-string` heißen und besitzt die beiden Unterelemente `xsl:matching-substring` und `xsl:non-matching-substring`. Die oben genannte Funktion `regex-group` existiert ebenfalls in XSLT 2.0 und wurde in STX übernommen.

Im Gegensatz zu STX kann in XSLT 2.0 jedoch jeweils nur ein regulärer Ausdruck pro `xsl:analyze-string` angegeben werden. Mehrfache Ersetzungen innerhalb des gleichen Textes sind daher nur mittels geschachtelter Konstruktionen möglich.

Die in STX enthaltene Variante ermöglicht es, mehrfache Ersetzungen einfacher zu programmieren und deckt darüber hinaus die XSLT-Funktionalität vollständig mit ab. Der Vorschlag an das W3C, die mächtigere STX-Version ebenfalls in XSLT umzusetzen, wurde für die Version 2.0 mit dem Argument abgelehnt, dass die Festlegung des neuen Funktionsumfang bereits abgeschlossen sei.[15]

5.6.7 Zusammenarbeit mit externen Filterprozessen

Der serielle Charakter von STX bedingt, dass diese Sprache nur für bestimmte Arten von Transformationen gut geeignet ist. So lassen sich komplexere strukturelle Änderungen der XML-Daten nur relativ aufwändig realisieren. Transformationen, die den freien Zugriff auf die zu transformierenden Daten benötigen, sollten in XSLT gelöst werden.

Hybride Transformationen

Nun lassen sich jedoch nicht alle Transformationsaufgaben klar in eine der beiden Kategorien »baumorientiert« (XSLT) und »streamorientiert« (STX) einordnen. In den Grenzbereich fallen Transformationen,

- die abschnittsweise die Gesamtsicht auf ein Fragment der XML-Daten benötigen und
- die einzelne Abschnitte bzw. Fragmente weitgehend unabhängig voneinander verarbeiten können.

Bei großen Datenmengen sind Transformationen dieses Typs auf der einen Seite zu speicherintensiv für die Verarbeitung mit XSLT, auf der anderen Seite jedoch zu komplex, um sie mit STX überschaubar und effizient zu realisieren.

Eine augenscheinliche Lösung dieses Dilemmas wäre eine Erweiterung von STX, mit der die vollständige XPath-Funktionalität auf in Puffern gespeicherten Fragmenten (siehe Kapitel 5.6.5) genutzt werden kann. Dies würde bedeuten, dass in STX zwei Typen von Pfadsprachen (STXPath und W3C-XPath) definiert werden müssten. Die Transformationssprache STX und damit die Implementierungen würden komplexer und aufwändiger.

Anstatt von einem STX-Prozessor zu verlangen »das Rad neu zu implementieren«, bietet es sich an, vorhandene Software zu benutzen; in diesem Fall existierende XSLT-Implementierungen.

Externe Filter

Dieser Ansatz kann in zweierlei Hinsicht verallgemeinert werden:

1. Ein XML-Fragment kann an eine beliebige XML-Software übergeben werden, die mit diesen Daten eine Transformation ausführt.
2. Ein XML-Fragment kann als XML-Strom direkt an eine solche Software geleitet werden, ohne dass es vorher in einem Puffer zwischengespeichert werden muss.

Die benutzte XML-Software muss als Ergebnis wieder ein XML-Fragment produzieren, das an die aufrufende STX-Komponente zurückgeliefert wird. Insofern muss es sich immer um eine XML-Transformation handeln, wenngleich nicht notwendigerweise um XSLT. Die externe Komponente fungiert hier als Filter, die XML-Daten konsumiert und XML-Daten produziert.

[15] siehe http://lists.w3.org/Archives/Public/public-qt-comments/2004Apr/0020.html

Teile und Herrsche

Genau genommen ist die Idee, ein umfangreiches Dokument in kleinere Teile zu zerlegen und diese einzeln zu verarbeiten, nicht neu. Es handelt sich um das bekannte Prinzip *divide et impera*, das hier auf die Eingabedaten angewandt wird. Im Zusammenhang mit XML muss jedoch beachtet werden, dass die einzelnen Teile wohlgeformte XML-Fragmente darstellen müssen. Die Identifizierung der Fragmente stellt hier die eigentliche Herausforderung dar, wenn man versucht, diese Aufgabe mit einfachen Textfilterwerkzeugen oder APIs wie SAX zu bewältigen.

In STX ist die Lösung dieser Aufgabe jedoch kein Problem. Der Matching-Mechanismus, der über Muster gezielt Knoten im Eingabestrom identifiziert, ist Bestandteil der Sprache. Dasjenige Template, das auf diesem Weg zur Bearbeitung eines Knotens ausgewählt wurde, kann über die folgenden Anweisungen festlegen, welche Knoten der Eingabedaten im Inneren des Template verarbeitet werden sollen:

- die Kindknoten (`stx:process-children`)
- der Knoten selbst (`stx:process-self`)
- die nachfolgenden Geschwister (`stx:process-siblings`)
- die Attributknoten (`stx:process-attributes`)

Darüber hinaus können externe Dokumente (`stx:process-document`) oder der Inhalt eines Puffers (`stx:process-buffer`) verarbeitet werden.

Adressierung von XML-Fragmenten

Jede der angegebenen Anweisungen bestimmt eine Knotenmenge zur weiteren Verarbeitung durch den STX-Prozessor. Dabei ist leicht zu erkennen, dass die Kindknoten, die Geschwisterknoten und erst recht der aktuelle Knoten selbst XML-Fragmente darstellen. Ein externes Dokument und der Inhalt eines Puffers sind per Definition ebenfalls XML-Fragmente. Allein die Attribute fallen aus dieser Liste heraus. Ein Attribut gehört immer zu einem Element; eine Serialisierung ohne das zugehörige Element ist nicht möglich. Attribute allein bilden kein XML-Fragment.

Mit Ausnahme von `stx:process-attributes` adressiert damit jede der oben genannten Anweisungen ein XML-Fragment. Diese Anweisungen können nun dahingehend erweitert werden, dass sie das adressierte Fragment durch einen externen XML-Filter verarbeiten lassen.

Filteridentifikation

Eine solche Erweiterung benötigt zwei Informationen:

1. Welche Art von Transformation bzw. Filterung soll auf dem Fragment ausgeführt werden?

   Es könnte sich beispielsweise um eine XSLT-Transformation oder um das Anbringen einer digitalen Signatur handeln.

2. Welche Anweisungen sollen für diese Transformation benutzt werden?

   Beispielsweise benötigt eine XSLT-Transformation ein XSLT-Stylesheet als »Programm«.

Idealerweise sollte die Angabe der zu verwendenden externen Filterkomponente unabhängig von einer konkreten Implementierung erfolgen. Nur so lässt sich gewährleisten, dass der STX-Code portabel auf verschiedenen STX-Prozessoren ausgeführt werden kann. Die Angabe eines konkreten Programms, einer Bibliothek oder Klasse würde deren Verfügbarkeit in der aktuellen Umgebung erfordern. Die bessere Lösung besteht in der Angabe eines symbolischen Namens, der durch den STX-Prozessor auf die aktuell verfügbare Filter-Implementierung abgebildet wird.

Auch hierfür gibt es Analogien:

- In einem XML-Dokument können der Dokumenttyp und externe Entities sowohl über einen *System Identifier* (einen URI) als auch über einen so genannten *Public Identifier* beschrieben werden. Während der System Identifier auf eine konkrete Datei verweist, die dann bei der Verarbeitung erreichbar sein muss, ist ein Public Identifier ein abstrakter, prinzipiell weltweit eindeutiger Bezeichner, der einer festgelegten Syntax genügt. Beispielsweise bezeichnet "`-//W3C//DTD SVG 1.0//EN`" den Dokumenttyp für SVG 1.0. Die Abbildung auf eine reale (meist lokale) Ressource geschieht in der Regel über Kataloge. Auf diese Weise kann ein Public Identifier als einheitlicher Bezeichner plattformübergreifend verwendet werden. Auf unterschiedlichen Systemen verweist er auf die jeweils vorhandene lokale Repräsentation der Ressource.
- XSLT erlaubt die Bereitstellung zusätzlicher, implementationsspezifischer Funktionen, so genannter Erweiterungsfunktionen. Insbesondere ermöglichen Java-basierte XSLT-Prozessoren die Benutzung von Methoden aus Java-Klassen. Die jeweilige Klasse wird über den Namensraum der Funktion identifiziert. Allerdings gibt es keinen Standard für Abbildungen von Funktionsaufrufen in XSLT auf Methoden in Java, insbesondere auf überladene Methoden. Im Ergebnis können Stylesheets mit solchen Erweiterungsfunktionen nur durch Java-basierte Prozessoren ausgeführt werden, und selbst dort kann es zu unterschiedlichen Ergebnissen kommen.

  Zur Lösung dieses Problems hat die exslt-Initiative [EXSLT] eine Reihe häufig benötigter Erweiterungsfunktionen prozessorunabhängig beschrieben. Neben der genauen Syntax und Semantik wird insbesondere der Namensraum dieser Funktionen einheitlich festgelegt. Stylesheets, die exslt-Funktionen benutzen, können prozessorübergreifend ausgeführt werden. Die jeweilige Implementierung erfolgt prozessorspezifisch und ist nicht an eine konkrete Klasse o.ä. gebunden.

Namen für Filtermethoden

Welche konkrete Form eines symbolischen Namens nun für externe Filterprozesse in STX gewählt wird, ist Geschmackssache. Es gibt keine prinzipiellen Vor- oder Nachteile bei der Entscheidung zwischen einem Public Identifier und einem Namensraum-URI. Da der Einsatz der Public Identifiers bisher jedoch auf die Lokalisierung externer Entities oder DTDs beschränkt war, wird hier dem modernen Konzept der URIs der Vorrang gegeben. Eine vergleichbare Aufgabe kommt URIs in XLink zur Kennzeichnung der Rolle eines Link-Elements in den Attributen `xlink:role` und `xlink:arcrole` zu [W3C01d].

Für einige durch einen externen Filter anzuwendende Transformationsmethoden existieren bereits kanonische URIs:

http://www.w3.org/1999/XSL/Transform
ist der Namensraum für die Transformationssprache XSLT

http://stx.sourceforge.net/2002/ns
ist der Namensraum für die Transformationssprache STX

http://xml.org/sax
ist der Basis-URI für Eigenschaften in SAX

http://www.w3.org/2000/09/xmldsig#
ist der Namensraum für XML Signature

Jede STX-Implementierung kann zusätzliche URIs festlegen, die spezielle Filtermethoden beschreiben. Über die Methode `filter-available` kann zur Laufzeit die Verfügbarkeit einer gewünschten Methode überprüft werden. Eine Liste in der STX-Spezifikation standardisierter URIs stellt sicher, dass diese Filtermethoden prozessorunabhängig benutzt werden können.

Der URI der Filtermethode wird in dem Attribut `filter-method` angegeben. Gegebenenfalls notwendige Filteranweisungen werden über das Attribut `filter-src` spezifiziert. Soll das aktuelle Element und dessen Inhalt durch das XSLT-Stylesheet *foo.xsl* transformiert werden, kann dazu die folgende Anweisung benutzt werden:

```
<stx:process-self
        filter-method="http://www.w3.org/1999/XSL/Transform"
        filter-src="url('foo.xsl')" />
```

Die Angabe eines URL in `filter-src` muss dabei der durch XSLFO [W3C01e] definierten Syntax genügen.

Puffer als Filterquelle

Ein Spezialfall liegt vor, wenn die für die jeweilige Transformation notwendigen Anweisungen selbst wieder in XML notiert werden, wie es in XSLT und STX der Fall ist. In diesem Fall kann der Inhalt eines STX-Puffers als Quelle der Filteranweisungen genutzt werden. Für das Attribut `filter-src` muss hier der Name des Puffers innerhalb der Spezifikation `buffer(...)` angegeben werden. Diese Technik eröffnet zwei interessante Anwendungen:

- Es lassen sich in einem einzigen Dokument kombinierte Transformationen beschreiben. Diese können z.B. aus einem Teil STX und einem Teil XSLT bestehen. Das XSLT-Stylesheet ist in diesem Fall in einem STX-Puffer eingebettet und damit Teil der STX-Datei.
- Der Inhalt eines Puffers kann dynamisch erzeugt oder modifiziert werden. Auf diese Weise lassen sich Transformationen beschreiben, die ihren Code dynamisch generieren.

Der zweite Anwendungsfall soll an zwei Beispielen demonstriert werden.

Schematron

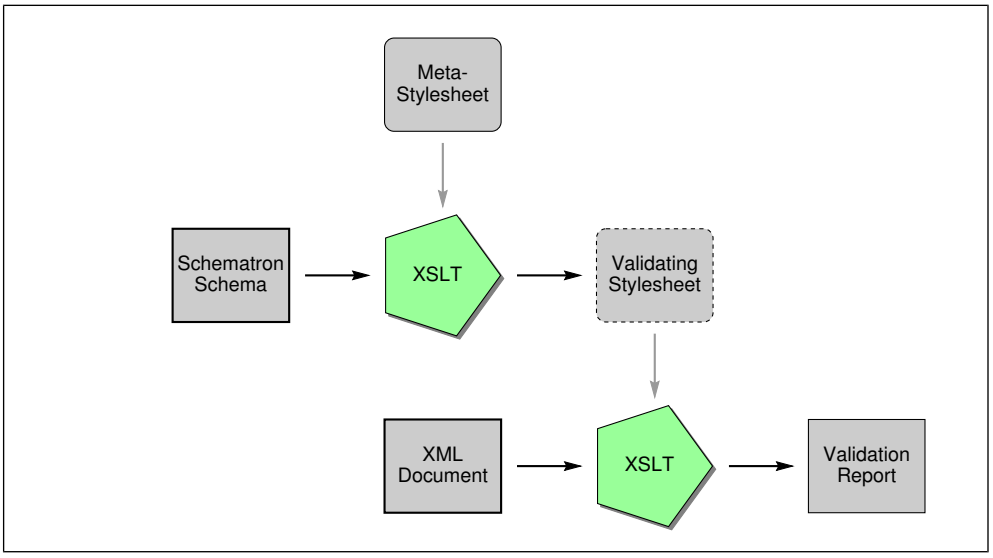
Schematron [Schtrn] ist eine Schemasprache, die Aussagen über Zusicherungen (*Assertions*) enthält und es erlaubt, ein konkretes XML-Dokument hinsichtlich der Einhaltung dieser Zusicherungen zu überprüfen. Damit unterscheidet sich Schematron von grammatikorientierten Schemasprachen.

Schematron verwendet XPath zur Formulierung dieser Bedingungen. Die kanonische Implementierung eines Schematron-Prozessors besteht in der Transformation der Schematron-Instanz in ein XSLT-Stylesheet, welches dann ein zu testendes XML-Eingabedokument in eine Menge von Fehlermeldungen »transformiert«, siehe Abbildung 6. Das Format dieser Fehlermeldungen variiert von Implementierung zu Implementierung. Es ist insbesondere unabhängig von der Schemasprache selbst.

Das Besondere an dieser Art der Implementierung besteht darin, dass keine zusätzliche spezifische Software benötigt wird. Die Implementierung besteht allein in einem so genannten Meta-Stylesheet, welches eine Schematron-Instanz nach XSLT transformiert. Als Software genügt daher jeder beliebige XSLT-Prozessor.

Abbildung 6

Ablauf einer Schematron-Validierung



Diese Art der Schematron-Anwendung mag auf den ersten Blick ungewöhnlich erscheinen, da die Trennung von Programm und Daten durch die gemeinsam benutzte XML-Syntax verschwindet. Ein XSLT-Prozessor generiert in einem ersten Schritt ein XSLT-Stylesheet, welches dann in einem zweiten Schritt das Eingabedokument transformiert (und damit validiert). Mit Hilfe der externen Filterschnittstelle können beide Schritte direkt durch STX ausgeführt werden, siehe Listing 10.

Listing 10 *Schematron und STX*

```
1  <stx:transform xmlns:stx="http://stx.sourceforge.net/2002/ns" version="1.0">
2
3    <stx:param name="schema" required="yes" />
4
5    <stx:buffer name="schematron">
6      <stx:process-document filter-src="url('schematron-basic.xsl')"
7                            filter-method="http://www.w3.org/1999/XSL/Transform"
8                            href="$schema" />
9    </stx:buffer>
10
11   <stx:template match="/">
12     <stx:process-self filter-src="buffer(schematron)"
13                       filter-method="http://www.w3.org/1999/XSL/Transform" />
14   </stx:template>
15
16 </stx:transform>
```

Zunächst wird der Puffer namens `schematron` mit dem dynamisch generierten Validierungs-Stylesheet gefüllt. Dies geschieht, indem die Schematron-Instanz in `$schema` durch das zu Schematron gehörende Meta-Stylesheet mit der Anweisung `stx:process-document` transformiert wird (Zeile 6). Anschließend wird der Inhalt dieses Puffers benutzt, um die zu transformierenden Eingabedaten per `stx:process-self` zu validieren (Zeile 12). Der implementierte Ablauf ist in Abbildung 7 dargestellt.

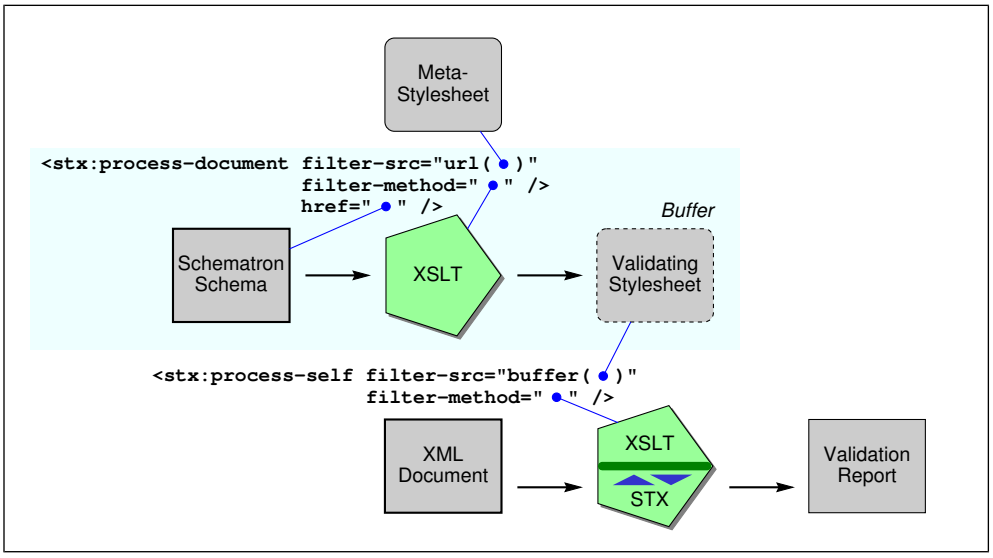


Abbildung 7

Ablauf der Schematron-Validierung mit STX

Dieses Beispiel ist etwas untypisch für die Verwendung externer Filterprozesse, da STX hier allein als Umgebung für den Aufruf zweier XSLT-Transformationen dient. Es findet keinerlei Transformation in STX selbst statt. Trotzdem demonstriert dieses Beispiel, wie mehrere aufeinander aufbauende Transformationsschritte durch eine STX-Transformation gekapselt werden können. Aus Anwendersicht vereinfacht sich damit die Schematron-Benutzung erheblich. Bei der Verwendung des STX-Prozessors *Joost* wäre beispielsweise die folgende Kommandozeile einzugeben:

```
joost file.xml schematron.stx schema=test.sch
```

Die Datei *file.xml* enthält dabei das zu validierende XML-Dokument, *schematron.stx* das in Listing 10 aufgeführte STX-Transformations-Sheet und *test.sch* die Schematron-Regeln.

Dynamische Pfadangaben

Transformationen, die Teile ihres Codes dynamisch generieren, müssen die dazu notwendigen Informationen aus den Eingabedaten selbst ableiten. Da STX nur sequentiell auf die Daten zugreifen kann, müssen die Informationen zur Code-Generierung (im folgenden *Meta-Code* genannt) zu Beginn der Eingabedaten angegeben werden. Mit Hilfe des in STX-Anweisungen übersetzten Meta-Codes kann dann der Rest der Eingabedaten transformiert werden.

Zum Vergleich: bei der oben vorgestellten Schematron-Implementierung ist zum einen dieser Meta-Code in einer separaten Datei, der Schematron-Instanz, enthalten; zum anderen wird hier XSLT generiert, welches die Eingabedaten als Ganzes, d.h. nicht sequentiell, transformiert.

Ein relativ einfacher Anwendungsfall liegt vor, wenn der Meta-Code nur aus Pfadausdrücken besteht, die auf bestimmte Teile der Eingabedaten verweisen. Allerdings sind diese Ausdrücke als Teil der Eingabe zunächst nur einfache Zeichenketten. Eine direkte Auswertung ist damit nicht möglich.[16]

[16] Eine solche Zeichenkette muss *geparst* werden, um als Ausdruck erkannt und berechnet werden zu können. Vergleichbar wäre die Aufgabe, in Java oder C aus der Zeichenkette `"1+2"` die Zahl 3 zu bestimmen. Einige Skriptsprachen wie z.B. Perl bieten eine solche Funktionalität.

Auch in XSLT stoßen Anwender häufig auf die Beschränkung, dass XPath-Ausdrücke nicht dynamisch erstellt und ausgewertet werden können. Einige XSLT-Prozessoren bieten hier spezifische Erweiterungen, so beispielsweise *Saxon* [Saxon] die Funktion `saxon:evaluate` . Mit hoher Wahrscheinlichkeit wird auch die Version 2.0 von XSLT eine solche Funktionalität nicht standardmäßig enthalten, da sie die Kompilierbarkeit von XSLT-Stylesheets beträchtlich erschwert.

Nun unterstützt STX eine solche Funktion zwar ebenfalls nicht, allerdings lässt sich mit dem vorgestellten Filtermechanismus eine vergleichbare Funktionalität erreichen.

Das folgende in Listing 11 dargestellte Beispiel zeigt einen typischen Aufbau eines Dokuments, das zu Beginn Pfadausdrücke bzw. STXPath-Muster enthält.

Listing 11 *Eingabedaten, die Pfadausdrücke enthalten*

```
1  <?xml version="1.0"?>
2  <magic>
3    <head>
4      <important path="charm[2]/@spell" />
5      <important path="charm[4]/@spell" />
6    </head>
7    <body>
8      <charm spell="Abracadabra">All-purpose charm</charm>
9      <charm spell="Obliviate">
10        Modifies or erases portions of a person's memory.</charm>
11     <charm spell="Peskipiksi Pesternomi">Freezing charm?</charm>
12     <charm spell="Wingardium Leviosa">Causes a feather to levitate.</charm>
13   </body>
14 </magic>
```

Dieses Dokument ist in zwei Teile `head` und `body` gegliedert. Innerhalb des Elements `head` sind Pfade angegeben, die als Meta-Code auf die im `body`-Element enthaltenen Daten verweisen, siehe Zeilen 4 und 5. Die Aufgabe einer Transformation könnte es sein, nur die Daten der durch diese Pfade adressierten Knoten auszugeben.

Eine STX-Transformation muss damit den Inhalt des `head`-Elements in entsprechende STX-Transformationsanweisungen umwandeln, diese in einem Puffer abspeichern und schließlich den Inhalt des Puffers zur Verarbeitung des `body`-Elements benutzen.

Die in Listing 12 abgedruckte STX-Transformation generiert STX-Code, der die durch STXPath-Muster adressierten Knoten ausgibt.

Listing 12 *STX-Transformation, die dynamisch STX-Code generiert*

```
1  <?xml version="1.0"?>
2  <stx:transform xmlns:stx="http://stx.sourceforge.net/2002/ns"
3                 xmlns:alias="urn:stx-alias"
4                 version="1.0" output-method="text">
5
6    <stx:namespace-alias stylesheet-prefix="alias" result-prefix="stx" />
7
8    <!-- Puffer, der den generierten STX-Code aufnimmt -->
9    <stx:buffer name="code" />
10
11   <!-- Zwei Templates für die Codegenerierung -->
12   <stx:template match="head">
13     <stx:result-buffer name="code">
```

```
14        <!-- Das dynamisch erzeugte STX-Transformations-Sheet -->
15        <alias:transform version="1.0">
16          <!-- Ein Template, das für die Verarbeitung der Attribute sorgt -->
17          <alias:template match="*" priority="2">
18            <alias:process-attributes />
19            <alias:process-self />
20          </alias:template>
21          <stx:process-children />
22        </alias:transform>
23      </stx:result-buffer>
24    </stx:template>
25
26    <stx:template match="important">
27      <alias:template match="{@path}">
28        <alias:value-of select="concat(., '&#xA;')" />
29      </alias:template>
30    </stx:template>
31
32    <!-- Das Template, das den generierten Code anwendet -->
33    <stx:template match="body">
34      <stx:process-self filter-method="http://stx.sourceforge.net/2002/ns"
35                        filter-src="buffer(code)" />
36    </stx:template>
37
38  </stx:transform>
```

Da in dieser Transformation STX-Code erzeugt wird, muss er über ein Namensraum-Alias geschützt werden (Zeile 6). Alle folgenden Elemente mit dem Präfix `alias` werden damit zu STX-Elementen.

Das Template in Zeile 12 für das `head`-Element füllt den Puffer namens `code` mit einem vollständigen STX-Transformations-Sheet. Dieses enthält als erstes ein Template, das für jedes Element auch dessen Attribute verarbeiten lässt (Zeile 17). Seine höhere Priorität bewirkt, dass es Vorrang gegenüber allen anderen (noch zu generierenden Templates) besitzt. Ohne ein solches Template würde nur für Kindknoten, aber nicht für Attribute, nach passenden Templates gesucht werden. Das anschließende `stx:process-children` in Zeile 21 sorgt dann dafür, dass die unterhalb von `head` stehenden `important`-Elemente verarbeitet werden. In diesem Fall wird für jedes `important`-Element ein weiteres Template generiert, das den Inhalt des `path`-Attributes als `match`-Muster enthält (Zeile 27) und in seinem Inhalt den Wert des gefundenen Knotens ausgibt.

Damit wurde das `head`-Element in ein STX-Transformations-Sheet transformiert und im Puffer `code` abgespeichert.

Das letzte Template für `body` benutzt dann diesen dynamisch generierten STX-Code, um das `body`-Element selbst zu verarbeiten (Zeile 34).

Geplante Änderungen

Wie der Anwendungsfall der Web Services in Kapitel 6.3 zeigen wird, ist der hier beschriebene grundlegende Mechanismus zur Einbindung externer Filter verbesserungswürdig. In der gegenwärtigen Fassung lässt sich der Ausgabestrom eines externen Filters nicht durch STX-Templates verarbeiten, sondern mündet direkt in die Ausgabe der STX-Transformation. Würde der Filterausgabestrom stattdessen der aktuellen Transformation als Eingabe dienen, könnten Filterergebnisse ebenfalls transformiert werden. Eine entsprechende Änderung wird sich im kommenden Entwurf der STX-Spezifikation [STX] wieder finden.

5.6.8 Fehlerbehandlung und Logging in STX

Fehlerkategorien

Die in der Sprache STX auftretenden Fehler lassen sich grob in die folgenden Kategorien einteilen:

Syntaxfehler
sind Fehler, die auf nicht wohlgeformtes XML oder Verstöße gegen das Schema für STX zurückzuführen sind.

Statische Semantikfehler
sind Fehler, die zur Compile-Zeit erkannt werden können, z.B. der Aufruf einer unbekannten Funktion, die Referenzierung einer nichtdefinierten Variable oder die Verwendung eines bereits vergebenen Gruppennamens.

Dynamische Fehler
sind Fehler, die erst zur Laufzeit auftreten und von den konkreten Eingabedaten abhängen, wie beispielsweise die dynamische Erzeugung von Elementen, Kommentaren oder Verarbeitungsanweisungen.

Logische Fehler
sind Fehler, die nicht vom STX-Prozessor erkannt werden können, die sich jedoch durch falsche Transformationsergebnisse bemerkbar machen.

Syntaxfehler

Fehler der ersten Kategorie sind sehr leicht zu erkennen. So werden Fehler auf XML-Ebene bereits vom XML-Parser gemeldet. Verstöße gegen das XML-Schema (siehe Anhang A) können durch geeignete Validierungswerkzeuge erkannt werden. Auch ohne entsprechende Werkzeugunterstützung sollte ein STX-Prozessor bei der Analyse des STX-Transformations-Sheet immer Schemaverstöße melden und die Verarbeitung abbrechen. Häufige Ursachen für Fehler in dieser Kategorie sind Schreibfehler bei Element- oder Attributnamen, fehlende Attribute oder inkorrekte Verschachtelungen der einzelnen STX-Anweisungen.

In die gleiche Kategorie fallen Syntaxfehler innerhalb von STXPath-Ausdrücken. Für solche Ausdrücke existiert eine eigene Grammatik (siehe Anhang B.2), mit deren Hilfe der in einem STX-Prozessor enthaltene STXPath-Parser diese Ausdrücke analysieren, in eine interne Repräsentation überführen oder gegebenenfalls als fehlerhaft zurückweisen kann.

Statische Semantikfehler

Statische Semantikfehler dagegen lassen sich nicht allein durch die Validierung gegen das Schema für STX entdecken, sondern setzen Wissen über die Bedeutung der einzelnen STX-Anweisungen voraus. Dies sind im Wesentlichen Eindeutigkeitsanforderungen für alle benannten Elemente (Gruppen, Prozeduren, Variablen) sowie die Existenz des entsprechenden Elements bei der Referenzierung eines solchen Namens. Da STX den Einschluss anderer STX-Transformations-Sheets mittels `stx:include` vorsieht, lassen sich diese Eindeutigkeitsanforderungen nicht in XML-Schema ausdrücken. Die dort mittels `xs:unique`, `xs:key` und `xs:keyref` formulierbaren Bedingungen gelten immer nur innerhalb des Dokuments.

Für statische Semantikfehler gilt ebenfalls, dass ein STX-Prozessor einen solchen Fehler meldet und die Verarbeitung abbricht.

Dynamische Fehler

Schwieriger gestaltet sich die Situation für dynamische Fehler. Eine solche Fehlerquelle lässt sich nicht allein durch Analyse des STX-Transformations-Sheet entdecken. Beispielsweise kann die Anweisung

```
<stx:attribute name="{@attname}" select="@attval" />
```

fehlschlagen, weil an dieser Stelle im Resultat kein Attribut erzeugt werden kann (es wurde kein Element unmittelbar zuvor eröffnet) oder weil das Attribut `attname` des aktuellen Eingabeknotens entweder gar nicht existiert oder einen unerlaubten Wert für einen XML-Attributnamen enthält.

Behebbare Fehler in XSLT

In XSLT wurde eine Reihe der dynamischen Fehler als behebbar (*recoverable*) klassifiziert. Jede XSLT-Implementierung darf frei entscheiden, ob sie den Fehler meldet (und die Transformation beendet) oder fortfährt, indem sie die in der Spezifikation vorgesehene Aktion durchführt. Im Falle des oben angegebenen Beispiels einer dynamischen Attributerzeugung darf ein XSLT-Prozessor beispielsweise die entsprechende Anweisung `xsl:attribute` im Fehlerfall einfach ignorieren. Er ist noch nicht einmal dazu verpflichtet, den Anwender in irgendeiner Weise (etwa durch eine Warnung) vom Auftreten des Fehlers zu unterrichten.

Diese halbherzige Fehlerbehandlung ist ein Ergebnis des besonderen Einsatzgebietes von XSLT. Ein XSLT-Stylesheet, das im Browser des Anwenders die empfangenen XML-Daten in ein Präsentationsformat (in der Regel HTML) überführt, sollte den Anwender möglichst nie mit Fehlerausgaben konfrontieren. Auf der anderen Seite ist bei der Transformation in eigene, spezifische XML-Formate eine Fehlermeldung in solch einem Fall häufig wünschenswert, da das bloße Ignorieren einer Anweisung in der Regel nicht das erhoffte Ergebnis liefert.

In XSLT, das selbst keinen nutzerdefinierten Mechanismus zur Fehlerbehandlung enthält, wurden daher behebbare Fehler als Kompromiss zwischen diesen beiden Anwendungsfällen eingeführt. Diese Freiheit bei der Fehlerbehandlung kann sich jedoch als Problem für die Stylesheet-Entwicklung herausstellen. Im schlechtesten Fall besitzt ein Stylesheet-Autor einen sehr toleranten XSLT-Prozessor, der alle behebbaren Fehler ignoriert, während der Anwender oder Kunde einen strengen XSLT-Prozessor im Einsatz hat, der empfindlich auf diese Fehler reagiert. Beide XSLT-Prozessoren verhalten sich konform zur Spezifikation, trotzdem kann das Stylesheet nicht portabel auf beiden XSLT-Prozessoren eingesetzt werden. Im letzten Entwurf von XSLT 2.0 [W3C03b] hat sich an dieser Situation nichts geändert. Für Stylesheet-Autoren kann damit nur als Konsequenz folgen, einen möglichst strengen XSLT-Prozessor bei der Entwicklung zu benutzen.

Meldung von Fehlern in STX

In STX existiert dagegen keine Wahlfreiheit für STX-Prozessoren, einen dynamischen Fehler zu melden oder zu ignorieren. Insbesondere wurde die Variante, für alle diese Fehlersituationen ein Ausweichverhalten festzulegen, verworfen. Fehlermeldungen sind ein entscheidendes Hilfsmittel, um die Korrektheit eines Programms sicherzustellen (vergleiche auch [SMQ03]).

Stattdessen wird STX einen einfachen Fehlerbehandlungsmechanismus anbieten,[17] der dem STX-Autor die Entscheidung überlässt, wie mit bestimmten Fehlersituation umgegangen werden soll. Dieser Mechanismus orientiert sich an dem aus objektori-

[17] Das hier beschriebene Fehlerbehandlungskonzept findet sich zum Zeitpunkt der Entstehung dieser Arbeit noch nicht in der STX-Spezifikation wieder. Ebenso wenig existiert derzeit eine Implementation dafür.

entierten Sprachen bekannten Konzept der Ausnahmen (*Exceptions*). Beim Auftreten eines Laufzeitfehlers wird eine Ausnahme ausgelöst, die separat an einer geeigneten Stelle im Code behandelt werden kann. Insbesondere werden Anwendungslogik und Fehlerbehandlung voneinander getrennt.

Fehlertypen

Für das Transformationsmodell in STX bedeutet dies, dass zunächst alle potenziellen dynamischen Fehler identifiziert und mit einem Namen versehen werden. Beispielsweise bezeichnet `invalid-name` den Fehler, der bei dem Versuch auftritt, eine Zeichenkette, die keinen XML-Namen repräsentiert, für die dynamische Konstruktion eines Elements, Attributes etc. zu verwenden. Die STX-Spezifikation enthält eine Liste aller möglichen dynamischen Fehler. Diese bilden keine Hierarchie, sondern stehen gleichberechtigt nebeneinander.

`stx:recover`

Ein solcher dynamischer Fehler kann in STX durch das Element `stx:recover` behandelt werden. Dieses Element verhält sich wie ein Template. Es kann innerhalb einer Gruppe neben den Templates angegeben werden und es gelten die gleichen Sichtbarkeitsregeln und Vorrangkategorien wie für Templates (siehe Kapitel 5.6.4). Während ein Template jedoch wie ein Callback-Element für die Knoten der XML-Eingabe aufgerufen wird, fungiert ein `stx:recover`-Element als Callback für dynamische Fehler. Der Name des Elementes verdeutlicht, dass in seinem Inhalt eine *Recovery*-Aktion definiert werden kann. Ob und wie ein dynamischer Fehler behoben werden soll, wird durch den STX-Autor durch die Bereitstellung geeigneter `stx:recover`-Elemente festgelegt.

Das `stx:recover`-Element besitzt ein optionales Attribut `from`, das die konkreten Fehlertypen bezeichnet. Ohne die Angabe dieses Attributs werden alle behebbaren dynamischen Fehler durch das `stx:recover`-Element behandelt. Ein spezifisches `stx:recover`-Element mit `from`-Attribut hat immer den Vorrang vor einem allgemeinen `stx:recover`-Element. Soll ein `stx:recover` für mehrere Fehlertypen zuständig sein, werden deren Namen durch Leerzeichen getrennt im `from`-Attribut angegeben. Mehrere `stx:recover`-Elemente für den gleichen Fehlertyp in der gleichen Vorrangkategorie führen zu einem statischen Fehler. Existiert im Fehlerfall kein passendes `stx:recover`-Element innerhalb der Vorrangkategorien, wird ein normaler Fehler ausgelöst. Dies führt zum Abbruch der Transformation.

Fehlerparameter

Informationen über den konkreten Fehler werden in Form von speziellen Parametern übergeben. Die Syntax ähnelt hier ebenfalls der von Templates (und auch Prozeduren), wobei die Parameterübergabe implizit durch den STX-Prozessor geschieht. Folgende Parameter werden übergeben:

`error`
: der aufgetretene Fehlertyp als Zeichenkette

`message`
: eine vom STX-Prozessor erzeugte Fehlermeldung, die den aufgetretenen Fehler näher beschreibt

`element`
: der Name des Elements, das den Fehler ausgelöst hat

`system-id`
: die System-ID des Transformations-Sheet, zu dem dieses Element gehört

`line-number`
: die Zeilennummer des Elements, das den Fehler ausgelöst hat (sofern vorhanden)

`column-number`
die Spaltennummer des Elements, das den Fehler ausgelöst hat (sofern vorhanden)

Für nicht benötigte Parameterwerte kann die Parameterdeklaration entfallen. Beispielsweise hat in einem spezifischen `stx:recover`-Element der Parameter `error` immer einen bekannten Wert und ist daher irrelevant. Alle Parameter eines `stx:recover`-Elements sind automatisch obligatorisch, d.h. ihr `required`-Attribut besitzt implizit den Wert "`yes`". Auf diese Weise können Schreibfehler in der Parameterdeklaration durch den STX-Prozessor erkannt werden.

Innerhalb des `stx:recover`-Elements kann im einfachsten Fall mittels `stx:message` eine Fehlermeldung ausgegeben werden. Gegebenenfalls kann über das Attribut `terminate` die Transformation sogar beendet werden.

Das Element `stx:recover` ist jedoch in erster Linie dazu gedacht, die Transformation trotz eines aufgetretenen Fehlers sinnvoll fortzusetzen. Nachdem alle innerhalb von `stx:recover` vorkommenden Elemente ausgeführt worden sind, wird die Transformation nach dem Element, das den Fehler ausgelöst hatte, fortgesetzt. Der Inhalt des fehlerhaften Elementes wird ignoriert.

Falls innerhalb von `stx:recover` wiederum ein behandelbarer Fehler auftritt, wird ebenso ein passendes `stx:recover`-Element gesucht und benutzt. Allerdings werden alle aktiven `stx:recover`-Elemente bei dieser Suche ausgeschlossen. So kann es nicht zu unendlichen Fehlerbehandlungsschleifen kommen.

Logische Fehler

Fehler der letzten Kategorie kann ein STX-Prozessor nicht allein erkennen. Hier ist der STX-Programmierer auf zusätzliche Hilfsmittel angewiesen, mit deren Hilfe der Transformationsablauf verfolgt und damit die Fehlerursache ermittelt werden kann. Mit Hilfe des Elements `stx:message` lassen sich bereits einfache Debug-Ausschriften erzeugen. Weitaus mächtiger sind spezielle STX-Debugger, wie zum Beispiel der für *Joost* entwickelte *StDB* [StDB].

Logging

STX-basierte Transformationen laufen häufig im Hintergrund ab. Typisch sind so genannte Batch-Prozesse, die für die Verarbeitung einer großen Datenmenge eine gewisse Zeit benötigen, oder Transformationen innerhalb eines Servers, in dem XML-Daten auf Anfrage transformiert und an die Klienten ausgeliefert werden. In beiden Szenarien kommt der Anwender nicht direkt mit dem STX-Prozessor in Kontakt. Um so wichtiger ist es, dass eventuell auftretende Fehlernachrichten oder Mitteilungen über den Status der Transformation in entsprechenden Log-Dateien aufgezeichnet werden können. Das aus XSLT übernommene Element `stx:message` bietet nur die Erzeugung einer einfachen Nachricht, ohne dass auf STX-Ebene verschiedene Empfänger oder unterschiedliche Schweregrade eines Fehlers unterschieden werden können.

Für verschiedene Programmiersprachen wurden bereits geeignete APIs entwickelt, die ein komfortables Erzeugen von Log-Nachrichten ermöglichen. Insbesondere für die Programmiersprache Java existieren gleich mehrere solcher APIs [Sei03].

Eigenschaften von Logging-APIs

Logging-APIs zeichnen sich durch die folgenden Punkte aus:

- Es können im Quelltext Nachrichten unterschiedlichen Schweregrads produziert werden.

- Es können unterschiedliche Empfänger (*Logger*) verwendet werden.
- Es stehen häufig umfangreiche Konfigurierungsmöglichkeiten zur Verfügung, über die sich das Ausgabeformat und die Zuordnung eines Loggers zu einem physikalischen Ziel (z.B. einer Datei) einstellen lassen.

Insbesondere erfolgt die Konfiguration des Logging-Framework unabhängig von der Erzeugung der eigentlichen Nachricht. Eine einmal erstellte Anwendung kann so unabhängig vom Quelltext hinsichtlich ihres Log-Verhaltens konfiguriert werden, in der Regel über separate Konfigurationsdateien.

Die Sprache STX enthält kein eigenes Logging-Framework, sondern bietet eine einfache Schnittstelle zu darunter liegenden Logging-Mechanismen. Die jeweilige Konfigurierung muss implementationsspezifisch durch den jeweiligen STX-Prozessor gelöst werden.

`stx:message`

Das Element `stx:message` wird zu diesem Zweck um zwei Attribute erweitert:

`level`
: Der Schweregrad der Nachricht. Mögliche Werte sind in aufsteigender Reihenfolge `"trace"`, `"debug"`, `"info"`, `"warn"`, `"error"` und `"fatal"`.[18]

`logger`
: Der Name eines Loggers, der das Ziel der Nachricht repräsentiert.

Ein STX-Prozessor ohne Logging-Unterstützung darf diese Attribute ignorieren und alle per `stx:message` erzeugten Nachrichten gleich behandeln.

5.7 Typische STX-Transformationstypen

Im Folgenden werden einige typische Transformationsaufgaben, für die sich eine STX-Lösung anbietet, kurz vorgestellt und diskutiert.

5.7.1 Datenfilter

Unter einem *Datenfilter* soll eine Transformation verstanden werden, die eine Teilmenge der Eingabedaten produziert. Dabei werden nur bestimmte Daten extrahiert und ohne Veränderung in die Ausgabe kopiert. Für die Realisierung solcher Datenfilter existieren zwei Vorgehensweisen:

Explizites Einschließen

In diesem Fall müssen Daten, die in der Ausgabe erscheinen sollen, in speziellen Regeln explizit kopiert werden. Alle anderen XML-Daten werden durch das Setzen des Attributs `pass-through` auf den Wert `"none"` standardmäßig ignoriert.

Listing 13 *Kopieren auf Anforderung*

```
1 <stx:transform xmlns:stx="http://stx.sourceforge.net/2002/ns"
2                version="1.0" pass-through="none">
```

[18] Diese sechs Stufen wurden in Anlehnung an die Logging-Stufen im API *Commons Logging* des Jakarta-Projekts [ASFc] gewählt, siehe http://jakarta.apache.org/commons/logging/. Das bedeutet jedoch nicht, dass die Implementation eines STX-Prozessors ebenfalls dieses API benutzen muss.

```
3     <stx:template match="data[@important='true']">
4       <stx:process-self group="copy" />
5     </stx:template>
6     <stx:group name="copy" pass-through="all" />
7   </stx:transform>
```

Die im Listing 13 angegebenen Regeln erlauben das Kopieren ganzer Teilbäume der Eingangsdaten. In diesem Fall werden die Elemente, die das Muster im `match`-Attribut des Template erfüllen (hier genau die `data`-Elemente, deren Attribut `important` den Wert `"true"` besitzt), mit der Anweisung `stx:process-self` in Zeile 4 an die Gruppe namens `"copy"` in Zeile 6 zur weiteren Verarbeitung verwiesen. Da deren `pass-through`-Attribut auf den Wert `"all"` gesetzt ist und sie selbst keine weiteren Templates enthält, wird das Element selbst einschließlich aller enthaltenen Daten in die Ausgabe kopiert.

Explizites Ausschließen

Dies ist genau die umgekehrte Variante des letzten Beispiels. Prinzipiell wird alles kopiert (`pass-through="all"`), es sei denn, ein passendes Template verhindert dies. Wie Listing 14 zeigt, kann eine solche einfache Transformation bereits mit 4 Zeilen realisiert werden.

Ignorieren auf Anforderung **Listing 14**

```
1  <stx:transform xmlns:stx="http://stx.sourceforge.net/2002/ns"
2                 version="1.0" pass-through="all">
3    <stx:template match="data[@important='false']" />
4  </stx:transform>
```

Beide Varianten lassen sich sehr einfach miteinander kombinieren, indem innerhalb der explizit zu kopierenden Daten wiederum Teile ausgeschlossen werden. Dies erreicht man durch das Erweitern der Gruppe `"copy"` in Listing 13 um solche Templates, wie sie Listing 14 enthält.

5.7.2 Umbenennungen

Bei dieser Klasse von Transformationen bleiben die XML-Daten strukturell bis auf Namensänderungen unverändert. Solche Umbenennungen betreffen in der Regel die Namen von Elementen oder Attributen.

Eine STX-Transformationsvorlage für diesen Zweck kopiert als Standardregel alle Knoten der Eingabe und enthält zusätzliche Regeln für neu zu benennende Bestandteile.

Es handelt sich ausschließlich um Elementnamen

In diesem Fall müssen ausschließlich zusätzliche Regeln für diese Elemente angegeben werden:

Einfache Elementumbenennung in STX **Listing 15**

```
1  <stx:transform xmlns:stx="http://stx.sourceforge.net/2002/ns"
2                 version="1.0" pass-through="all">
```

```
3    <stx:template match="frage">
4      <question>
5        <stx:process-attributes />
6        <stx:process-children />
7      </question>
8    </stx:template>
9  </stx:transform>
```

Alle `frage`-Elemente werden durch die Transformationsvorlage in Listing 15 durch `question`-Elemente ersetzt. Eventuelle Attribute der `frage`-Elemente werden separat verarbeitet und damit durch die Vorgaberegel kopiert. Besitzt ein umzubenennendes Element keine Attribute, kann die Anweisung `stx:process-attributes` in Zeile 5 entfallen.

Es handelt sich um frei vorkommende Attributnamen

Neben einer Regel für die umzubenennenden Attribute (Zeile 9 in Listing 16) ist hier eine zusätzliche Regel (Zeile 3) notwendig, die auf Elemente passt. Diese sorgt dafür, dass überhaupt Attribute durch Regeln bearbeitet werden.

Listing 16 *Einfache Attributumbenennung in STX*

```
1  <stx:transform xmlns:stx="http://stx.sourceforge.net/2002/ns"
2                 version="1.0" pass-through="all">
3    <stx:template match="*">
4      <stx:copy>
5        <stx:process-attributes />
6        <stx:process-children />
7      </stx:copy>
8    </stx:template>
9    <stx:template match="@xml:lang">
10     <stx:attribute name="lang" select="." />
11   </stx:template>
12 </stx:transform>
```

Die im Listing 16 dargestellten Transformationsregeln ersetzen beispielsweise alle `xml:lang`-Attribute durch einfache `lang`-Attribute.

Namensraumänderungen

Die Änderung von Namensräumen tritt vorrangig während der Entwicklung eines XML-Vokabulars auf. Bereits vorhandene Dokumente, die den alten Namensraum benutzen, sollen in den neuen Namensraum überführt werden.

Auf rein textueller Ebene ist das Problem einfach zu lösen. Es muss lediglich die Namensraumdeklaration geändert werden. Bei vernünftigem Einsatz von Namensräumen bedeutet dies, dass am Anfang des Dokuments genau eine Änderung pro Namensraum vorgenommen werden muss. Wenn die zu transformierenden Daten jedoch nicht als Text vorliegen, sondern bereits als interne Repräsentation (als Instanz eines Datenmodells) geliefert werden, steht diese Variante nicht zur Verfügung.

Auf der Datenmodell-Ebene ist eine solche Transformation etwas aufwändiger. Die Änderung eines Namensraumes bedeutet nämlich, dass jedes Element (bzw. Attribut) geändert werden muss, das sich in dem betreffenden Namensraum befindet.

Das folgende Beispiel demonstriert eine solche Änderung für Elemente.

Änderung von Namensräumen in STX **Listing 17**

```
1  <stx:transform xmlns:stx="http://stx.sourceforge.net/2002/ns"
2                 xmlns:src="source-namespace"
3                 version="1.0" pass-through="all">
4    <stx:template match="src:*">
5      <stx:element name="{name()}" namespace="target-namespace">
6        <stx:process-attributes />
7        <stx:process-children />
8      </stx:element>
9    </stx:template>
10 </stx:transform>
```

Das Template in Zeile 4 bearbeitet nur Elemente aus dem `src`-Namensraum. Die enthaltene Anweisung `stx:element` in Zeile 5 generiert dann ein Element gleichen Namens in dem abweichenden Namensraum *`target-namespace`*.

5.7.3 Umwandlung zwischen Elementen und Attributen

Bei der Modellierung eines XML-Vokabulars stellt sich häufig die Frage, ob bestimmte Daten als Attribute oder innerhalb von Unterelementen repräsentiert werden sollen. In vielen Fällen gibt es hier keine eindeutig richtige Antwort, sodass beide Varianten sinnvoll erscheinen. Entsprechend häufig existiert damit auch die Notwendigkeit, die eine Darstellung in die andere zu überführen.

Umwandlung von Attributen in Elemente

Die Umwandlung von Attributen in Elemente ist unproblematisch, da die Daten aller Attribute während der Verarbeitung des dazugehörigen Elements verfügbar sind. Beispielsweise wandelt das in Listing 18 wiedergegebene Template alle Attribute dynamisch in gleichnamige Unterelemente um.

Umwandlung von Attributen in Elemente **Listing 18**

```
1  <stx:template match="*">
2    <stx:copy>
3      <stx:for-each-item select="@*" name="att">
4        <stx:element name="{name($att)}">
5          <stx:value-of select="$att" />
6        </stx:element>
7      </stx:for-each-item>
8      <stx:process-children />
9    </stx:copy>
10 </stx:template>
```

Zunächst wird das Element ohne Attribute mittels `stx:copy` kopiert. Innerhalb der Anweisung `stx:for-each-item` in Zeile 3 werden dann alle Attribute durchlaufen und für jedes Attribut ein Unterelement erzeugt.

Umwandlung von Elementen in Attribute

Für die andere Richtung muss zunächst vorausgesetzt werden, dass die fraglichen Elemente ausschließlich Textinhalt besitzen und keine weiteren Unterelemente ent-

halten. In der Transformation wird dann jedes Attribut separat durch eine Template-Instanz erzeugt.

Zur Illustration wird das folgende XML-Fragment als Eingabe verwendet:

```
<person>
  <name>von Humboldt</name>
  <vornamen>Alexander Georg</vornamen>
</person>
```

Das Transformations-Sheet in Listing 19 wandelt diese Eingabe in das leere Element

```
<person name="von Humboldt" vornamen="Alexander Georg" />
```

um.

Listing 19 *Umwandlung von Elementen in Attribute*

```
1 <stx:transform xmlns:stx="http://stx.sourceforge.net/2002/ns"
2                version="1.0" pass-through="all" strip-space="yes">
3   <stx:template match="person/*">
4     <stx:attribute name="{name()}" select="." />
5   </stx:template>
6 </stx:transform>
```

Das Template in Zeile 3 verarbeitet die Kindelemente von `person` und erzeugt für diese ein gleichnamiges Attribut. Das Element `person` selbst wurde durch ein Vorgabe-Template in das Ergebnis kopiert. Durch das zusätzliche Attribut `strip-space` im Wurzelelement `stx:transform` muss jedoch verhindert werden, dass der Leerraum innerhalb von `person` ebenfalls kopiert wird. Die anschließende dynamische Erzeugung von Attributen würde hier sonst einen Laufzeitfehler bewirken.

Es gibt Fälle, in denen sich der Wert eines zu erzeugenden Attributs aus mehreren Elementen der Eingabe zusammensetzen soll, etwa

```
<person name="Alexander Georg von Humboldt" />
```

In diesem Beispiel würde man zunächst den Inhalt des Elements `person` verarbeiten. Dabei wären die in den Unterelementen abgelegten Daten in Variablen zu speichern. Anschließend kann die gewünschte Ausgabe erzeugt werden.

Das Transformations-Sheet in Listing 20 deklariert zu diesem Zweck zunächst zwei Variablen `name` und `vornamen` (Zeile 3), die später den Inhalt der gleichnamigen Elemente aufnehmen (Zeilen 16 und 19).

Listing 20 *Umwandlung mehrerer Elemente in ein Attribut*

```
1  <stx:transform xmlns:stx="http://stx.sourceforge.net/2002/ns" version="1.0">
2
3    <!-- Deklarieren der Variablen -->
4    <stx:variable name="name" />
5    <stx:variable name="vornamen" />
6
7    <stx:template match="person">
8      <!-- Erst werden die Kindknoten verarbeitet ... -->
9      <stx:process-children />
10     <!-- ... danach wird die Ausgabe erzeugt -->
```

```
11      <person name="{$vornamen} {$name}" />
12    </stx:template>
13
14    <!-- Speichern der Elementwerte in den Variablen -->
15    <stx:template match="name">
16      <stx:assign name="name" select="." />
17    </stx:template>
18    <stx:template match="vornamen">
19      <stx:assign name="vornamen" select="." />
20    </stx:template>
21
22  </stx:transform>
```

Die in Listing 20 demonstrierte Vorgehensweise zeigt das allgemeine Lösungsmuster für solche Transformationsprobleme, in denen erst nach der vollständigen Verarbeitung des Inhalts eines Elements eine geeignete Ausgabe generiert werden kann.

5.7.4 Tabellen

Tabellen sind ein Layoutmittel, in dem Daten allein in Zeilen und Spalten verwaltet werden. Die Bedeutung eines konkreten Tabellenfeldes ergibt sich aus seiner Position innerhalb der Tabelle. Eine typische Transformationsaufgabe besteht darin, solche Tabellen und spezifische XML-Strukturen ineinander zu überführen.

Transformation einer Tabelle in eine XML-Struktur

In diesem Fall sollen generische Tabellenfelder durch spezifische Elemente ersetzt werden. Beispielsweise soll die Tabellenzeile

```
<row><col>von Humboldt</col><col>Alexander Georg</col></row>
```

wieder in ein `person`-Element mit den Unterelementen `name` und `vornamen` umgewandelt werden. Die Zuordnung der semantischen Bedeutung zu den einzelnen Tabellenfeldern lässt sich mit Hilfe von Positionsprädikaten sehr einfach ausdrücken. Listing 21 zeigt die für diese Transformation relevanten Templates.

Transformation eines generischen Tabellenmodells **Listing 21**

```
1  <stx:template match="col[1]">
2    <name>
3      <stx:value-of select="." />
4    </name>
5  </stx:template>
6  <stx:template match="col[2]">
7    <vornamen>
8      <stx:value-of select="." />
9    </vornamen>
10 </stx:template>
```

Der Einfachheit halber sei hier angenommen, dass jedes Tabellenfeld (`col`) nur einfachen Text als Inhalt besitzt und keine Unterelemente. In diesem Fall lässt sich mittels `stx:value-of` direkt auf diesen Text zugreifen.

Transformation einer XML-Struktur in eine Tabelle

Diese Aufgabe besitzt große Ähnlichkeit mit der Umwandlung von Elementen in Attribute aus Kapitel 5.7.3. Sind immer alle spezifischen Elemente in der Eingabe vorhanden und liegen diese zudem in einer festen Reihenfolge vor, dann genügt eine einfache Regel, siehe Listing 22.

Listing 22 *Transformation bei vollständiger und geordneter Eingabe*

```
<stx:template match="name | vornamen">
  <col>
    <stx:process-children />
  </col>
</stx:template>
```

Schwieriger ist der Fall, wenn die Reihenfolge nicht garantiert ist oder sogar Elemente fehlen. Die Benutzung von XML mit spezifischen Elementen zur Datenrepräsentation ermöglicht es gerade, die Bedeutung jedes einzelnen Wertes anhand des eingesetzten Markup zu erkennen, ohne auf Positionen o.ä. zurückgreifen zu müssen.

In einer solchen Transformation müssen nun die Werte in eine feste Position gebracht und gegebenenfalls leere Felder für fehlende Werte generiert werden. Das bedeutet, dass eine Tabellenzeile erst dann erzeugt werden kann, wenn der Datensatz vollständig eingelesen wurde. Für Probleme dieser Art wurde bereits in Listing 20 auf Seite 108 das generelle Vorgehen gezeigt. In diesem Fall wird es nun für die Erzeugung einer Tabelle leicht erweitert, indem die Variablen zu Beginn jedes Datensatzes (`person`) zurückgesetzt werden (siehe Zeile 7 in Listing 23).

Listing 23 *Transformation bei unvollständiger Eingabe*

```
<stx:transform xmlns:stx="http://stx.sourceforge.net/2002/ns"
               version="1.0" pass-through="all">

  <!-- Deklarieren der Variablen wie bekannt ... -->

  <stx:template match="person">
    <!-- Rücksetzen der Variablen -->
    <stx:assign name="name" />
    <stx:assign name="vornamen" />

    <!-- Verarbeiten der Kindelemente und dabei Belegen der Variablen -->
    <stx:process-children />

    <!-- Ausgabe des Ergebnisses -->
    <row>
      <col>
        <stx:value-of select="$name" />
      </col>
      <col>
        <stx:value-of select="$vornamen" />
      </col>
    </row>
  </stx:template>

  <!-- Speichern der Elementwerte in den Variablen wie bekannt ... -->
```

```
26
27 </stx:transform>
```

5.7.5 Rekursive Strukturen

In rekursiven Strukturen wiederholt sich der Typ des Elternelements in seinem Inhalt. Das XML-Fragment in Listing 24 veranschaulicht diesen Fall, indem es `person` als Kindelement von `person` verwendet.

Die Familie von Humboldt **Listing 24**

```
1  <person>
2    <name>von Humboldt</name>
3    <vornamen>Alexander Georg</vornamen>
4    <person>
5      <name>von Humboldt</name>
6      <vornamen>Friedrich Wilhelm Christian Karl Ferdinand</vornamen>
7    </person>
8    <person>
9      <name>von Humboldt</name>
10     <vornamen>Friedrich Heinrich Alexander</vornamen>
11   </person>
12 </person>
```

Diese Struktur soll nun analog zu der in Kapitel 5.7.3 vorgestellten Aufgabe in die folgende Struktur transformiert werden:

Die Familie von Humboldt, Version 2 **Listing 25**

```
1 <person name="Alexander Georg von Humboldt">
2   <person name="Friedrich Wilhelm Christian Karl Ferdinand von Humboldt" />
3   <person name="Friedrich Heinrich Alexander von Humboldt" />
4 </person>
```

Die in Listing 20 auf Seite 108 vorgestellte Methode, nach der zunächst die Kindknoten vollständig verarbeitet und gespeichert werden, bevor eine Ausgabe erzeugt wird, ist hier jedoch nicht anwendbar. So muss die Ausgabe des `person`-Elements für den Vater vor seinen Söhnen geschehen. Dies kann nicht an das Ende des dazugehörigen Eingabeelements verschoben werden.

Allerdings kann die Ausgabe eines neuen `person`-Elements solange verzögert werden, bis dessen erstes `person`-Kindelement in der Eingabe auftritt. In diesem Fall benötigt man ein Template, das auf den ersten Kindknoten eines `person`-Elements passt (`match="person/node()[1]"`). Dieses Template steuert dann mit Hilfe der Anweisung `stx:process-siblings`, wieviele Geschwister dieses Kindknotens verarbeitet werden sollen. In unserem Fall muss die Verarbeitung beim nächsten `person`-Element unterbrochen werden. Nun kann die gewünschte Ausgabe erzeugt und anschließend per `stx:process-siblings` die Verarbeitung mit allen Geschwistern fortgesetzt werden.

Listing 26 zeigt dieses Template als den Kern einer solchen Transformation. Die benötigten Variablendeklarationen sowie die Templates, die diese Variablen mit Werten belegen, wurden bereits in den anderen Beispielen beschrieben.

Listing 26 *Rekursives Transformieren von Elementen in Attribute*

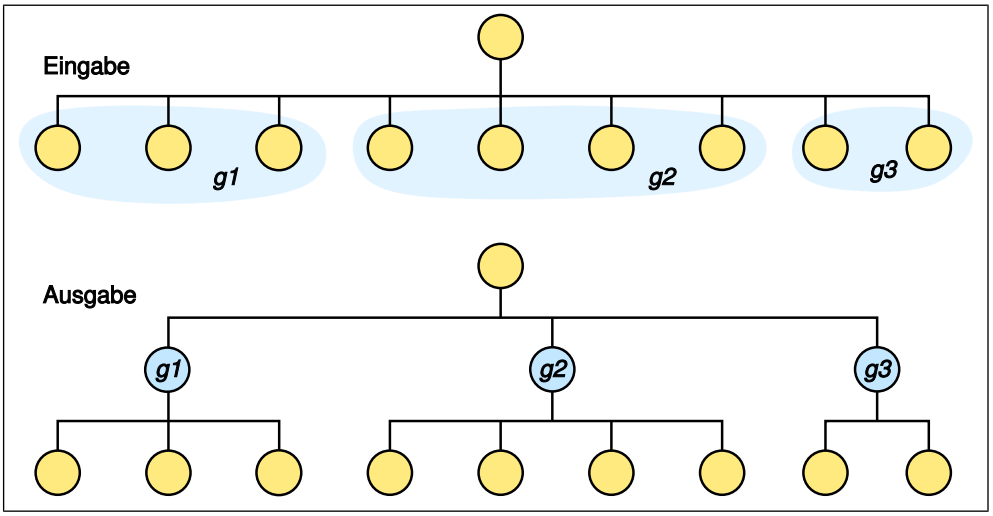
```
<stx:template match="person/node()[1]">
  <!-- Variablen zurücksetzen -->
  <stx:assign name="name" />
  <stx:assign name="vornamen" />
  <!-- diesen Knoten verarbeiten -->
  <stx:process-self />
  <!-- alle Geschwister bis zum ersten <person>-Element verarbeiten -->
  <stx:process-siblings until="person" />
  <!-- jetzt die Ausgabe erzeugen -->
  <person name="{$vornamen} {$name}">
    <!-- und den Rest der Geschwister verarbeiten -->
    <stx:process-siblings />
  </person>
</stx:template>
```

In zwei Fällen würde dieses Template jedoch nicht wie gewünscht funktionieren: zum einen, wenn der erste Kindknoten bereits ein `person`-Element ist; zum anderen, wenn leere `person`-Elemente in der Eingabe vorkommen. Beide Fälle lassen sich jedoch mit zusätzlichen Templates leicht behandeln.

5.7.6 Sequentielles Gruppieren

Beim sequentiellen Gruppieren besteht die Aufgabe darin, eine gewisse Anzahl aufeinander folgender Elemente einer Liste zusammenzufassen. Die zusammengehörigen Elemente bilden eine Gruppe und sollen ein gemeinsames neues Elternelement bekommen. In Abbildung 8 wird diese Aufgabe mit Hilfe der Baumdarstellung verdeutlicht. Es wird eine zusätzliche Ebene von Elementen `g1` bis `g3` eingefügt, auf die die Liste der Kindknoten abschnittsweise aufgeteilt wird.

Abbildung 8
Sequentielles Gruppieren



Gruppenbildung

Es gibt mehrere Möglichkeiten, die Gruppe zu identifizieren:

- Es werden die Positionen der Elemente in der ursprünglichen Liste benutzt, etwa bei der Überführung der Liste in eine Tabelle, die eine vorgegebene Anzahl Spalten besitzt.

- Es werden Informationen des verwendeten Markup benutzt. Beispielsweise werden in XHTML Abschnitte durch den Typ der vorangehenden Überschrift definiert.
- Es werden Daten als Gruppenmerkmal benutzt, indem z.B. Gruppen mit jeweils gleichem Anfangsbuchstaben aus einer sortierten Liste gebildet werden

Prinzipiell müssen Gruppierungsprobleme in STX anders gelöst werden als in XSLT. Während in XSLT das Bestimmen der Mitglieder einer Gruppe auf funktionalem Weg häufig eine Herausforderung an den Stylesheet-Autor darstellt, muss in STX aufgrund der sequentiellen Verarbeitung der Daten eine prozedurale Lösung gefunden werden. Ziel ist es, die einzelnen Gruppenmitglieder sofort in den Ausgabestrom zu schreiben. Das zwischenzeitliche Speichern in einem STX-Puffer ist zwar möglich, stellt hier aber keine zufrieden stellende Lösung dar, da die Verarbeitung unnötig Speicherplatz und Laufzeit verbraucht.

Lösungsansätze

STX bietet zwei grundsätzliche Lösungsansätze:

1. die Auswahl der aktuellen Gruppenmitglieder mit Hilfe der Anweisung `stx:process-siblings`
2. das getrennte Erzeugen von Start und Ende des umschließenden Elements per `stx:start-element` und `stx:end-element`

Einsatz von `stx:process-siblings`

Die Anweisung `stx:process-siblings` bearbeitet alle nachfolgenden Geschwisterknoten, die ein vorgegebenes Muster erfüllen (Attribut `while`) bzw. nicht erfüllen (Attribut `until`). Zunächst soll eine Lösung für den oben genannten ersten Anwendungsfall angegeben werden, in dem eine Gruppierung anhand von Positionen erfolgt. Eine flache Liste bestehend aus `entry`-Elementen soll in eine Tabelle umgewandelt werden. Mit Hilfe der Funktion `position` lässt sich im `until`-Attribut auf die Position der Elemente innerhalb dieser Liste zugreifen. Es werden nun alle Geschwisterelemente bearbeitet, bis eines an einer Position gefunden wird, die eine neue Zeile verlangt. Diese speziellen Positionen sind alle Vielfachen der Spaltenanzahl der gewünschten Tabelle, siehe Listing 27.

Positionales Gruppieren, Version 1 **Listing 27**

```
<stx:template match="entry">
  <tr>
    <stx:process-self group="entry-group" />
    <stx:process-siblings until="entry[(position()-1) mod $cols = 0]"
                          group="entry-group" />
  </tr>
</stx:template>

<stx:group name="entry-group">
  <stx:template match="entry">
    <td>
      <stx:process-children />
    </td>
  </stx:template>
</stx:group>
```

Die `until`-Bedingung in Zeile 4 bewirkt, dass alle folgenden `entry`-Elemente vor der Ausgabe des schließenden `tr` bearbeitet werden. Das erste Element, das sich an einer Position direkt nach einem Vielfachen der Spaltenzahl (`$cols`) befindet, beendet die Gruppe.

Die eigentliche Umwandlung der `entry`-Elemente in `td`-Elemente muss hier in einem anderen Template geschehen, für das zu diesem Zweck eine spezielle Gruppe angelegt wurde (Zeile 9). Anderenfalls würde das erste Template für die folgenden Geschwister erneut benutzt werden und somit eine verschachtelte `tr`-Struktur erzeugen.

Beispiel: Verarbeitung von XHTML

Der zweite Anwendungsfall, die Verwendung des enthaltenen Markup am Beispiel von XHTML, soll hier ebenfalls mittels `stx:process-siblings` gelöst werden. Ein Abschnitt wird in XHTML durch eine Überschrift (`h1`, `h2`, usw.) eingeleitet und reicht bis zur nächsten Überschrift der gleichen oder einer übergeordneten Stufe. So beginnt beispielsweise mit einem `h2`-Element ein Unterkapitel. Diesem Element folgende `h3`-Elemente beschreiben eine weitere Unterteilung innerhalb dieses Unterkapitels. Das Unterkapitel selbst wird entweder durch das nächste Unterkapitel (ein folgendes `h2`-Element) oder durch ein neues Kapitel (ein folgendes `h1`-Element) beendet.

Die Transformation einer durch `h2` eingeleiteten Folge von XHTML-Elementen, die zum gleichen Unterkapitel gehören, in eine *DocBook*-ähnliche Struktur, lässt sich damit folgendermaßen beschreiben:

Listing 28 *Gruppieren anhand des Markup*

```
1 <stx:template match="h2">
2   <sect2>
3     <title><stx:value-of select="." /></title>
4     <stx:process-siblings until="h1 | h2" />
5   </sect2>
6 </stx:template>
```

Das Muster im `until`-Attribut in Zeile 4 beendet die Verarbeitung der Geschwister, wenn ein `h1`- oder ein `h2`-Element in der Eingabe gefunden wird.

Getrenntes Erzeugen von Start und Ende eines Elements

Diese Vorgehensweise entspricht dem Erzeugen von Tags auf textueller Ebene. Sie birgt allerdings die Gefahr in sich, dass bei fehlerhafter Programmierung der erzeugte XML-Ausgabestrom nicht mehr wohlgeformt ist. Ein STX-Prozessor meldet allerdings solche Fehler.

Dieser Lösungsansatz soll in einer alternativen Lösung für den bereits besprochenen Anwendungsfall der Gruppierung anhand von Positionen demonstriert werden. Bei der Bearbeitung eines `entry`-Elements muss nun gegebenenfalls eine neue Tabellenzeile begonnen werden. Dies geschieht wiederum bei allen Vielfachen der Spaltenzahl (Variable `$cols`) durch Beenden des aktuell geöffneten `tr`-Elements und Öffnen eines neuen.

Listing 29 *Positionales Gruppieren, Version 2*

```
1 <stx:template match="list">
2   <table>
```

```
3       <tr>
4         <stx:process-children />
5       </tr>
6     </table>
7   </stx:template>
8
9   <stx:template match="entry">
10    <stx:if test="(position()-1) mod $cols = 0 and position() != 1">
11      <stx:end-element name="tr" />
12      <stx:start-element name="tr" />
13    </stx:if>
14    <td>
15      <stx:process-children />
16    </td>
17  </stx:template>
```

Die `stx:if`-Anweisung in Zeile 10 nutzt die durch die Funktion `position` gelieferte Position innerhalb der Liste der `entry`-Elemente, um zu entscheiden, ob eine neue Zeile begonnen werden soll. Dieser Test geschieht vor der Verarbeitung des eigentlichen Elementinhalts. Deshalb muss eine neue Zeile bei allen Elementen, die sich direkt nach den Vielfachen der Tabellenspalten befinden, erzeugt werden.

Allgemein ist leicht einzusehen, dass jede Variante mit `stx:process-siblings` in eine Version umgeformt werden kann, die ein umschließendes Element mittels `stx:start-element` und `stx:end-element` erzeugt. Die entgegengesetzte Aussage gilt leider nicht.

Beispiel: Gruppieren anhand der Daten

Dies zeigt sich an der Lösung des dritten Anwendungsfalls. Hier soll zum Gruppieren auf den Inhalt der einzelnen Listenelemente zurückgegriffen werden. Der Einsatz von `stx:process-siblings` ist nur dann möglich, wenn sich aus den Daten des betreffenden Elements die Gruppenbedingung direkt ableiten lässt. Bei den beiden Beispielen dieser Kategorie war das der Fall: Es handelte sich um eine Abfrage der Position bzw. des Elementtyps. Wenn jedoch zur Bestimmung des fraglichen Wertes zunächst die Verarbeitung des Inhalts (d.h. der Kindknoten) eines Elementes notwendig ist, kann `stx:process-siblings` nicht eingesetzt werden.

Als konkreter Anwendungsfall soll hier eine Liste von Personen dienen, deren Namen – und andere Daten – innerhalb von Kindelementen gespeichert sind, siehe Listing 30. Diese Daten sind also erst verfügbar, nachdem die Kinder mittels `stx:process-children` verarbeitet worden sind.

Zu gruppierende Beispieldaten **Listing 30**

```
1  <list>
2    <person>
3      <name>von Humboldt</name>
4      <vorname>Wilhelm</vorname>
5      <!-- weitere Daten -->
6    </person>
7    <person>
8      <!-- nächste Person -->
9    </person>
10   <!-- usw. -->
11 </list>
```

Diese Daten sollen nach HTML transformiert werden. Für jeden neuen Anfangsbuchstaben wird eine Zwischenüberschrift (`h3`) eingefügt. Alle Personen mit dem gleichen Anfangsbuchstaben stehen unter dieser Überschrift innerhalb der gleichen Liste (`ul`).

Listing 31 *Gruppieren mit Daten aus Unterelementen*

```
1  <stx:variable name="initial" />
2  <stx:variable name="name" />
3  <stx:variable name="vorname" />
4
5  <stx:template match="list">
6    <stx:process-children />
7    <stx:if test="$initial">
8      <stx:end-element name="ul" />
9    </stx:if>
10 </stx:template>
11
12 <stx:template match="person">
13   <stx:process-children />
14   <stx:variable name="first" select="substring($name,1,1)" />
15   <stx:if test="$first != $initial">
16     <stx:if test="$initial">
17       <stx:end-element name="ul" />
18     </stx:if>
19     <h3><stx:value-of select="$first" /></h3>
20     <stx:start-element name="ul" />
21     <stx:assign name="initial" select="$first" />
22   </stx:if>
23   <li>
24     <stx:value-of select="$name"/>, <stx:value-of select="$vorname"/>
25   </li>
26 </stx:template>
27
28 <stx:template match="person/name">
29   <stx:assign name="name" select="." />
30 </stx:template>
31
32 <stx:template match="person/vorname">
33   <stx:assign name="vorname" select="." />
34 </stx:template>
```

Der zunächst unbedeutend erscheinende Umstand, dass der Name einer Person als Unterelement modelliert ist, macht die STX-Transformation etwas umständlicher. Wären diese Informationen in Form von Attributen vorhanden, könnte auf sie direkt im Template für `person`-Elemente zugegriffen werden. So aber müssen zunächst die Kindknoten verarbeitet und die relevanten Daten in entsprechenden Variablen zwischengespeichert werden. Die beiden Templates in den Zeilen 28 und 32 dienen diesem Zweck.

Anschließend sind die Daten für Name und Vorname verfügbar und können ausgewertet werden. Wenn sich der erste Buchstabe des Namens (`$first`) vom aktuellen Buchstaben (`$initial`) unterscheidet (Zeile 15), muss eine vorherige `ul`-Liste beendet, eine neue Überschrift erzeugt und eine neue `ul`-Liste eröffnet werden. Erst dann können die eigentlichen Daten ausgegeben werden (Zeile 24).

Beim Erreichen des Endes der Personenliste muss das letzte `ul`-Element geschlossen werden. Die vorherige Abfrage des letzten Initialbuchstabens (Zeile 7) verhindert,

dass bei leeren Personenlisten ein Fehler auftritt, weil in diesem Fall kein geöffnetes `ul`-Element existiert.

Kapitel 6
Fallbeispiele

Dieses Kapitel stellt drei verschiedene Anwendungsbeispiele für STX vor. Das erste dieser Beispiele ist eher von theoretischem Interesse und zeigt, dass STX turing-vollständig ist. STX kann damit prinzipiell jede Transformationsaufgabe lösen. Die dazu angewandte Vorgehensweise lehnt sich an den Beweis der Turing-Vollständigkeit von XSLT an.

Die beiden folgenden Unterkapitel besitzen dagegen eine große Praxisrelevanz, indem sie je eine exemplarische Lösung für die in der Einleitung dieser Arbeit genannten problematischen Anwendungsfälle vorstellen. So wird im Kapitel 6.2 die Transformation eines großen XML-Dokuments am Beispiel der Daten des Open Directory gezeigt. Kapitel 6.3 demonstriert die Transformation eines XML-Datenstroms am Beispiel eines Web Service.

6.1 Simulation einer Turing-Maschine

Turing-Maschinen sind ein Mittel der theoretischen Informatik, die Mächtigkeit von Berechnungskalkülen zu bewerten. Gemäß der Church-Turing-These ist die Menge der algorithmisch berechenbaren Funktionen gleich der Menge der turing-berechenbaren Funktionen [Weg99]. Diese These ist zwar formal nicht bewiesen, sie wird jedoch allgemein als gültig anerkannt. Um für einen beliebigen Kalkül nachzuweisen, dass mit diesem alle berechenbaren Funktionen berechnet werden können, reicht es damit aus, die Gleichmächtigkeit mit der Klasse der Turing-Maschinen zu beweisen. In diesem Fall nennt man einen solchen Kalkül *turing-vollständig*. Ein solcher Beweis lässt sich beispielsweise führen, indem in dem fraglichen Kalkül eine Turing-Maschine nachgebildet wird.

Im Folgenden wird ein STX-Transformations-Sheet vorgestellt, das eine als XML kodierte Turing-Maschine simuliert. Alle Aktionen, die eine Turing-Maschine ausführen kann, werden hier durch das Transformations-Sheet vorgenommen. Damit ist STX turing-vollständig, d.h. ebenso mächtig wie eine vollwertige Programmiersprache und insbesondere so mächtig wie XSLT. Jede in XSLT programmierte Transformation kann auch in STX realisiert werden. Diese Erkenntnis besitzt jedoch keine unmittelbare praktische Bedeutung, da die Ausführung komplexer Transformationen in STX oftmals nur mit erheblich höherem Laufzeit- und Speicherplatzaufwand möglich ist.

Aufbau von Turing-Maschinen

Turing-Maschinen sind sehr einfach aufgebaut: als Datenspeicher fungiert ein ausreichend langes Speicherband, auf das ein Schreib- und Lesekopf zugreifen kann. Die Menge der möglichen Symbole auf dem Band ist begrenzt. Außerdem ist für jede Maschine eine endliche Menge von Zuständen festgelegt. Das »Programm« einer Turing-Maschine besteht aus einer Funktion, die einem Paar (Symbol, Zustand) ein Folgesymbol, einen Folgezustand und eine Bewegung des Kopfes auf dem Speicherband zuordnet. Diese Funktion kann in einer endlichen Tabelle realisiert werden, da sie einen endlich abzählbaren Definitionsbereich besitzt. In der Zustandsmenge existieren zwei ausgezeichnete Zustände für den Start und den Halt des Programmes. Zu Beginn enthält das Speicherband die Eingabe des Programmes. Am

Ende, wenn die Turing-Maschine den Haltezustand erreicht hat, enthält das Speicherband das Ergebnis der Berechnung. Ansonsten führt die Turing-Maschine für das aktuelle Symbol im aktuellen Zustand ihr »Programm« aus, d.h. sie schreibt das Folgesymbol auf die aktuelle Stelle des Bandes, sie wechselt in den Folgezustand und versetzt den Kopf um maximal eine Position nach rechts oder links.

TMML

Dieser Aufbau einer Turing-Maschine lässt sich sehr leicht in XML ausdrücken. Von Bob Lyons wurde dazu die *Turing Machine Markup Language (TMML)* [TMML] entwickelt. Das Listing 32 enthält eine Turing-Maschine, die das Einerkomplement einer auf dem Speicherband gegebenen Bitfolge berechnet.

Listing 32 *Einerkomplement mit einer Turing-Maschine*

```
<?xml version="1.0" encoding="iso-8859-1"?>
<turing-machine version="0.1">

  <!-- Symbole -->
  <symbols>01</symbols>

  <!-- Zustände -->
  <states>
    <state start="yes">start</state>
    <state>back</state>
    <state halt="yes">stop</state>
  </states>

  <!-- Das Programm -->
  <transition-function>
    <mapping>
      <from current-state="start" current-symbol="1" />
      <to next-state="start" next-symbol="0" movement="right" />
    </mapping>
    <mapping>
      <from current-state="start" current-symbol="0" />
      <to next-state="start" next-symbol="1" movement="right" />
    </mapping>
    <mapping>
      <from current-state="start" current-symbol=" " />
      <to next-state="back" next-symbol=" " movement="left" />
    </mapping>
    <mapping>
      <from current-state="back" current-symbol="1" />
      <to next-state="back" next-symbol="1" movement="left" />
    </mapping>
    <mapping>
      <from current-state="back" current-symbol="0"/>
      <to next-state="back" next-symbol="0" movement="left"/>
    </mapping>
    <mapping>
      <from current-state="back" current-symbol=" " />
      <to next-state="stop" next-symbol=" " movement="right" />
    </mapping>
  </transition-function>
</turing-machine>
```

Die wichtigsten Elemente sind im Folgenden kurz zusammengefasst:

`symbols`
: enthält die erlaubten Symbole dieser Turing-Maschine.

  Über ein Attribut `blank-symbol` kann zusätzlich ein Symbol festgelegt werden, das als Leersymbol gilt (alle leeren Felder des Speicherbandes enthalten per Definition dieses Symbol). Standardmäßig wird das Leerzeichen (Unicode 0x0020) verwendet.

`states`, **`state`**
: legen die möglichen Zustände fest. Mit Hilfe von `start`- bzw. `halt`-Attributen werden der Start- bzw. Haltezustände vereinbart.

`transition-function`, **`mapping`**, **`from`**, **`to`**
: Tabelle, die die Zustandsübergänge (das »Programm«) der Turing-Maschine beschreibt.

  Im `from`-Element sind Eingabewerte für Zustand und Symbol aufgeführt. Das `to`-Element enthält die dazugehörigen Werte für den Folgezustand, das Folgesymbol sowie eine Bewegung des Kopfes. Die möglichen Werte für `movement` sind `"left"`, `"right"` und `"none"`.

Die obige Turing-Maschine startet im Zustand *start*. Sie lässt den Kopf in diesem Zustand nach rechts wandern und dabei die vorgefundenen Bits umkehren (Zeilen 16 und 20), bis ein Leerfeld erreicht ist. Dann wechselt sie in den Zustand *back* (Zeile 24) und verschiebt den Kopf wieder an den Anfang der Eingabe zurück (Zeilen 28 und 32). Dort angekommen wechselt sie in den Zustand *stop* und hält an (Zeile 36).

Simulation in STX

Eine STX-Transformation, die eine solcherart aufgebaute XML-Datei in ein Berechnungsergebnis transformiert, benötigt zum einen eine Repräsentation des Speicherbandes und zum anderen den freien Zugriff auf die Tabelle der Zustandsübergänge. Der Inhalt des Speicherbandes lässt sich als einfache Zeichenkette darstellen. Das `transition-function`-Element wird zusammen mit der Zustandsliste `states` in einem STX-Puffer abgelegt.[1] Nun kann der Inhalt dieses Puffers solange mit Hilfe von Templates verarbeitet werden, bis der Haltezustand erreicht wurde. Diese Templates vollziehen die Arbeit der Turing-Maschine nach, indem sie Variablen für das Speicherband, für die aktuelle Position und für den aktuellen Zustand modifizieren.

Im Folgenden wird das Grundgerüst dieser STX-Transformation angegeben. Der vollständige Code ist im Anhang C.1 abgedruckt.

Zunächst sei kurz in Listing 33 das »Haupt-Template« vorgestellt, das semantisch in etwa dem Hauptprogramm in prozeduralen Programmiersprachen entspricht.

Turing-Maschine in STX **Listing 33**

```
1  <stx:transform xmlns:stx="http://stx.sourceforge.net/2002/ns"
2                 version="1.0" pass-through="none" strip-space="yes">
3
4    <!-- Anfangsbelegung des Speicherbandes -->
```

[1] Die Verwendung von STX-Puffern erleichtert an dieser Stelle lediglich die Programmierung. Es wäre ebenso möglich, die Übergangstabelle mittels Zeichenketten intern darzustellen. Damit sind nicht STX-Puffer ausschlaggebend für die Turing-Vollständigkeit, sondern die Möglichkeit, Schleifen (`stx:while`) bzw. alternativ Rekursionen (`stx:call-procedure`) programmieren zu können.

```
5   <stx:param name="tape" />
6
7   <!-- Variable, die im Haltezustand umgesetzt wird -->
8   <stx:variable name="run" select="true()" />
9
10  <!-- weitere Variablen ... -->
11
12  <stx:template match="turing-machine">
13    <!-- speichere den Inhalt des Turing-Programms in einem Puffer -->
14    <stx:buffer name="turing">
15      <stx:process-self group="store"/>
16    </stx:buffer>
17    <!-- verarbeite iterativ diesen Puffer -->
18    <stx:while test="$run">
19      <stx:process-buffer name="turing" group="process" />
20    </stx:while>
21    <!-- Ausgabe der Endbelegung des Speicherbandes -->
22    <stx:value-of select="concat('Result: ', $tape, '&#xA;')" />
23  </stx:template>
```

Die Anfangsbelegung des Speicherbandes der Turing-Maschine wird der STX-Transformation über einen externen Parameter namens `tape` übergeben (Zeile 5). Das Template für das Dokumentelement in TMML (Zeile 12) sorgt zunächst dafür, dass das Eingabedokument in einem STX-Puffer abgelegt wird (Zeile 14). Die angegebene Gruppe `store` (hier nicht abgedruckt) kopiert mittels `pass-through="all"` standardmäßig alle Knoten in die Ausgabe, d.h. in diesem Fall in den Puffer. Darüber hinaus werden die Liste erlaubter Symbole und der Startzustand in Variablen abgelegt. Nachdem der Puffer gefüllt ist, kann sein Inhalt in der Gruppe `process` solange in einer Schleife verarbeitet werden, bis durch die Variable `run` das Ende des Turing-Programms angezeigt wird (Zeile 18).

In der Gruppe `process` wird die Turing-Funktion berechnet, siehe Listing 34.

Listing 34 *Turing-Maschine in STX (Fortsetzung von Listing 33)*

```
24  <stx:group name="process">
25
26    <stx:variable name="found" />
27
28    <stx:template match="turing-machine">
29      <stx:assign name="found" select="'no'" />
30      <stx:process-children />
31    </stx:template>
32
33   <!-- Haltezustand erreicht? -->
34    <stx:template match="state">
35      <stx:if test=". = $state and @halt = 'yes'">
36        <stx:assign name="run" select="false()" />
37        <stx:assign name="found" select="'done'" />
38      </stx:if>
39    </stx:template>
40
41    <!-- aktueller Zustand und aktuelles Symbol gefunden -->
42    <stx:template match="from[$found = 'no' and
43                              @current-state = $state and
44                              @current-symbol = $symbol]">
45      <stx:assign name="found" select="'yes'" />
```

```
46      </stx:template>
47
48      <!-- Berechnung der Folgewerte -->
49      <stx:template match="to[$found='yes']">
50        <!-- neuer Zustand -->
51        <stx:assign name="state" select="@next-state" />
52        <!-- neue Speicherbandbelegung -->
53        <stx:assign name="tape" select="concat(substring($tape, 1, $pos - 1),
54                                               @next-symbol,
55                                               substring($tape, $pos + 1))" />
56        <!-- Bewegung des Kopfes -->
57        <stx:if test="@movement = 'right'">
58          <stx:assign name="pos" select="$pos + 1" />
59          <stx:assign name="symbol" select="substring($tape, $pos, 1)" />
60          <!-- Ende des Bandes erreicht? -->
61          <stx:if test="$symbol = ''">
62            <stx:assign name="symbol" select="$blank" />
63            <stx:assign name="tape" select="concat($tape, $blank)" />
64          </stx:if>
65        </stx:if>
66        <!-- analog für @movement='left' -->
67        <stx:assign name="found" select="'done'" />
68      </stx:template>
69
70    </stx:group>
71  </stx:transform>
```

Die TMML-Repräsentation der Übergangsfunktion, insbesondere die Trennung in `from`- und `to`-Elemente, erschwert deren Verarbeitung mittels STX ein wenig. So dient die Variable `found` hier einzig der Anzeige, dass ein zu den aktuellen Werten passendes `from`-Element gefunden wurde.

Zu Beginn jedes Durchlaufs wird diese Variable auf den Wert `'no'` gesetzt (Zeile 29). Bevor die eigentliche Übergangsfunktion ausgeführt wird, muss zunächst getestet werden, ob der aktuelle Zustand bereits ein Haltezustand ist (Zeile 35).

Ansonsten wird in der Übergangstabelle die Variable `found` auf den Wert `'yes'` gesetzt, wenn ein passendes `from`-Element gefunden wurde (Zeile 45). Das dann folgende `to`-Element (jenes, für das `$found='yes'` gilt, Zeile 49) bewirkt die Veränderung der entsprechenden Variablen `state`, `tape`, `symbol` und `pos`. Zum Schluss wird die Variable `found` auf den Wert `'done'` gesetzt um anzuzeigen, dass für diesen Durchlauf die Berechnung abgeschlossen ist (Zeile 67).

Wie zu sehen ist, bewirkt eine einmalige Verarbeitung des Pufferinhaltes die Berechnung eines einzelnen Schrittes der Turing-Maschine. Die Einbettung der Anweisung `stx:process-buffer` in eine `stx:while`-Schleife wie oben ausgeführt führt damit zur Ausführung des Turing-Programmes bis ein Haltezustand erreicht wird.

6.2 Verarbeitung der Daten des Open Directory

Das Open Directory Projekt (ODP)[2] bezeichnet sich selbst als den umfassendsten, von einer internationalen Gemeinschaft freiwilliger Redakteure manuell gepflegten Web-Katalog. Neue Einträge werden nach einer Begutachtung eines vorgeschlagenen Links in diesen Katalog aufgenommen. Das Projekt verfolgt keine kommerziellen Ziele mit diesem Katalog. Viele populäre Suchmaschinen greifen auf den Datenbestand des Open Directory zurück, darunter AOL Search, Netscape Search, Google, Lycos, DirectHit und HotBot.

Die kompletten ODP-Daten stehen per Download in mehreren Versionen in Form von XML-Dateien[3] zur Verfügung. So existieren beispielsweise ein XML-Dokument für die gesamte Kategorie-Struktur (*structure.rdf.u8*) sowie eines mit dem vollständigen Inhalt des Katalogs (*content.rdf.u8*). Dieses zweite Dokument enthält alle im Katalog aufgeführten Internet-Adressen und deren Beschreibungen in den jeweiligen Kategorien. Zur Repräsentation wird ein RDF-ähnliches Vokabular verwendet [RDF].

Größenverhältnisse

Die so in XML repräsentierten Daten des Open Directory belegen für XML-Verhältnisse enorme Mengen Speicherplatz:[4]

structure.rdf.u8	470.090.335 Bytes	(ca. 450 MByte)
content.rdf.u8	1.325.163.389 Bytes	(ca. 1,2 GByte)

Diese Größenordnungen sind mit XSLT oder jeder anderen XML-Applikation, die eine Gesamtansicht der Daten als internen Baum aufbaut, nicht handhabbar. Zum Vergleich: der XSLT-Prozessor Saxon [Saxon] kann unter Normalbedingungen bis zu 12 MByte an XML-Daten verarbeiten (siehe Kapitel 4.4). Selbst die Vergrößerung des maximalen Speicherplatzes auf den für Java größtmöglichen Wert (knapp 4GB) reicht für die Datei *content.rdf.u8* nicht aus.

Eine Verarbeitung der ODP-Daten mit STX setzt eine geeignete, d.h. sequentielle Struktur dieser Daten voraus. Diese liegt hier vor. Im Folgenden werden beispielhaft die Daten der Datei *content.rdf.u8* ausgewertet. Der Aufbau dieser Datei wird im Listing 35 dargestellt.

Listing 35 *Struktur der ODP-Daten in content.rdf.u8*

```
<?xml version='1.0' encoding='UTF-8'?>
<RDF xmlns:r="http://www.w3.org/TR/RDF/"
     xmlns:d="http://purl.org/dc/elements/1.0/"
     xmlns="http://dmoz.org/rdf">

<Topic r:id="Top">
  <catid>1</catid>
```

[2] Informationen über das ODP-Projekt finden sich unter der Adresse http://www.dmoz.org/about.html. Der Domainname DMOZ steht für *Directory Mozilla* und reflektiert dessen lockere Verbindung zum Mozilla-Projekt (siehe http://mozilla.org/).

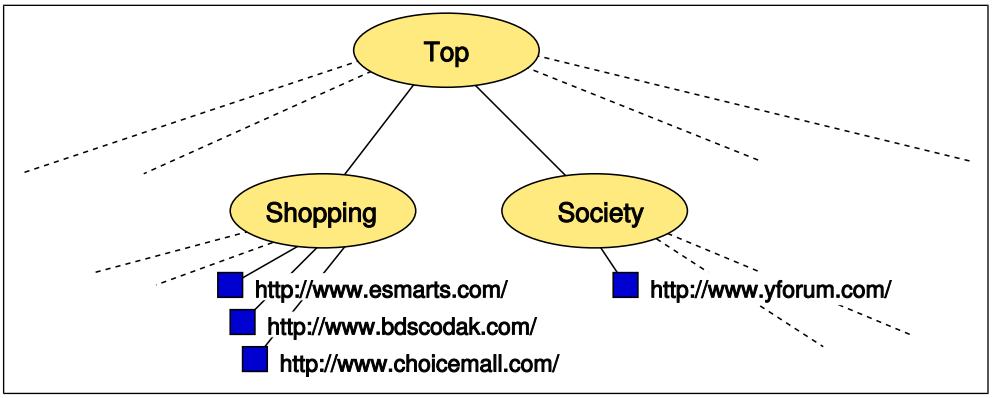
[3] Leider enthalten diese XML-Dateien derzeit noch Kodierungsfehler. Einige Bytefolgen entsprechen keiner gültigen UTF8-Sequenz und damit keinem definierten Unicode-Zeichen. Ein XML-Parser wird an diesen Stellen beim Einlesen einen Fehler melden. Die ODP-Betreuer arbeiten daran, solche Fehler zukünftig zu verhindern. Die provisorische Lösung in der vorliegenden Arbeit besteht darin, die Daten vor der XML-Verarbeitung durch ein spezielles Filterprogramm bearbeiten zu lassen, das die fehlerhaften Bytes entfernt.

[4] Diese Größenangaben stammen vom Oktober 2003.

```
</Topic>
...
<Topic r:id="Top/Shopping">
  <catid>13<catid>
  <link r:resource="http://www.esmarts.com/"/>
  <link r:resource="http://www.bdscodak.com"/>
  <link r:resource="http://www.choicemall.com/"/>
</Topic>

<ExternalPage about="http://www.esmarts.com/">
  <d:Title>eSmarts</d:Title>
  <d:Description>
    eSmarts helps consumers find the lowest possible prices on the web.  They
    compare prices at different Internet stores, list coupons (including many
    $10 off coupons), discuss sales and share great shopping tips.
  </d:Description>
</ExternalPage>

<ExternalPage about="http://www.bdscodak.com">
  <d:Title>
    BD Scodak - personalized children's books for your child's education
  </d:Title>
  <d:Description>
    BD Scodak is your source for personalized children's books customized with
    your child's information right next to popular cartoon, religious, sports,
    and tv characters and themes.
  </d:Description>
</ExternalPage>

<ExternalPage about="http://www.choicemall.com/">
  <d:Title>Choice World</d:Title>
  <d:Description>
    Choice Mall - The #1 global marketplace on the Internet.  Thousands of
    stores offer quality, unique products and services, art and entertainment,
    books and music, gifts, food, real estate, health, sports, and fitness --
    all under one roof!
  </d:Description>
</ExternalPage>

<Topic r:id="Top/Society">
  <catid>14</catid>
  <link r:resource="http://www.yforum.com/"/>
</Topic>

<ExternalPage about="http://www.yforum.com/">
  <d:Title>Y? The National Forum on People's Differences</d:Title>
  <d:Description>
    The nation's only forum allowing people to ask and receive answers to the
    uncomfortable and even embarrassing questions they've always wanted to ask
    people who are different from themselves.  All questions and answers are
    acceptable, as long as they promote dialogue and are not asked or answered
    out of hate.
  </d:Description>
</ExternalPage>

<!-- weitere Topic- und ExternalPage-Elemente -->
...
</RDF>
```

Die hier auszugsweise abgedruckten Daten enthalten drei Einträge für die Kategorie `Top/Shopping` und einen Eintrag für die Kategorie `Top/Society`. Beide gehören zu übergeordneten Kategorie `Top`. Abbildung 9 veranschaulicht diesen Aufbau.

Abbildung 9
ODP-Baum



Auf diese Weise wird der gesamte Inhalt des Open Directory in XML beschrieben. Sein hierarchischer Aufbau kann mit einem Dateisystem verglichen werden: die Elemente `Topic` entsprechen den Verzeichnissen und die Elemente `ExternalPage` entsprechen den Dateien in diesen Verzeichnissen.

Wird der so aufgebaute Verzeichnisbaum in einer Tiefe-zuerst-Suche durchlaufen und jeder besuchte Knoten in Form eines XML-Elementes protokolliert, ergibt sich der im Listing 35 dargestellte XML-Inhalt in Form einer langen und flachen XML-Struktur.

ODP-Elemente

Im Folgenden werden die enthaltenen Elemente kurz charakterisiert:

`RDF`
: ist das Wurzelelement, das als Container für die Liste der anderen Elemente dient und alle benötigten Namensräume deklariert.

`Topic`
: repräsentiert eine Kategorie. Das `r:id`-Attribut enthält den vollständigen Pfad innerhalb der Kategoriehierarchie. In der Datei *content.rdf.u8* ist diese Angabe die einzige Quelle für die Rekonstruktion der Verzeichnisstruktur.

`catid`
: enthält eine eindeutige Identifikationsnummer der Kategorie.

`link`
: verweist auf eine zu dieser Kategorie gehörenden Web-Ressource. Das Attribut `r:resource` enthält die Adresse (URL) dieser Ressource und kennzeichnet sie dadurch eindeutig.

`ExternalPage`
: enthält zusätzliche Informationen zu einer Web-Ressource, die zuvor in einem `link`-Element aufgelistet wurde. Über das `about`-Attribut ist festgelegt, um welche Ressource es sich handelt.

`d:Title`, **`d:Description`**
: werden als Kindelemente von `ExternalPage` verwendet und enthalten den Titel bzw. eine Beschreibung der jeweiligen Ressource. Diese Elemente stammen aus dem Metadaten-Vokabular Dublin Core [DCMI03].

Mit Hilfe einer STX-Transformation lassen sich aus diesen Daten einzelne XHTML-Seiten generieren, die der Web-Schnittstelle des Open Directory entsprechen.

Transformation in XHTML

Dabei wird für jede Kategorie eine eigene XHTML-Seite erzeugt, die zum einen alle enthaltenen Ressourcen inklusive der in `d:Title` und `d:Description` angegebenen Informationen auflistet und zum anderen Verweise auf die jeweiligen Unterkategorien enthält. Mit STX kann auf diese Weise nicht nur eine lokale, mit einem WWW-Browser navigierbare Sicht auf die ODP-Daten erstellt werden, diese Vorgehensweise erlaubt zudem die Anpassung an ein eigenes Layout, die Einbindung zusätzlicher Informationen bzw. Links, das Ausblenden unerwünschter Ressourcen und sonstige Verarbeitungsschritte.

Diese Aufgabe ist vom Wesen her ein Gruppierungsproblem, wie sie bereits im Kapitel 5.7.6 beschrieben wurden. In diesem Fall beginnt eine Gruppe bei einem `Topic`-Element und endet beim ersten `Topic`-Element, dessen `r:id`-Attribut keine Unterkategorie der aktuellen Gruppe bezeichnet. Unterkategorien lassen sich hier leicht daran erkennen, dass der entsprechende Pfad im `r:id`-Attribut mit dem Pfad der aktuellen Kategorie beginnt.

So beginnt beispielsweise der Pfad `Top/Shopping` mit dem Pfad `Top` und kennzeichnet so eine Unterkategorie von `Top`. Dagegen ist `Top/Society` keine Unterkategorie von `Top/Shopping`.

Da die XML-Quelle in einer Tiefe-zuerst-Suche erstellt wurde, ist sichergestellt, dass die enthaltenen Ressourcen und Unterkategorien unmittelbar nach dem übergeordneten `Topic`-Element aufgelistet werden. Wird der Beginn einer neuen gleichrangigen Kategorie festgestellt, kann die XHTML-Repräsentation der aktuellen Kategorie abgeschlossen werden. Der Fall, dass später noch weitere Einträge oder gar neue Unterkategorien in den Daten erscheinen, kann nicht auftreten. Diese Eigenschaft ist essenziell für die Bearbeitung mit STX.

Das folgende Listing 36 zeigt das etwas vereinfachte Grundgerüst der besprochenen Transformation. Der vollständige Quelltext ist im Anhang C.2 abgedruckt.

STX-Transformation der ODP-Daten **Listing 36**

```
<stx:transform xmlns:stx="http://stx.sourceforge.net/2002/ns"
               xmlns:r="http://www.w3.org/TR/RDF/"
               xmlns:d="http://purl.org/dc/elements/1.0/"
               xmlns:od="http://dmoz.org/rdf"
               xmlns="http://www.w3.org/1999/xhtml"
               version="1.0" strip-space="yes"
               exclude-result-prefixes="#all">

  <stx:template match="od:RDF"> .... </stx:template>

  <stx:template match="od:Topic">
    <stx:param name="base" />
    <stx:param name="parent" select="'.'" />
    <stx:variable name="filename" select="substring-after(@r:id, $base)" />
    ...
    <li>
      <a href="{$parent}/{$filename}.html">
        <stx:value-of select="$filename" />
      </a>
    </li>
    <stx:result-document href="{concat(@r:id, '.html')}">
```

```
22        <html>
23          ...
24          <dl>
25            <stx:process-siblings while="od:ExternalPage" />
26          </dl>
27          ...
28          <stx:variable name="id" select="concat(@r:id, '/')" />
29          <ul>
30            <stx:process-siblings until="od:Topic[not(starts-with(@r:id, $id))]">
31              <stx:with-param name="base" select="$id" />
32              <stx:with-param name="parent" select="$filename" />
33            </stx:process-siblings>
34          </ul>
35          ...
36        </html>
37      </stx:result-document>
38    </stx:template>
39
40    <stx:template match="d:Title">
41      <dt>
42        <strong>
43          <a href="{escape-uri(../@about, false())}" target="_blank">
44            <stx:value-of select="." />
45          </a>
46        </strong>
47      </dt>
48    </stx:template>
49
50    <stx:template match="d:Description">
51      <dd>
52        <stx:value-of select="." />
53      </dd>
54    </stx:template>
55
56  </stx:transform>
```

Für jede neue Kategorie, d.h. für jedes `Topic`-Element wird eine separate Ausgabedatei erzeugt (Zeile 21). Als Dateiname wird hier der Einfachheit halber der um die Endung `".html"` erweiterte Pfad im `r:id`-Attribut verwendet. Die vollständige Version des Transformations-Sheet verhindert an dieser Stelle das Entstehen von Dateinamen, die möglicherweise lokal nicht darstellbare Unicode-Zeichen enthalten. Diese neue Datei enthält zunächst alle direkt in dieser Kategorie enthaltenen Ressourcen innerhalb einer `dl`-Liste. Dazu werden alle `ExternalPage`-Elemente verarbeitet, die unmittelbar auf das aktuelle `Topic`-Element folgen (Zeile 25). Die Ausgabe des verlinkten Titels und der Beschreibung der Ressource leisten die Templates in den Zeilen 40 und 50.

Anschließend werden alle zugehörigen Unterkategorien in einer `ul`-Liste aufgeführt.[5] Wie oben am Beispiel erläutert wurde, sind Unterkategorien daran zu erkennen, dass sie mit dem gleichen Pfad beginnen, wie die aktuelle Kategorie. Mit Hilfe der Funktion `starts-with` kann diese Bedingung überprüft werden. In der Transformation

[5] Hier würde für Kategorien, die keine weiteren Unterkategorien besitzen, ein leeres `ul`-Element erzeugt. Da dies der XHTML-DTD widerspricht, enthält die im Anhang C.2 abgedruckte vollständige Version diesen Fehler nicht mehr.

werden damit alle folgenden Geschwister verarbeitet, bis ein `Topic`-Element auftritt, das dieser Bedingung nicht mehr genügt (Zeile 30).

Die Zeilen 16 bis 20 sorgen schließlich dafür, dass in dem aktuell zum Schreiben geöffneten XHTML-Dokument auch Verweise auf die neuen Unterkategorien erzeugt werden.

Die Parameter in den Zeilen 12 bzw. 31 erleichtern das Erzeugen relativer Verweise zwischen den erzeugten Dateien.

Bewertung

Die Struktur der Ausgangsdaten ermöglicht eine sehr gute Verarbeitung mit STX. Insbesondere die korrekte Reihenfolge der `Topic`- und `ExternalPage`-Elemente erweist sich als unabdingbare Voraussetzung. Der Speicherbedarf ist für diese Daten konstant: die XML-Quelle besitzt eine Maximaltiefe von 3 Element-Ebenen, die enthaltenen Textknoten sind vergleichsweise kurz.

Im Rahmen dieser Arbeit wurde die Verarbeitungsgeschwindigkeit für diese Transformation experimentell bestimmt. Auf einer Sun Blade 1000 (600 MHz, 1024 MB RAM) unter SunOS 5.8 mit Java 1.4.1 ohne weitere Last ergaben sich folgende Messwerte für die Verarbeitung der Datei *content.rdf.u8*[6] mit der im Anhang C.2 gegebenen STX-Transformation:

Zeitaufwand für das Generieren der Verzeichnisstruktur (Gesamtzeit der STX-Transformation)	2:56 Stunden
Anzahl generierter Dateien	581.172
Anzahl generierter Verzeichnisse	137.913
Größe der entstandenen Verzeichnisstruktur	2,4 GByte
Zeitaufwand für das Löschen der Verzeichnisstruktur (mittels `rm -rf`)	1:11 Stunden

Hier wird deutlich, dass ein Prozess zum Erzeugen der Verzeichnisstruktur mit knapp 3 Stunden Rechenzeit problemlos als Batch-Job beispielsweise in der Nacht laufen kann. Die generierten Daten belegen insgesamt doppelt so viel Speicherplatz wie die Originaldatei. Dies liegt im Wesentlichen am HTML-Rahmen jeder Seite und dem in jeder Datei enthaltenen Verweis auf das ODP, der laut den Lizenzbedingungen enthalten sein muss.

Das Löschen der Verzeichnisstruktur benötigt etwa 40% der Zeit, die zum Generieren aufgebracht wird. Anders ausgedrückt: mindestens 40% der Rechenzeit sind auf Dateizugriffe zurückzuführen, die unabhängig von der eigentlich Transformation sind. Allein das Bestimmen der Verzeichnisgröße und das Zählen der Dateien und Verzeichnisse benötigt gut 45 Minuten. Durch Optimierungen im STX-Prozessor wird man daher die Gesamtrechenzeit höchstens an diese 40%-Grenze annähern können.

[6] Aufgrund der in der Originaldatei enthaltenen Kodierungsfehler wurde eine korrigierte Version verwendet, die 3065 Byte, d.h. vernachlässigbare 0,23‰ kürzer ist.

6.3 Web Services am Beispiel Google

Web Services sind eine noch recht junge Technologie. Mit ihnen lassen sich verteilte Applikationen auf der Basis bewährter Web-Techniken realisieren. Ein Web Service lässt sich kurz durch die folgenden Eigenschaften charakterisieren:

- Es wird ein Dienst öffentlich auf einem Server zur Verfügung gestellt.
- Dieser Dienst verfügt über eine in WSDL (*Web Service Definition Language*) beschriebene Schnittstelle.
- Über Internet-Protokolle kann auf diesen Dienst unter Benutzung von SOAP zugegriffen werden.
- Die übertragenen Daten werden in XML kodiert.

Ein Web Service ist in der Regel »nur« eine Fassade, d.h. eine neuartige Schnittstelle für bereits bestehende Systeme. Eine Anfrage an einen Web Service wird als XML-Dokument an den Server geschickt. Dieser liefert das Ergebnis auf diese Anfrage ebenfalls in XML. Das Format der ausgetauschten XML-Daten wird dabei durch SOAP festgelegt.

Web Service APIs

Für Web Services existieren eine Reihe von Werkzeugen (bzw. APIs), die die Details der übertragenen XML-Daten vor dem Programmierer verbergen. Diese APIs ermöglichen die Anwendung vertrauter Paradigmen (z.B. entfernte Methodenaufrufe) in der jeweils benutzten Programmiersprache und übernehmen die Umwandlung der Daten von und nach XML (das so genannte *Marshalling/Unmarshalling*). Web Services und die dazugehörigen Werkzeuge entwickeln sich damit zu einer leichtgewichtigen Ergänzung ausgewachsener Middleware-Plattformen wie CORBA oder DCOM. Derzeit existieren allerdings noch keine einheitlichen Sprach-Mappings (siehe [Bro03]), sodass hier weitere Standardisierungen folgen müssen.

Transformation von SOAP-Antworten

Betrachtet man den Fall, dass die von einem Web Service gelieferten Daten in ein eigenes XML-Format umgewandelt werden sollen, so bietet es sich an, dies auf dem Weg einer XML-Transformation zu erreichen. In diesem Fall kann der Prozess des *Unmarshalling* entfallen, der die Daten des ankommenden XML-Datenstroms in Datenstrukturen der verwendeten Programmiersprache überführt. Ebenso entfällt der zusätzliche Aufwand, diese eigenen Datenstrukturen anschließend wieder in einem speziellen XML-Format zu serialisieren. Mehr noch: soll nur ein Teil der durch einen Web Service gelieferten Informationen als XML weiterverarbeitet werden, so kann dies in einem seriellen Transformationsprozess weitaus effizienter geschehen.

Im Folgenden soll der Web Service der Suchmaschine Google [Google] verwendet werden. Dieser ermöglicht neben dem bekannten Suchen nach Web-Ressourcen auch die Abfrage gespeicherter (*cached*) Seiten sowie die Benutzung einer Rechtschreibkorrektur. Für dieses Fallbeispiel besteht die Aufgabe darin, die von Google gelieferten Suchergebnisse in kompakter Form in eine dynamisch generierte Web-Seite einzufügen. Dabei sind nicht alle mitgeschickten Details von Interesse, sondern nur der Titel und der URL jeder gefundenen Web-Ressource.

SOAP-Anfrage

Eine Suchanfrage an den Google-Web-Service besitzt den in Listing 37 dargestellten Aufbau.

Eine Suchanfrage per SOAP an Google **Listing 37**

```
1  <?xml version='1.0' encoding='UTF-8'?>
2  <SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
3                     xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
4                     xmlns:xsd="http://www.w3.org/1999/XMLSchema">
5    <SOAP-ENV:Body>
6      <ns1:doGoogleSearch xmlns:ns1="urn:GoogleSearch"
7            SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
8        <key xsi:type="xsd:string">XXX</key>
9        <q xsi:type="xsd:string">Joost STX</q>
10       <start xsi:type="xsd:int">0</start>
11       <maxResults xsi:type="xsd:int">10</maxResults>
12       <filter xsi:type="xsd:boolean">true</filter>
13       <restrict xsi:type="xsd:string"/>
14       <safeSearch xsi:type="xsd:boolean">true</safeSearch>
15       <lr xsi:type="xsd:string"/>
16       <ie xsi:type="xsd:string">UTF-8</ie>
17       <oe xsi:type="xsd:string">UTF-8</oe>
18     </ns1:doGoogleSearch>
19   </SOAP-ENV:Body>
20 </SOAP-ENV:Envelope>
```

Der von SOAP definierte »Umschlag« (`Envelope`) enthält im Körper (`Body`) die konkreten Daten der Anfrage, in diesem Fall einen Aufruf der Methode `doGoogleSearch`. In deren Inneren sind die einzelnen Parameter des Methodenaufrufs kodiert. Der Inhalt des in Zeile 8 stehenden Elements `key` muss dabei durch einen persönlichen Schlüssel ersetzt werden, den man von Google nach einer Registrierung erhält. Das Element `q` in Zeile 9 enthält den jeweiligen Suchstring (hier "Joost STX"). Darüber hinaus wird festgelegt, wieviele Ergebnisse (`maxResult`[7]) ab welchem Index (`start`) zurückgegeben werden sollen, und dass mögliche Duplikate (`filter`) sowie nicht jugendfreie Einträge (`safeSearch`) entfernt werden sollen. Eine Einschränkung auf Teile des Google-Index (`restrict`) oder Länder bzw. Sprachen (`lr`) soll nicht stattfinden. Die letzten beiden Parameter `ie` und `oe` sind inzwischen veraltet und werden vom Google-Web-Service ignoriert. Sie müssen aber angegeben werden, da die Signatur der Methode `doGoogleSearch` noch nicht entsprechend angepasst wurde. In dem hier besprochenen Anwendungsfall bleiben alle Einträge bis auf `q` in jeder Anfrage unverändert.

Wird ein solches Dokument an den Google-Web-Service geschickt, wird damit der dazugehörige Suchdienst aufgerufen. In diesem konkreten Fall muss die Anfrage mit der HTTP-Methode POST [IETF99] an die zur derzeitigen Beta-Version gehörenden Adresse http://api.google.com/search/beta2 gesendet werden. Das im Ergebnis dieses Aufrufs gelieferte XML-Dokument kann anschließend durch STX-Transformationsregeln verarbeitet werden.

Filtermethode HTTP-POST

Den Versand von XML-Daten mittels HTTP-POST unterstützt *Joost* über die externe Filtermethode `http://www.ietf.org/rfc/rfc2616.txt#POST`. In diesem Fall wird damit die Übertragung von Daten an einen HTTP-Server und die Entgegen-

[7] Leider akzeptiert Google in der derzeitigen Beta-Version für das Element `maxResults` nur Werte bis einschließlich 10. Eine Suchanfrage, die einen längeren Datenstrom zurückliefert, ist mit Google auf diese Weise derzeit nicht realisierbar. Prinzipiell lässt sich die hier vorgestellte Vorgehensweise jedoch für Suchmaschinen oder Datenbanken mit XML-Schnittstelle anwenden, die beliebig große Resultate als XML liefern.

nahme einer Antwort als Filter bzw. Transformation verstanden. Das Anfrage-XML-Dokument aus Listing 37 wird in ein Antwort-XML-Dokument »transformiert«. Der Ziel-URL wird hier über einen Parameter an den Filter übergeben und ermöglicht so den Zugriff auf beliebige Web Services. In STX lässt sich somit – anders als in XSLT – ein standardisierter Weg für die Benutzung der HTTP-Methode POST beschreiben.

Der in Listing 38 dargestellte Ausschnitt aus dem STX-Transformations-Sheet dieser Anwendung demonstriert die beschriebene Vorgehensweise.

Listing 38 *Versenden einer SOAP-Anfrage mittels HTTP-POST*

```
1  <stx:param name="search" />
2
3  <stx:buffer name="request">
4    <SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
5                       xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
6                       xmlns:xsd="http://www.w3.org/1999/XMLSchema">
7      <SOAP-ENV:Body>
8        <ns1:doGoogleSearch xmlns:ns1="urn:GoogleSearch"
9              SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
10
11         <!-- Hier muss der key eingetragen werden ... -->
12
13         <q xsi:type="xsd:string">
14           <stx:value-of select="$search" />
15         </q>
16
17         <!-- Hier erscheinen die restlichen Parameter ... -->
18
19       </ns1:doGoogleSearch>
20     </SOAP-ENV:Body>
21   </SOAP-ENV:Envelope>
22 </stx:buffer>
23
24 <stx:template match="/">
25   <stx:process-buffer name="request"
26                       filter-method="http://www.ietf.org/rfc/rfc2616.txt#POST">
27     <stx:with-param name="target"
28                     select="'http://api.google.com/search/beta2'" />
29   </stx:process-buffer>
30 </stx:template>
```

Die SOAP-Anfrage ist hier in einem STX-Puffer mit dem Namen `request` abgelegt. Der aktuelle Suchstring wird dabei dynamisch aus dem globalen Parameter `search` in das Element `q` eingefügt (siehe Zeile 14). Anschließend wird mit Hilfe der Anweisung `stx:process-buffer` (Zeile 25) und der bereits genannten Filtermethode `http://www.ietf.org/rfc/rfc2616.txt#POST` der Puffer-Inhalt an den durch den Parameter `target` (Zeile 27) adressierten Web Service geschickt .

SOAP-Antwort

Das daraufhin von Google als Antwort gelieferte XML-Dokument ist in Listing 39 auszugsweise angegeben:

Listing 39 *Ergebnis der Anfrage aus Listing 37 auf Seite 131*

```
1  <?xml version='1.0' encoding='UTF-8'?>
2  <SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
```

```
3                     xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
4                     xmlns:xsd="http://www.w3.org/1999/XMLSchema">
5  <SOAP-ENV:Body>
6  <ns1:doGoogleSearchResponse xmlns:ns1="urn:GoogleSearch"
7         SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
8  <return xsi:type="ns1:GoogleSearchResult">
9  <documentFiltering xsi:type="xsd:boolean">true</documentFiltering>
10 <estimatedTotalResultsCount xsi:type="xsd:int">137</estimatedTotalResultsCount>
11 <directoryCategories xmlns:ns2="http://schemas.xmlsoap.org/soap/encoding/"
12                       xsi:type="ns2:Array"
13                       ns2:arrayType="ns1:DirectoryCategory[0]">
14 </directoryCategories>
15 <searchTime xsi:type="xsd:double">0.189574</searchTime>
16 <resultElements xmlns:ns3="http://schemas.xmlsoap.org/soap/encoding/"
17                 xsi:type="ns3:Array" ns3:arrayType="ns1:ResultElement[10]">
18 <item xsi:type="ns1:ResultElement">
19 <cachedSize xsi:type="xsd:string">37k</cachedSize>
20 <hostName xsi:type="xsd:string"></hostName>
21 <snippet xsi:type="xsd:string"> &lt;b&gt;...&lt;/b&gt; Project:
22 &lt;b&gt;Joost&lt;/b&gt; &lt;b&gt;STX&lt;/b&gt; processor: Summary.
23 &lt;b&gt;...&lt;/b&gt; &lt;b&gt;Joost&lt;/b&gt; is a Java implementation of the
24 Streaming&lt;br&gt; Transformation for XML (&lt;b&gt;STX&lt;/b&gt;) language
25 (see http://&lt;b&gt;stx&lt;/b&gt;.sourceforge.net/). &lt;b&gt;...&lt;/b&gt;
26 </snippet>
27 <directoryCategory xsi:type="ns1:DirectoryCategory">
28 <specialEncoding xsi:type="xsd:string"></specialEncoding>
29 <fullViewableName xsi:type="xsd:string"></fullViewableName>
30 </directoryCategory>
31 <relatedInformationPresent xsi:type="xsd:boolean">true</relatedInformationPresent>
32 <directoryTitle xsi:type="xsd:string"></directoryTitle>
33 <summary xsi:type="xsd:string"></summary>
34 <URL xsi:type="xsd:string">https://sourceforge.net/projects/joost</URL>
35 <title xsi:type="xsd:string">SourceForge.net: Project Info -
36 &lt;b&gt;Joost&lt;/b&gt; &lt;b&gt;STX&lt;/b&gt; processor</title>
37 </item>
38
39 <!-- 9 weitere item-Elemente ... -->
40
41 </resultElements>
42 <endIndex xsi:type="xsd:int">10</endIndex>
43 <searchTips xsi:type="xsd:string"></searchTips>
44 <searchComments xsi:type="xsd:string"></searchComments>
45 <startIndex xsi:type="xsd:int">1</startIndex>
46 <estimateIsExact xsi:type="xsd:boolean">false</estimateIsExact>
47 <searchQuery xsi:type="xsd:string">Joost STX</searchQuery>
48 </return>
49 </ns1:doGoogleSearchResponse>
50
51 </SOAP-ENV:Body>
52 </SOAP-ENV:Envelope>
```

Diese SOAP-Antwort gleicht im Grundaufbau der SOAP-Anfrage: Innerhalb von `Envelope` und `Body` steht das Element `doGoogleSearchResponse`, das alle Daten des Resultats enthält. Für die Bedeutung der einzelnen Einträge sei an dieser Stelle auf die Dokumentation unter http://www.google.com/apis/reference.html verwiesen. Aus der Fülle der übermittelten Daten interessieren für diesen Anwendungsfall nur zwei Details der gefundenen Ressourcen: der URL und der Titel. Beide Angaben stehen in entsprechenden Elementen `URL` (Zeile 34) und `title` (Zeile 35) und um-

fassen nur etwa 18% der Gesamtdaten pro gefundener Ressource. Aufgabe der STX-Transformation ist es, eine einfache HTML-Liste (`ul`) mit den gefundenen Links zu erzeugen.

Der Listenkörper wird aus dem Element `resultElements` (Zeile 16) abgeleitet. Jedes `item`-Element (Zeile 18) repräsentiert eine gefundene Ressource und wird in ein `li`-Element transformiert. Hier bietet es sich zunächst an, die Angaben zu URL und Titel für jedes `item`-Element in Variablen zu speichern. Nachdem das Ende eines `item`-Elements erreicht wurde, kann der Link als Listenpunkt erzeugt werden.

Umwandlung von HTML-Markup

Als problematisch stellt sich hier die Verarbeitung des Titels heraus. Google kennzeichnet die im Titel gefundenen Suchbegriffe durch HTML-Markup (in diesem Fall durch Einschluss in `b`-Tags). Dieses Markup ist allerdings als Text kodiert (vergleiche Zeile 35):

```
SourceForge.net: Project Info - &lt;b&gt;Joost&lt;/b&gt;
&lt;b&gt;STX&lt;/b&gt; processor
```

Aus XML-Sicht handelt es sich um reinen Text, der in der Ausgabe wieder geeignet maskiert werden muss. Ohne weitere Vorkehrungen würde im Ergebnis ein HTML-Browser das HTML-Markup ebenfalls als Text anzeigen:

```
SourceForge.net: Project Info - <b>Joost</b> <b>STX</b>
processor
```

Gewünscht ist jedoch die Interpretation des HTML-Markup, sodass im Browser folgendes zu lesen ist:

```
SourceForge.net: Project Info - Joost STX processor
```

Damit das im Text enthaltene Markup als solches erkannt wird, muss der Text analysiert (geparst) werden. Der Prozessor *Joost* unterstützt diesen Anwendungsfall durch die Bereitstellung einer weiteren externen Filtermethode `http://xml.org/sax`, die die gelieferten Textdaten analysiert und in SAX-Events umwandelt. Da ein SAX-Parser nur wohlgeformte XML-Dokumente verarbeiten kann, muss der zu analysierende Text des `title`-Elements in ein zusätzliches Dokumentelement (in diesem Fall `span`) eingeschlossen werden. Dies geschieht, indem der gesamte Text zunächst in einen STX-Puffer geschrieben wird (siehe Zeile 9 in Listing 40), dessen Inhalt dann durch den SAX-Parser (Zeile 16) verarbeitet wird.

Das Template, das die `item`-Elemente unterhalb von `resultElements` verarbeitet, ist in Listing 40 angegeben.

Listing 40 *STX-Template zur Bearbeitung der Google-Ergebnisse*

```
1  <stx:variable name="url" />
2  <stx:variable name="title" />
3
4  <stx:template match="resultElements/item">
5    <!-- Belegung der Variablen url und title -->
6    <stx:process-children />
7
8    <!-- Anlegen eines Puffers mit dem zu parsenden Text -->
9    <stx:buffer name="text">
10     &lt;span&gt;<stx:value-of select="$title" />&lt;/span&gt;
11   </stx:buffer>
12
13   <!-- Erzeugen eines Listeneintrages -->
14   <li>
15     <a href="{$url}">
```

```
16        <stx:process-buffer name="text" filter-method="http://xml.org/sax" />
17      </a>
18    </li>
19  </stx:template>
```

Dieses Beispiel demonstriert, dass der Einsatz von STX für den Zugriff auf Web Services mit der gegenwärtigen Funktionsweise zwei separate Transformationsschritte benötigt:

1. Übermittlung der SOAP-Anfrage und Entgegennahme des SOAP-Ergebnisses
2. Transformation des SOAP-Ergebnisses in das gewünschte Zielformat

Pipeline-Verarbeitung

Beide Schritte können jedoch direkt nacheinander in einer STX-Pipeline ausgeführt werden. Der Ergebnis-Datenstrom aus dem ersten Schritt wird unmittelbar im zweiten Schritt verarbeitet. Der für diese Anwendungsfälle eingesetzte STX-Prozessor *Joost* unterstützt dieses Szenario durch einen Aufruf auf der Kommandozeile, indem mehrere Transformations-Sheets angegeben werden können. In diesem Beispielfall würde der Aufruf folgendermaßen lauten:

```
joost empty.xml google-request.stx search="Joost STX" google2html.stx
```

Die Datei *empty.xml* ist hier eine minimale (leere) XML-Datei, die aus technischen Gründen als formale Eingabe dient. Das Transformations-Sheet *google-request.stx* führt die Anfrage an Google mit dem in `search` angegebenen Suchstring durch, *google2html.stx* übernimmt die anschließende Formatierung als HTML. Beide Transformations-Sheets sind vollständig im Anhang C.3 abgedruckt.

Für eine prozessorunabhängige Lösung muss es jedoch möglich sein, die Ausgabe eines externen Filters durch die Templates des gleichen Transformations-Sheet verarbeiten zu lassen. Die Konstruktion einer Pipeline außerhalb von STX kann damit entfallen. Am Ende von Kapitel 5.6.7 wurde die dazu erforderliche Änderung an STX bereits kurz skizziert.

Bewertung

Die von einem Web Service gelieferten SOAP-Antworten lassen sich sehr gut durch STX verarbeiten. Insbesondere kann hier aus dem seriellen Charakter von STX großer Nutzen gezogen werden:

- Die XML-Daten werden von einem entfernten Server geliefert. Die Transformation kann sofort beginnen, selbst wenn es zu Verzögerungen beim Transport kommen sollte. Eintreffende Daten werden unmittelbar transformiert.
- Nur ein Bruchteil der gelieferten Daten ist für das gewünschte Resultat relevant. In diesem Fall sollten allein Titel und URL der gefundenen Ressourcen und damit weniger als 20% der Ergebnisdaten weiterverarbeitet werden. STX arbeitet hier sehr viel speichereffizienter als XSLT oder auch als typische Web-Service-APIs, die alle enthaltenen Daten einlesen und im Speicher repräsentieren müssten.
- Es sind Web Services denkbar, die als Antwort beliebig große Datenmengen liefern. Im Falle von Google wäre dies bei einer Aufhebung der Beschränkung auf 10 Ergebnisse pro Suchanfrage der Fall. XSLT oder Web-Service-APIs können jeweils nur eine begrenzte Datenmenge verarbeiten und stoßen bei sehr langen Datenströmen an ihre Grenzen.

Kapitel 7
STX-Integration

STX ist wie XSLT eine Sprache, die keine Aussagen darüber trifft, auf welche Weise die beschriebene Transformation aufgerufen und ausgeführt wird. Eine Software, die ein XML-Dokument mit Hilfe eines STX-Transformations-Sheet in ein anderes Dokument umwandelt (im folgenden *STX-Prozessor* genannt), kann und muss selbst eine geeignete Aufrufschnittstelle festlegen.

Bei XSLT führte dies sowohl zur Entwicklung von eigenständigen Applikationen (z.B. *Instant Saxon*) als auch zu Code-Bibliotheken, die durch andere Applikationen genutzt werden (z.B. *msxml.dll* im *MS Internet Explorer*). In den meisten Fällen findet man beide Eigenschaften in einem Produkt: eine Bibliothek, deren interne Schnittstelle durch eine Kommandozeilen-Applikation benutzt wird und die damit ebenfalls als eigenständiges Programm erscheint.

Joost

Als prototypische Implementierung entstand im Ergebnis dieser Dissertation der STX-Prozessor *Joost*.[1] Dieser wurde als Open-Source-Projekt auf *SourceForge* veröffentlicht [Joost]. Während der vergangenen zwei Jahre ist die Zahl der *Joost*-Nutzer auf über 130 angewachsen.[2] *Joost* ist in Java programmiert und sowohl als eigenständige Applikation, als auch über interne Java-Schnittstellen verwendbar. Insbesondere diese Schnittstellen erlauben es, *Joost* in fremde Java-Applikationen einzubetten, dort STX-Transformationen auf internen XML-Daten auszuführen und sogar Java-Code aus STX heraus aufzurufen.

Im Folgenden werden einige der damit verbundenen Aspekte im Detail dargestellt.

7.1 SAX-Filter

Als *Streaming Transformer* basiert *Joost* auf dem De-facto-Standard SAX (Simple API for XML). Dieses API wurde bereits in Kapitel 3.2.1 kurz vorgestellt und charakterisiert. Eine umfassende Darstellung gibt Brownell in [Bro02].

SAX ist in erster Linie ein Parser-API, über das ein XML-Text für eine anschließende Verarbeitung eingelesen wird. Die XML-Daten werden dabei durch einen *SAX-Eventstrom* repräsentiert. Das typische Einsatzszenario beinhaltet auf der Produzentenseite einen XML-Parser, der beim Einlesen eines XML-Textes einen solchen SAX-Eventstrom produziert, und auf der Verbraucherseite applikationsspezifische Handler-Objekte, die diesen Strom konsumieren. Solche Handler-Objekte müssen die durch SAX spezifizierten Callback-Interfaces implementieren. Die für die eigentlichen XML-Daten relevanten Interfaces sind dabei `ContentHandler` und `LexicalHandler`.

[1] Auch *Joost* spielt mit der in der Open-Source-Szene beliebten Tradition rekursiver Akronyme und lässt sich zu *Joost is Oli's Original Streaming Transformer* expandieren. In Wirklichkeit handelt es sich hier jedoch um ein Backronym.

[2] Diese Zahl ergibt sich aus der durchschnittlichen Anzahl von Downloads pro veröffentlichter Version. Die erste *Joost*-Version wurde im August 2002 auf SourceForge veröffentlicht.

Handler-Interfaces

Über das Interface `ContentHandler` werden einer Applikation die folgenden Informationen übergeben:

- Anfang und Ende des Dokuments (`startDocument` und `endDocument`),
- Anfang und Ende von Elementen (`startElement` und `endElement`),
- Anfang und Ende des Gültigkeitsbereichs von Namensraumdeklarationen (`startPrefixMapping` und `endPrefixMapping`),
- die in den Elementen enthaltenen Textdaten (`characters` und `ignorableWhitespace`),
- sowie die im Dokument auftretenden Verarbeitungsanweisungen (`processingInstruction`).

Über das Interface `LexicalHandler` werden dagegen lexikalische Informationen gemeldet, beispielsweise

- im Dokument auftretende Kommentare (`comment`),[3]
- Anfang und Ende von CDATA-Abschnitten (`startCDATA` und `endCDATA`),
- Anfang und Ende von DTD-Deklarationen (`startDTD` und `endDTD`).

Weitere Interfaces nehmen Events für Informationen aus der DTD entgegen oder fungieren als Empfänger von Fehlermeldungen.

Der Einsatz von SAX ist jedoch nicht nur auf dieses Szenario beschränkt, in dem ein XML-Parser SAX-Events direkt an eine spezifische Applikation liefert. Tatsächlich können beliebige Komponenten in der Rolle des Event-Produzenten agieren. So könnte eine Applikation (etwa eine Datenbank) eine XML-Sicht auf ihre Daten in Form eines SAX-Eventstroms anbieten. Spezielle Parser für Nicht-XML-Daten könnten geeignete SAX-Events generieren und auf diese Weise eine anschließende Verarbeitung dieser Daten durch beliebige XML-Werkzeuge ermöglichen.

SAX-Filter

Darüber hinaus kann ein Konsument eines SAX-Eventstroms ebenfalls als Produzent auftreten. Eine solche Komponente, die SAX-Events empfängt und SAX-Events produziert, wird *SAX-Filter* genannt. SAX stellt dafür das Interface `XMLFilter` zur Verfügung. Dieses kann von einem Konsumenten genauso wie ein SAX-Parser (ein `XMLReader`-Objekt) benutzt werden. Es unterscheidet sich jedoch von einem echten Parser dadurch, dass es keinen XML-Text einliest, sondern bereits geparste XML-Daten als SAX-Events von einem anderen `XMLReader`, dem so genannten *parent*, entgegen nimmt.

Üblicherweise werden durch eine Filter-Komponente in SAX an den eingehenden XML-Daten nur Änderungen auf Eventebene vorgenommen. Komplexere Änderungen, die den Charakter einer XML-Transformation besitzen, lassen sich auf SAX-Ebene nur sehr aufwändig realisieren. Aus Sicht des Nutzers einer Filter-Komponente besteht jedoch kein Unterschied zwischen einem Filter und einem Transformator. So stellt *Joost* mit der Klasse `net.sf.joost.stx.Processor` eine Implementation für das Interface `XMLFilter` zur Verfügung. Ein solches `Processor`-Objekt erscheint von außen wie ein einfacher SAX-Filter, führt im Inneren jedoch eine komplexe

[3] Anders als SAX betrachtet das Infoset Kommentare durchaus als semantisch relevant und nicht nur als reine lexikalische Information.

XML-Transformation aus. SAX-basierte Anwendungen können auf diese Weise sehr einfach STX-Transformationen benutzen.

Filterketten

Filter-Komponenten in SAX lassen sich unkompliziert miteinander kombinieren. Da ein Filter immer sowohl Konsument als auch Produzent ist, können mehrere Filterobjekte zu einer Filterkette zusammengesetzt werden. Listing 41 zeigt den Einsatz von *Joost* in einer solchen Filterkette. In diesem Beispiel werden zur Demonstration zwei STX-Transformationsschritte hintereinander ausgeführt. Komplexere Transformationen lassen sich so gegebenenfalls als Verkettung mehrerer einfacher Transformationen realisieren.

Joost als XMLFilter **Listing 41**

```
1  // SAX-Produzent: Beginn der Filterkette
2  XMLReader myXMLReader = ...
3  // SAX-Konsument: Ende der Filterkette
4  ContentHandler myHandler = ...
5
6  // neue Joost-Processor-Objekte aus entsprechendem STX-Code erzeugen
7  Processor stxProc1 = new Processor(new InputSource("trans1.stx"));
8  Processor stxProc2 = new Processor(new InputSource("trans2.stx"));
9
10 // der erste Filter liest den SAX-Eingabestrom von myXMLReader
11 stxProc1.setParent(myXMLReader);
12
13 // der zweite Filter wird mit dem ersten verbunden
14 stxProc2.setParent(stxProc1);
15
16 // der SAX-Ausgabestrom wird durch myHandler verarbeitet
17 stxProc2.setContentHandler(myHandler);
18
19 // transformiere die Datei input.xml
20 stxProc2.parse("input.xml");
```

Die beiden `Processor`-Objekte werden hier zwischen einen XML-Parser (`myXMLReader`) und den XML-Endverbraucher (`myHandler`) platziert. Das erste Objekt (`stxProc1`) konsumiert die Daten des Parser (siehe Zeile 11), das zweite (`stxProc2`) konsumiert die Daten des ersten (Zeile 14). Die Architektur einer SAX-Filterkette ist eine Art Stack, in dem beim letzten Filter schließlich das Handler-Objekt (`myHandler`) angemeldet wird (Zeile 17). Dort muss dann auch die `parse`-Methode aufgerufen werden. Abbildung 10 veranschaulicht dieses Szenarium noch einmal. Die Tatsache, dass `parse` am Ende der Kette aufgerufen wird, ist nicht unbedingt intuitiv und irritiert vor allem SAX-Anfänger gelegentlich.

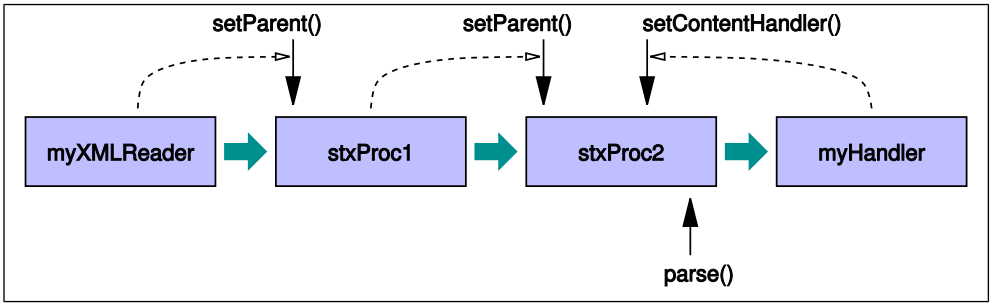


Abbildung 10
SAX-Filterkette

Für rein SAX-basierte Anwendungen ist diese Art der Einbindung von *Joost* und damit von STX-Transformationen durchaus ausreichend. Im Vergleich zu dem im nächsten Kapitel vorgestellten API TrAX fällt jedoch nachteilig ins Gewicht, dass es keinen standardisierten Weg gibt, solche Filter-Objekte zu erzeugen und zu konfigurieren. Ein Austausch der konkreten STX-Implementierung hat damit in aller Regel weitere Änderungen im Quellcode zur Folge.

In Listing 41 war zu sehen, dass der Konstruktor des zu *Joost* gehörenden `Processor`-Objekts ein SAX-`InputSource`-Objekt für das STX-Transformations-Sheet benötigt. Dabei handelt es sich jedoch um eine *Joost*-spezifische Schnittstelle, die durch keinerlei standardisiertes API vorgegeben ist. Jede andere STX-Implementierung in Java kann hier einen anderen Weg beschreiten.

Mit den weiteren Methoden und Variablen der Klasse `Processor` verhält es sich analog. So kann vor dem Beginn der Transformation ein `Processor`-Objekt durch den Aufruf spezieller Methoden geeignet konfiguriert werden. Listing 42 enthält einen Ausschnitt aus der Klasse `Processor`. Dieser zeigt exemplarisch die beiden Methoden für den Zugriff auf die externen Parameter eines STX-Transformations-Sheet sowie zwei Methoden für die Registrierung spezieller Callback-Objekte.

Listing 42 *Ausschnitt aus der Klasse net.sf.joost.stx.Processor*

```
1  package net.sf.joost.stx;
2
3  public class Processor extends XMLFilterImpl
4     implements Constants, LexicalHandler
5  {
6     ...
7     // übergibt einen Wert für einen globalen Parameter im Transformations-Sheet
8     public void setParameter(String name, Object value)
9
10    // erfragt den Wert eines globalen Parameters im Transformations-Sheet
11    public Object getParameter(String name)
12
13    // registriert einen ErrorListener für diese Transformation
14    public void setErrorListener(ErrorListener listener)
15
16    // registriert einen URIResolver für diese Transformation
17    public void setURIResolver(URIResolver resolver)
18
19    ...
20 }
```

Java-Applikationen, die die Klasse `Processor` direkt verwenden, sind damit an *Joost* gebunden und können nicht ohne Anpassung des Quelltextes die STX-Implementation wechseln. In Bezug auf STX ist solcher Code nicht portabel.

7.2 Das Transformations-API in Java

Java-Anwendungen, die STX-basierte Transformationen ausführen wollen, benötigen dazu eine standardisierte Schnittstelle (ein API), die unabhängig von der konkreten STX-Implementierung funktioniert. Auf diese Weise kann leicht die Implementierung

der STX-Komponente durch eine andere ersetzt werden. Abhängigkeiten von einem bestimmten Produkt oder einem bestimmten Hersteller lassen sich vermeiden.

Für XSLT existiert in Java ein solches API. Es wurde ursprünglich unter dem Namen TrAX (*Transformation API for XML*) entwickelt und ist seit Version 1.1 (Februar 2001) Bestandteil von JAXP (*Java API for XML Processing*).

Unter den Designzielen für TrAX findet sich eine größtmögliche Unabhängigkeit von der konkreten Art der XML-Quelle und der Art des XML-Ergebnisses. Dazu werden die Interfaces `Source` und `Result` im Paket `javax.xml.transform` definiert. In JAXP selbst sind konkrete Klassen für Byteströme (`StreamSource`, `StreamResult`), SAX-Eventströme (`SAXSource`, `SAXResult`) sowie DOM-Bäume (`DOMSource`, `DOMResult`) enthalten. Andere APIs für die XML-Verarbeitung können eigene Ausprägungen bereitstellen, wie beispielsweise JDOM mit den Klassen `JDOMSource` und `JDOMResult` [JDOM].

In der Ankündigung von TrAX auf der XSLT-Mailingliste findet sich darüber hinaus der Satz »*While this interface is modeled on the XSLT process, it should be generic enough to use with many types of transformations besides XSLT.*«[4] Mangels Alternativen gab es in der Realität jedoch bisher ausschließlich XSLT-basierte Implementationen der TrAX-Schnittstellen.

Tatsächlich werden in diesem API keine Details der Transformationssprache sichtbar. Es ist z.B. nicht möglich, auf XSLT-Anweisungen oder XSLT-Variablen zuzugreifen. Ebenso existiert in TrAX keine Java-Repräsentation für XPath-Ausdrücke etc. Sämtliche Interna der Transformation bleiben verborgen. So gesehen entspricht das Transformationsobjekt `Transformer` einer »Black Box«, deren interne Funktionsweise nicht zugänglich ist. Über TrAX kann auch jede andere Transformationssprache gekapselt werden.

Überblick

Die in TrAX enthaltenen Funktionen ermöglichen

- die Erzeugung von Transformationsobjekten, deren Transformationsanweisungen (etwa XSLT) aus einer XML-Quelle (`Source`) stammen,
- das Setzen von Ausgabeeigenschaften für die Transformation,
- die Übergabe von externen Parametern an die Transformation
- und schließlich die Ausführung der Transformation.

Listing 43 zeigt diese Schritte anhand eines minimalen Beispiels.

Ein einfacher Aufruf einer Transformation in TrAX **Listing 43**

```
1 import javax.xml.transform.*;
2 import javax.xml.transform.stream.*;
3
4 public class TraxExample
5 {
6    public static void main(String[] args)
7    {
8       String sourceId = args[0];    // die XML-Quelle
9       String stylesheet = args[1];  // das XSLT-Stylesheet
```

[4] übersetzt: »Obwohl diese Schnittstelle basierend auf XSLT modelliert wurde, sollte sie generisch genug sein, um mit vielen Transformationsarten neben XSLT benutzt zu werden.«
Siehe http://www.biglist.com/lists/xsl-list/archives/200004/msg01138.html

```
10
11        try {
12           // Factory erzeugen
13           TransformerFactory factory = TransformerFactory.newInstance();
14
15           // Transformer-Objekt erzeugen
16           Transformer transformer =
17              factory.newTransformer(new StreamSource(stylesheet));
18
19           // Ausführung der Transformation,
20           // Ausgabe des Ergebnisses auf die Konsole
21           transformer.transform(new StreamSource(sourceId),
22                                 new StreamResult(System.out));
23        }
24        catch (TransformerException e) {
25           // Fehlerbehandlung
26           System.err.println(e.getMessage());
27        }
28     }
29  }
```

Der abgedruckte Code implementiert im wesentlichen ein Kommandozeilentool für XSLT. Der erste Kommandozeilenparameter (`args[0]`) enthält den Namen der zu transformierenden XML-Datei, der zweite Kommandozeilenparameter (`args[1]`) den Namen des Stylesheet, das die Transformationsanweisungen beinhaltet.

Innerhalb des `try`-Blockes werden dann die drei zentralen Schritte bei der Verwendung von TrAX ausgeführt: es wird eine Factory für `Transformer`-Objekte angelegt (Zeile 13), diese produziert anschließend unter Angabe der zu verwendenden Transformationsanweisungen ein geeignetes `Transformer`-Objekt (Zeile 17), welches schließlich die Transformation ausführt (Zeile 21).

Ausgabeeigenschaften und Parameter

Möchte man für die Transformation spezielle Ausgabeeigenschaften setzen oder externe Parameter an sie übergeben, so stellt das erhaltene `Transformer`-Objekt dafür spezielle Methoden zur Verfügung, siehe Listing 44.

Listing 44 *Ausgabeeigenschaften und Parameterübergabe in TrAX*

```
1  import java.util.*;
2  import javax.xml.transform.*;
3  import javax.xml.transform.stream.*;
4
5  public class TraxExample
6  {
7     public static void main(String[] args)
8     {
9        // Anlegen des transformer-Objekts wie bekannt ...
10
11       // Ausgabeeigenschaften setzen (bzw. ändern)
12       Properties oprops = new Properties();
13       oprops.put(OutputKeys.ENCODING, "iso-8859-1");
14       transformer.setOutputProperties(oprops);
15
16       // Wert für einen externen Parameter übergeben
17       transformer.setParameter("language", "en");
18
19       // Ausführung der Transformation wie bekannt ...
```

```
20      }
21  }
```

Hier wird die gewünschte Kodierung `iso-8859-1` als Ausgabeeigenschaft gesetzt (Zeile 13) sowie die Zeichenkette `"en"` für den Parameter `language` übergeben (Zeile 17). Dies ist natürlich nur dann sinnvoll, wenn das verwendete Stylesheet eine Parameterdefinition der Form `<xsl:param name="language" />` besitzt.

Für alle in XSLT 1.0 im Element `xsl:output` definierten Ausgabeeigenschaften existiert in der Klasse `OutputKeys` eine Konstante, deren Wert eine Zeichenkette ist. In obigem Beispiel hätte anstelle der Konstanten `OutputKeys.ENCODING` ebenso der Wert `"encoding"` angegeben werden können. Die Benutzung der Konstanten hat jedoch den Vorteil, dass eventuelle Schreibfehler bereits vom Compiler festgestellt werden.

Geparste Stylesheets

Neben der hier beispielhaft angegebenen Grundform für die Ausführung von XML-Transformationen über TrAX existiert eine weitere Variante, in der zunächst aus den Transformationsanweisungen ein `Templates`-Objekt erzeugt wird. Dieses stellt die Laufzeitrepräsentation eines Stylesheet dar und ermöglicht die effiziente Erzeugung mehrerer `Transformer`-Objekte. Auf diese Weise kann derselbe Transformationscode mehrmals verwendet werden, ohne dass die dazugehörigen Anweisungen mehrfach eingelesen werden müssen.

SAX-Anbindung

Für SAX existiert eine spezielle Variante, die die Transformation innerhalb einer SAX-Verarbeitungskette erlaubt. Die dafür zur Verfügung stehende Klasse `SAXTransformerFactory` ist ein Spezialfall der bekannten `TransformerFactory`. Sie enthält unter anderem die zusätzliche Methode `newXMLFilter`, die ein Transformationsobjekt in Form eines SAX-`XMLFilters` zurückgibt.

An dieser Stelle fällt jedoch eine Unzulänglichkeit in JAXP 1.1 auf: einer auf diese Weise ausgeführten Transformation lassen sich keine externen Parameter übergeben. Stylesheet-Parameter in XSLT (`xsl:param`) lassen sich somit nicht benutzen.

Transformer-Handler

JAXP definiert stattdessen für die Erstellung von SAX-Filterketten ein neues Interface namens `TransformerHandler`. Dieses ist von den SAX-Interfaces `ContentHandler` und `LexicalHandler` abgeleitet und lässt sich intuitiver einsetzen als ein `XMLFilter`. Über die zusätzliche Methode `setResult` können Instanzen verschiedener `Result`-Ergebnistypen bei einem `TransformerHandler` angemeldet werden.

Listing 45 demonstriert die Transformation eines SAX-Datenstroms in einer Pipeline mit zwei beteiligten XSLT-Stylesheets.

Verwendung von TransformerHandler **Listing 45**

```
1  TransformerFactory tFactory = TransformerFactory.newInstance();
2  if (tFactory.getFeature(SAXTransformerFactory.FEATURE)) {
3     // Ok, SAX-Transformationen werden von dieser Factory unterstützt
4     SAXTransformerFactory saxTFactory = (SAXTransformerFactory) tFactory;
5     // erzeuge zwei (XSLT) Transformer-Objekte
6     TransformerHandler tHandler1 =
7        saxTFactory.newTransformerHandler(new StreamSource("step1.xsl"));
8     TransformerHandler tHandler2 =
9        saxTFactory.newTransformerHandler(new StreamSource("step2.xsl"));
```

```
10     XMLReader myReader = XMLReaderFactory.createXMLReader();
11     ContentHandler mySerializer = ...
12
13     // Aufbau einer Transformationskette
14     myReader.setContentHandler(tHandler1);
15     myReader.setProperty("http://xml.org/sax/properties/lexical-handler",
16                          tHandler1);
17     tHandler1.setResult(new SAXResult(tHandler2));
18     tHandler2.setResult(new SAXResult(mySerializer));
19
20     // Start der Transformation
21     myReader.parse("input.xml");
22  }
23  else {
24     // SAX-basierte Transformation nicht möglich ...
25  }
```

Zunächst wird eine einfache `TransformerFactory` in eine `SAXTransformerFactory` umgewandelt, nachdem über die `getFeature`-Methode (Zeile 2) sichergestellt wurde, dass diese Factory SAX-basierte Transformationen unterstützt.

Anschließend erzeugt die Factory zwei `TransformerHandler`-Objekte, die die Transformationsanweisungen aus der jeweils angegebenen Quelle (hier *step1.xsl* und *step2.xsl*) benutzen. Das erste dieser Objekte (`tHandler1`) wird dabei wie ein normaler SAX-Konsument eingesetzt (siehe Zeilen 14 bis 16). Das zweite Objekt (`tHandler2`) wird danach über die Methode `setResult` mit der ersten Transformation verbunden, sodass eine zweistufige Transformationskette entsteht.

Stylesheet-Parameter müssen über einen kleinen Umweg an eine solche Transformation übergeben werden. Ein `TransformerHandler` liefert über die Methode `getTransformer` ein passendes `Transformer`-Objekt, an dem die Methode `setParameter` aufgerufen werden kann.

Auch wenn die konstruierte Transformationskette einen kontinuierlichen Fluss von SAX-Events nahe legt, geschieht doch die Transformation im Falle von XSLT schubweise: `tHandler1` konsumiert das gesamte Dokument, bevor das Ergebnis der Transformation in Form von SAX-Events an `tHandler2` übergeben wird. Dieser wartet wiederum das Ende des Eventstromes ab, bevor mit der zweiten Transformation begonnen wird. XSLT-Transformationen in einem SAX-Eventfluss wirken wie Staustufen. Allerdings wurde auch bereits darauf hingewiesen, dass TrAX keineswegs XSLT-spezifisch ist.

TrAX und STX

Wenn TrAX also so generisch entworfen wurde, dass es auch die Benutzung vieler anderer Arten von Transformationen ermöglicht, woher »wissen« die beteiligten Objekte dann, dass es sich um XSLT handelt? Die Antwort darauf lautet: Das ist implementationsabhängig.

Das in TrAX angewendete *Factory Pattern* (siehe [GHJ+95]) ermöglicht den Austausch der verwendeten konkreten Klassen, ohne dass dazu der Quelltext umgeschrieben werden muss. Die Klasse `TransformerFactory` ist abstrakt; sie kann nicht direkt instantiiert werden. Stattdessen liefert der Aufruf

```
TransformerFactory factory = TransformerFactory.newInstance();
```

ein Objekt vom Typ `TransformerFactory`. Von welcher konkreten abgeleiteten Klasse dieses Objekt eine Instanz ist, kann in Java über spezielle Properties eingestellt werden. Ohne diese zusätzliche Einstellung erhält man in Java 1.4 immer ein Factory-Objekt, das XSLT-Prozessoren »produziert«.

Das System-Property für den Namen der zu verwendenden Implementationsklasse heißt

```
"javax.xml.transform.TransformerFactory"
```

Um TrAX für eine andere Transformationssprache zu benutzen, muss man also nur für dieses Property eine Klasse angeben, die Transformationsobjekte für diese andere Sprache produziert.[5]

Die STX-Implementierung *Joost* unterstützt die TrAX-Interfaces. Der Klassenname für die `TransformerFactory` lautet in diesem Fall

```
"net.sf.joost.trax.TransformerFactoryImpl"
```

System-Properties können während der Programmausführung dynamisch gesetzt werden. Der folgende, in Listing 46 dargestellte Mechanismus entscheidet anhand des Namens der Transformationsdatei, ob eine XSLT- oder eine STX-Transformation durchgeführt werden soll.

Dynamische Auswahl der Transformations-Factory **Listing 46**

```
1  if (stylesheet.endsWith(".stx"))
2     System.setProperty("javax.xml.transform.TransformerFactory",
3                        "net.sf.joost.trax.TransformerFactoryImpl");
4  else
5     System.setProperty("javax.xml.transform.TransformerFactory",
6                        "com.icl.saxon.TransformerFactoryImpl");
7
8  TransformerFactory factory = TransformerFactory.newInstance();
9  Transformer transformer =
10    factory.newTransformer(new StreamSource(stylesheet));
11 // usw ...
```

Die Zeichenkette `"com.icl.saxon.TransformerFactoryImpl"` in Zeile 6 verweist auf den XSLT-Prozessor Saxon.[6] Leider sieht TrAX derzeit keine Möglichkeit vor, die gewünschte Transformationssprache zu spezifizieren, ohne eine konkrete Implementierung anzugeben. Sinnvoll wäre eine Zwischenstufe, auf der ein Nutzer des TrAX-API angeben kann, dass beispielsweise eine STX-Transformation ausgeführt werden soll, die konkrete Implementationsklasse jedoch plattformabhängig über geeignete Properties zur Laufzeit ausgewählt wird.

Wie bereits erwähnt, enthält die Klasse `OutputKeys` geeignete Konstanten für alle in XSLT 1.0 definierten Ausgabeeigenschaften. Sie ist damit XSLT-spezifisch. Dies schränkt jedoch die Unabhängigkeit des API nicht ein, da der Zugriff auf diese Eigenschaften über die Standardklasse `java.util.Properties` erfolgt. Jede Eigenschaft wird als Zeichenkette angesprochen. Andere Transformationssprachen können

[5] Es ist sogar denkbar, eine *generische* Factory zu implementieren, die erst beim Aufruf der Methode `newTransformer` anhand der vorliegenden Transformationsanweisungen erkennt, um welche Sprache es sich handelt. Ein gutes Erkennungsmerkmal wäre beispielsweise der Namensraum, in dem sich das Dokumentelement befindet.

[6] Genauer: auf die Version von Saxon, die XSLT 1.0 implementiert (derzeit 6.5.3). Ab Version 7 implementiert Saxon den kommenden Standard XSLT 2.0 und verwendet auch einen anderen Klassennamen.

ohne weiteres abweichende Ausgabeeigenschaften verwenden. Das Vorhandensein entsprechender Konstanten in `OutputKeys` bedeutet darüber hinaus nicht, dass jede Implementation diese Werte unterstützen muss. Unbekannte Schlüsselwerte können mit der Ausnahme `IllegalArgumentException` abgewiesen werden.

An dieser Stelle konnten nicht alle Möglichkeiten von TrAX erschöpfend behandelt werden. Eine ausführliche Darstellung über TrAX im Kontext von STX gibt Zubow in [Zub02].

Resümee

Ziel eines API für STX war zunächst das Bestreben, eine standardisierte Schnittstelle für STX-basierte Transformationen zu entwerfen. Ein solches API existiert für XSLT bereits unter dem Namen TrAX. Es ist ebenso für STX geeignet. Damit ergeben sich sogar weitere Vorteile:

- Existierende Applikationen, die über die TrAX-Schnittstellen XSLT-basierte Transformationen ausgeführt haben, können mit minimalem Aufwand ebenfalls STX-basierte Transformationen benutzen.
- Es kann dynamisch zwischen verschiedenen Transformationsarten umgeschaltet werden.

Unter anderem wurde *Joost* von den Apache-Entwicklern ohne großen Aufwand in das Publishing Framework Cocoon [ASFd] eingebunden. Dies demonstriert eindrucksvoll die Praxisrelevanz des gewählten TrAX-Ansatzes.

7.3 Interaktion mit externen SAX-Filtern

Im Kapitel 5.6.7 wurde ein STX-Sprachmittel vorgestellt, mit dem sich Teile des Eingabedokuments (XML-Fragmente) an externe Filterprozesse übergeben lassen. Der jeweilige Filterprozess wurde über einen URI spezifiziert. Alle weiteren Details der Zusammenarbeit des STX-Prozessors mit dem externen Filter werden der jeweiligen Implementierung überlassen und sind aus STX-Sicht transparent.

Da *Joost* auf SAX basiert, bietet es sich zunächst an, solche externen Filter über das Interface `XMLFilter` einzubinden. Allerdings sieht SAX selbst keinen implementationsunabhängigen Weg vor, auf dem sich solch ein Filterobjekt erzeugen lässt. Bei Verwendung der TrAX-Schnittstellen lassen sich hingegen, wie oben angesprochen, keine externen Parameter an den Filter übergeben.

Stattdessen nutzt *Joost* an dieser Stelle das vorgestellte Interface `TransformerHandler`. Bereits existierende SAX-Filterimplementationen lassen sich einbinden und damit für STX verfügbar machen, indem sie durch ein `TransformerHandler`-Objekt gekapselt werden. Die Zuordnung zwischen einem als `filter-method` angegebenen URI und der konkreten Implementationsklasse kann in *Joost* durch den Anwender konfiguriert werden.

Vom URI zum Filter

Das dazu verwendete Entwurfsmuster folgt dem `URIResolver` in TrAX: dieser liefert auf Anfrage zu einem URI ein `Source`-Objekt, das die durch den URI adressierten XML-Daten bereitstellt. `URIResolver` selbst ist ein Interface, das anwendungsspezifisch implementiert und dann über entsprechende TrAX-Methoden bei dem verwendeten Transformations-Prozessor angemeldet wird.

Diesem Muster folgend sieht *Joost* ein Interface `TransformerHandlerResolver` vor, das zu einem im STX-Code als `filter-method` angegebenen URI ein `TransformerHandler`-Objekt zurückliefert.[7] Abhängig von der Art der im Attribut `filter-src` eventuell angegebenen Transformations-Quelle existieren zwei Varianten der Methode `resolve` in diesem Interface, siehe Listing 47.

Das Interface TransformerHandlerResolver **Listing 47**

```
1  public interface TransformerHandlerResolver
2  {
3     // wird aufgerufen bei einer url(...) Angabe in filter-src
4     TransformerHandler resolve(String method, String href, String base,
5                                Hashtable params)
6        throws SAXException;
7
8     // wird aufgerufen bei einer buffer(...) Angabe in filter-src
9     TransformerHandler resolve(String method, XMLReader reader, Hashtable params)
10       throws SAXException;
11
12    // implementiert die STX-Function filter-available()
13    boolean available(String filter);
14 }
```

Die erste Variante von `resolve` (Zeile 4) wird aufgerufen, wenn im STX-Code für `filter-src` eine URL-Spezifikation angegeben wurde oder dieses Attribut fehlt. Der Parameter `href` enthält den dort angegebenen URL. Er enthält den Wert `null` bei Fehlen des Attributes. In `base` steht der Basis-URI des STX-Transformations-Sheet.

Die zweite Variante (Zeile 9) wird dann aufgerufen, wenn der Inhalt eines STX-Puffers als Quelle der Transformationsanweisungen benutzt werden soll (`filter-src="buffer(buffer-name)"`). Dieser Inhalt des Puffers wird über ein `XMLReader`-Objekt zur Verfügung gestellt.

In beiden Varianten enthält der erste Parameter `method` den URI der gewünschten Filtermethode (den Inhalt des Attributs `filter-method`) und der letzte Parameter `params` die mittels `stx:with-param` übergebenen Parameter. Analog zu `URIResolver` geben diese Methoden den Wert `null` zurück, wenn die Bestimmung eines `TransformerHandler`s dem STX-Prozessor (in diesem Falle *Joost*) überlassen werden soll.

Hervorzuheben ist, dass nicht nur eine einzige `resolve`-Methode mit einem Parameter vom Typ `SAXSource` vorgesehen ist. Zwar ließe sich in dieses Design der `URIResolver` für als `url(...)` übergebene Filterquellen ideal einbinden. Voraussetzung wäre jedoch, dass eine solche Filterquelle immer als XML vorliegt. Dies ist aber nicht notwendigerweise der Fall. Zwar muss der Filter selbst XML-Daten verarbeiten, allerdings kann die eventuell dafür anzugebende Quelle in einem beliebigen Format vorliegen. Aus diesem Grund kann nicht davon ausgegangen werden, dass ein URL immer als `SAXSource` ausgewertet werden kann.

[7] Leider ist die Terminologie an dieser Stelle nicht konsistent: ein `URIResolver` liefert für einen *URI* eine XML-Datenquelle (`Source`), ein `TransformerHandlerResolver` liefert jedoch für einen URI einen *`TransformerHandler`*.

Die letzte Methode `available` (Zeile 13) wird dann aufgerufen, wenn im STX-Code ein Aufruf der Funktion `filter-available` erfolgt.

Der durch die `resolve`-Methoden zurückgegebene `TransformerHandler` muss in seiner `setResult`-Methode ein Objekt vom Typ `SAXResult` akzeptieren. Für die anderen drei Methoden `setSystemId`, `getSystemId` und `getTransformer` können leere Implementierungen bereitgestellt werden. Sie werden von *Joost* niemals aufgerufen. Da die Transformations-Parameter bereits in den `resolve`-Methoden übergeben werden, entfällt hier insbesondere der Aufwand, allein für die Übergabe dieser Parameter ein `Transformer`-Objekt zu implementieren.

Bei der Programmierung eines solchen `TransformerHandler`-Objekts ist zu beachten, dass vom STX-Prozessor in der Regel keine vollständigen Dokumente, sondern nur XML-Fragmente übergeben werden. Das bedeutet, dass Textinhalt in der Form von `characters`-Events bereits vor dem ersten Start-Tag auftreten kann. Zudem kann es mehrere XML-Elemente auf der äußersten Ebene geben. Durch den STX-Prozessor wird sichergestellt, dass der gesamte Eventstrom durch die Events `startDocument` und `endDocument` begrenzt ist. Außerhalb des Fragments deklarierte Namensräume werden direkt im Anschluss an das `startDocument`-Event durch entsprechende `startPrefixMapping`-Events mitgeteilt.

Ein Beispiel

Im Folgenden soll kurz skizziert werden, wie ein existierender XML-Filter als `TransformerHandler` eingebunden werden kann. Die Klasse `AppFilter` sei die vorhandene Implementation eines solchen Filters. Dabei sei vorausgesetzt, dass `AppFilter` von der SAX-Hilfsklasse `XMLFilterImpl` abgeleitet ist und zudem das Interface `LexicalHandler` implementiert. Diese Klasse wird dann, wie in Listing 48 gezeigt, zu einem `TransformerHandler` erweitert. Darüber hinaus fungiert die neue Klasse ebenfalls als `TransformerHandlerResolver` für diesen Filter. Im Normalfall sollte man diese Funktionen auf zwei separate Klassen verteilen, insbesondere wenn ein `TransformerHandlerResolver` für mehrere `TransformerHandler`-Objekte zuständig ist.

Listing 48 *Beispiel-TransformerHandler*

```
1  public class AppFilterTransformer
2     extends AppFilter
3     implements TransformerHandler,
4                net.sf.joost.TransformerHandlerResolver
5  {
6     // --------------- spezifiziert durch TransformerHandler ---------------
7
8     public void setResult(Result result)
9     {
10       // Joost übergibt immer ein SAXResult-Objekt
11       SAXResult sresult = (SAXResult)result;
12       setContentHandler(sresult.getHandler());
13       try {
14          setProperty("http://xml.org/sax/properties/lexical-handler",
15                      sresult.getLexicalHandler());
16       } catch (SAXException ex) { /* das sollte nicht passieren */ }
17    }
18
19    // setSystemId, getSystemId, getTransformer sind leer oder liefern null ...
```

```
20
21
22      // ----------- spezifiziert durch TransformerHandlerResolver -----------
23
24      // Die diesen Filter identifizierende Methode
25      static final String METHOD = "http://example.org/AppFilter";
26
27      public TransformerHandler resolve(String method, String href, String base,
28                                        Hashtable params)
29         throws SAXException
30      {
31         // wurde die richtige Methode angegeben?
32         if (METHOD.equals(method)) {
33            // wurde das Attribut filter-src angegeben?
34            if (href != null)
35               throw new SAXException("filter-src not allowed for " + method);
36            // wurden Parameter übergeben?
37            if (!params.isEmpty())
38               throw new SAXException("parameters not allowed for " + method);
39
40            // alles ok, dieses Objekt selbst ist der TransformerHandler
41            return this;
42         }
43         else
44            // unbekannte Methode
45            return null;
46      }
47
48      // zweite resolve-Methode analog
49      // diese liefert aber bei METHOD.equals(method) immer eine Exception ...
50
51      public boolean available(String method)
52      {
53         return METHOD.equals(method);
54      }
55   }
```

Wie bereits erwähnt, muss bei einen `TransformerHandler` für den Einsatz in *Joost* nur die zusätzliche Methode `setResult` implementiert werden (Zeile 8). Da *Joost* hier immer ein `SAXResult` übergibt, können die dazugehörigen `ContentHandler`- und `LexicalHandler`-Objekte von diesem direkt erfragt und über geerbte Methoden der Basisklasse angemeldet werden.

Der zweite Teil der Klasse `AppFilterTransformer` enthält die durch das Interface `TransformerHandlerResolver` spezifizierten Methoden. Zunächst wird dazu in Zeile 25 eine Konstante für die zu diesem Filter gehörende Zeichenkette (Filtermethode) definiert. Die Implementierung der `resolve`-Methoden ist in diesem Beispiel sehr einfach, da weder eine Filterquelle noch Transformationsparameter ausgewertet werden müssen. Wenn hier entsprechende Werte angegeben wurden, lösen beide `resolve`-Methoden Exceptions aus, die zu einer STX-Fehlermeldung und dem Abbruch der Transformation führen. Ansonsten liefert die `resolve`-Methode das Objekt selbst als Ergebnis (Zeile 41). Für unbekannte Filtermethoden wird der Wert `null` zurückgegeben.

Registrierung beim STX-Prozessor

Ein solches `TransformerHandlerResolver`-Objekt muss schließlich beim STX-Prozessor angemeldet werden. Während für das Interface `URIResolver` spezielle Methoden `setURIResolver` und `getURIResolver` in den Klassen

`Transformer` und `TransformerFactory` existieren, muss dieses *Joost*-spezifische Objekt auf anderem Weg beim STX-Prozessor registriert werden.

Der von TrAX für diese Zwecke vorgesehene Mechanismus führt über die Methoden `getAttribute` bzw. `setAttribute` in der Klasse `TransformerFactory`. Diese Methoden sind dazu gedacht, implementationsspezifische Eigenschaften zu setzen bzw. abzufragen. Ein `TransformerHandlerResolver`-Objekt ist eine solche implementationsspezifische Eigenschaft. Jede dieser Eigenschaften wird durch eine Zeichenkette identifiziert. Im Falle von *Joost* existiert eine entsprechende Konstante in der Klasse `net.sf.joost.trax.TrAXConstants` mit dem Namen `KEY_TH_RESOLVER`.[8] Eine Java-Anwendung, die die in Listing 48 implementierte Klasse `AppFilterTransformer` in *Joost* benutzen möchte, muss dazu das in Listing 49 dargestellte Codefragment verwenden.

Listing 49 *Anmeldung eines TransformerHandlerResolvers*

```
// Joost verwenden
System.setProperty("javax.xml.transform.TransformerFactory",
                   "net.sf.joost.trax.TransformerFactoryImpl");

TransformerFactory tFac = TransformerFactory.newInstance();

// TransformerHandlerResolver-Objekt erzeugen
net.sf.joost.TransformerHandlerResolver thr = new AppFilterTransformer();
// und anmelden
tFac.setAttribute(net.sf.joost.trax.TrAXConstants.KEY_TH_RESOLVER, thr);

// Transformer-Objekt erzeugen wie bekannt
Transformer transformer = tFac.newTransformer(...);
```

7.4 STX als XML-Generator

Häufig müssen in einer Anwendung Daten als XML serialisiert werden. Dies erfolgt durch das Erzeugen eines XML-Textes, der die entsprechenden Daten der Anwendung enthält. Diese Aufgabe stellt sich beispielsweise

- beim Speichern oder Exportieren von Daten im XML-Format und
- bei der Bereitstellung externer Schnittstellen, durch die auf Anfragen Ergebnisse als XML zurückgegeben werden.

Integration in die Anwendung

Die zunächst nahe liegende Lösung besteht darin, eine entsprechende Funktion direkt in die Anwendung zu integrieren. In Java könnte man in Analogie zur Methode `toString()` eine Methode `toXML()` bereitstellen, die für die in der Klasse gekapselten Daten eine XML-Darstellung erzeugt, siehe Listing 50.

Listing 50 *Java-Klassen mit toXML()-Methoden*

```
/** einfache Repräsentation einer Person */
public class Person
{
```

[8] Dahinter verbirgt sich die Zeichenkette `"http://joost.sf.net/attributes/transformer-handler-resolver"`.

```
4     // gekapselte Daten
5     private String name, vorname;
6
7     // Konstruktor etc. weggelassen
8
9     /** @return eine XML-Repräsentation des Objektes */
10    public String toXML()
11    {
12       return "<person><name>" + name + "</name>\n"
13            + "<vorname>" + vorname + "</vorname></person>\n";
14    }
15 }
16
17 /** einfache Repräsentation eines Buches */
18 public class Buch
19 {
20    // gekapselte Daten
21    private Person[] autoren;
22    private String titel, isbn;
23
24    // Konstruktor etc. weggelassen
25
26    /** @return eine XML-Repräsentation des Objektes */
27    public String toXML()
28    {
29       String s = "<buch isbn='" + isbn + "'><autoren>";
30       for (int i=0; i<autoren.length; i++)
31          s += autoren[i].toXML();
32       return s + "</autoren>\n"
33                + "<titel>" + titel + "</titel></buch>\n";
34    }
35 }
```

Die Methode `toXML()` der Klasse `Person` in Zeile 10 gibt eine Zeichenkette zurück, die den XML-Text für eine Person enthält. In der gleichnamigen Methode der Klasse `Buch` wird der XML-Text für ein Buch generiert, wobei diese für die Autoren auf die `toXML`-Methode in `Person` zurückgreift (Zeile 31).

Diese Vorgehensweise hat jedoch einige Nachteile:

Nachteile der `toXML`-Variante

- Der Compiler kann nicht feststellen, ob durch eine solche Methode tatsächlich korrektes XML generiert wird. Eventuelle Fehler im Markup (beispielsweise fehlerhafte Tags) werden erst zur Laufzeit sichtbar.
- Die Daten dürfen keine Zeichen enthalten, die in XML eine besondere Bedeutung besitzen. In diesem Fall dürfen weder Ampersand (`&`) noch die öffnende spitze Klammer (`<`) in den Daten auftreten. Die `toXML`-Methode muss solche Zeichen daher gegebenenfalls maskieren. Das geschieht in der hier dargestellten einfachen Variante nicht. Daten, die als Attributwerte repräsentiert werden sollen, müssen darüber hinaus auf Leerraum- und Anführungszeichen überprüft werden.
- Die interne Modellierung der Daten und ihre Repräsentation in XML sind fest miteinander verbunden. Der Code zur Erzeugung der vollständigen XML-Repräsentation ist über mehrere Klassen verteilt. Änderungen an der XML-Struktur erfordern immer ein erneutes Kompilieren des Programmes.
- Spätere Anpassungen durch die Nutzer sind nicht möglich.

Der dritte Nachteil lässt sich durch eine Verbesserung der Architektur des Systems abmildern. Dazu muss der für die XML-Erzeugung verantwortliche Code separiert und in einer eigenen Klasse implementiert werden. Die schwerer wiegenden Probleme durch fehlerhaftes XML sowie die fehlende oder unzureichende Konfigurierbarkeit werden dadurch jedoch nicht beseitigt.

Verwendung von STX

Mit STX lässt sich dieses Problem elegant lösen. Dazu muss im Transformationscode auf die Anwendungsdaten zugegriffen werden. Tatsächlich handelt es sich hier um eine eher untypische Transformation, da die zu transformierenden Daten nicht aus einer XML-Quelle stammen, sondern direkt über Schnittstellen von der Anwendung bereitgestellt werden.

Der Nachteil dieser Vorgehensweise besteht darin, dass für den Zugriff auf die Applikationsdaten Erweiterungsmechanismen des jeweiligen STX-Prozessors genutzt werden müssen. In diesem Fall kann der STX-Code nicht mehr prozessor-unabhängig benutzt werden. Für das beschriebene Szenario ist dieser Effekt jedoch von untergeordneter Bedeutung:

- Die Anwender werden in einem installierten System nur in seltenen Fällen eine Teilkomponente austauschen wollen (hier den STX-Prozessor). Die Portabilität des STX-Codes spielt dann keine hervorgehobene Rolle.
- In Java-basierten XML-Transformatoren[9] hat sich für die Verwendung von Erweiterungsfunktionen ein Quasi-Standard herausgebildet, sodass der Wechsel zu einem anderen ebenfalls in Java implementierten STX-Prozessor einfach möglich sein sollte.

Java-Erweiterungsfunktionen

Dieser Mechanismus für Erweiterungsfunktionen soll hier kurz beschrieben werden. Zu STX oder XSLT gehören eine Menge von Standardfunktionen, die in den jeweiligen Spezifikationen beschrieben sind und von jeder konformanten Implementation unterstützt werden müssen. Darüber hinaus kann jede Implementation eigene Funktionen bereitstellen, die sich in einem prozessor-spezifischen Namensraum befinden müssen. In Java implementierte Prozessoren bieten zudem die Möglichkeit, Methoden in Java-Klassen direkt aus dem Transformationscode bzw. dem Stylesheet heraus aufzurufen. Dabei kann es sich um anwendungsspezifische Klassen oder um Standard-Klassen der Java-Laufzeitumgebung handeln.

Die Regeln für den Zugriff auf Java-Methoden sind

1. **Namensraum**
   Der Namensraum identifiziert die Klasse, aus der eine Methode benutzt werden soll. Er setzt sich zusammen aus der Zeichenkette `java:` und dem vollständig qualifizierten Klassennamen. Beispiele:

   `java:java.lang.Math` (aus der Java-Standardbibliothek)
   `java:com.enterprise.appl.part.Foo` (anwendungsspezifisch)

2. **Klassenmethoden**
   Klassenmethoden (`static`) können direkt aufgerufen werden. Durch den Namensraum ist definiert, zu welcher Klasse die Methode gehört. Zum Beispiel berechnet der folgende Aufruf die Quadratwurzel von 2. Das Präfix `m` gehört hier zum Namensraum `java:java.lang.Math`:

[9] Für STX existiert derzeit nur ein einziger in Java implementierter Prozessor (*Joost*). Dieser unterstützt jedoch den in den XSLT-Prozessoren Saxon, Xalan und jd.xslt implementierten Mechanismus.

```
<stx:value-of select="m:sqrt(2)" />
```

3. **Konstruktoren**
   Über den speziellen Funktionsnamen `new` werden Konstruktoren aufgerufen. Das zurückgegebene Objekt kann z.B. in einer STX-Variablen abgelegt werden:

   ```
   <stx:variable name="obj" select="foo:new()" />
   ```

4. **Instanzmethoden**
   Instanzmethoden werden wie Klassenmethoden aufgerufen. Dabei muss als zusätzlicher erster Parameter die Instanz des Objektes übergeben werden:

   ```
   <stx:value-of select="foo:toString($obj)" />
   ```

Für alle Methoden- und Konstruktorenaufrufe gilt, dass bei mehreren Möglichkeiten anhand der übergebenen Argumente entschieden wird, welche konkrete Methode aufgerufen werden soll. Solche Mehrdeutigkeiten treten bei überladenen Methoden auf. Auf die entsprechenden Regeln wird hier nicht näher eingegangen.

Mit Hilfe solcher Erweiterungsfunktionen lassen sich Rückgabewerte von Java-Methoden ohne großen Aufwand direkt in STX nutzen. Die zu erzeugende XML-Struktur wird unabhängig vom Java-Code in STX beschrieben. Durch den STX-Prozessor wird sichergestellt, dass korrektes XML erzeugt wird. Dies gilt sowohl im Hinblick auf das gewünschte Markup als auch auf die Repräsentation bestimmter Zeichen. Diese Eigenschaften gelten jedoch für eine Verwendung in XSLT ebenso.

Erweiterungsfunktionen in XSLT

Problematisch an einer Verbindung von XSLT und Java ist allerdings wieder der funktionale Charakter von XSLT. Es existieren keine Schleifenkonstrukte in XSLT. Große Datenmengen müssen als Ganzes der Transformation übergeben oder aber auf rekursivem Weg abgefragt werden. Hier ist jedoch nicht garantiert, dass ein XSLT-Prozessor den mehrfachen Aufruf der gleichen Funktion mit gleichen Parametern wirklich ausführt. Ein XSLT-Prozessor könnte solche Funktionen tatsächlich nur einmal aufrufen, da vorausgesetzt wird, dass Funktionsaufrufe keine Seiteneffekte besitzen.

Beispiel mit STX

Mit STX lässt sich eine speichereffiziente und zuverlässige Lösung implementieren. Das folgende Beispiel demonstriert die Serialisierung einer einfachen Bücherliste. Diese wird der STX-Transformation über einen externen Parameter namens `iter` in Form eines `Iterator`-Objekts zur Verfügung gestellt, das die einzelnen `Buch`-Objekte liefert. Listing 51 zeigt den Aufruf im Java-Code über das bereits vorgestellte TrAX-API.

Aufruf der STX-Transformation zum Serialisieren von Java-Daten **Listing 51**

```
1  // Es wird Joost verwendet
2  System.setProperty("javax.xml.transform.TransformerFactory",
3                     "net.sf.joost.trax.TransformerFactoryImpl");
4
5  TransformerFactory factory = TransformerFactory.newInstance();
6
7  // serializeCode verweist auf den STX-Code
8  Transformer transformer = factory.newTransformer(serializeCode);
9
10 // bookIterator iteriert über Buch-Objekte
11 transformer.setParameter("iter", bookIterator);
12
13 // dummySource verweist auf ein (fast) leeres XML-Dokument
```

```
14 transformer.transform(dummySource,
15                       new StreamResult(System.out));
```

Die in Zeile 8 übergebene Quelle `serializeCode` enthält den im Listing 52 angegebenen STX-Code. Die formalen XML-Eingabedaten sind in `dummySource` enthalten (Zeile 14). In diesem Fall spielen diese XML-Daten jedoch keine Rolle für das Ergebnis, sodass eine beliebige XML-Quelle benutzt werden kann. Da die enthaltenen Daten jedoch in jedem Fall vollständig gelesen werden müssen, bietet sich hier aus Effizienzgründen ein kurzes Dokument an, z.B. eines mit einem leeren Dokumentelement.

Listing 52 *STX-Transformation, die Java-Daten serialisiert*

```
1  <?xml version="1.0"?>
2  <stx:transform xmlns:stx="http://stx.sourceforge.net/2002/ns"
3                 xmlns:i="java:java.util.Iterator"
4                 xmlns:b="java:joost.example.Buch"
5                 xmlns:p="java:joost.example.Person"
6                 version="1.0"
7                 exclude-result-prefixes="i b p">
8
9    <stx:param name="iter" />
10
11   <stx:template match="/">
12     <buchliste>
13       <stx:while test="i:hasNext($iter)">
14         <stx:variable name="buch" select="i:next($iter)" />
15         <buch isbn="{b:getIsbn($buch)}">
16           <stx:variable name="autoren" select="b:getAutoren($buch)" />
17           <autoren>
18             <stx:while test="i:hasNext($autoren)">
19               <stx:variable name="autor" select="i:next($autoren)" />
20               <person>
21                 <name>
22                   <stx:value-of select="p:getName($autor)" />
23                 </name>
24                 <vorname>
25                   <stx:value-of select="p:getVorname($autor)" />
26                 </vorname>
27               </person>
28             </stx:while>
29           </autoren>
30           <titel>
31             <stx:value-of select="b:getTitel($buch)" />
32           </titel>
33         </buch>
34       </stx:while>
35     </buchliste>
36   </stx:template>
37
38 </stx:transform>
```

Zunächst werden alle benötigten Namensräume deklariert. Das Präfix `i` verweist auf die Klasse `Iterator` der Java-Standardbibliothek. Die Präfixe `b` und `p` verweisen auf die Klassen `Buch` und `Person` der eigenen Anwendung.

In Zeile 9 wird anschließend der externe Parameter `iter` deklariert, dem im vorherigen Listing 51 in Zeile 11 ein passendes `Iterator`-Objekt übergeben wurde.

Die eigentliche Transformation besteht dann darin, dass im Template für den Dokumentknoten eine `stx:while`-Schleife über den übergebenen Iterator ausgeführt wird (Zeile 13). Für jedes Element (jedes Buch) werden dabei die dazugehörigen Daten abgefragt und ausgegeben. Dies geschieht durch `stx:value-of`-Elemente oder innerhalb von Attributwert-Templates (z.B. in Zeile 15). Hier sei vorausgesetzt, dass die Klassen `Buch` und `Person` über entsprechende Methoden zur Abfrage ihrer Member-Variablen verfügen (so genannte *Getter*). Die Methode `getAutoren()` der Klasse `Buch` liefert dabei wiederum ein `Iterator`-Objekt, welches hier die Autoren des Buches durchläuft. Die folgende innere `stx:while`-Schleife funktioniert dann nach dem gleichen Muster wie die äußere.

Kapitel 8
Zusammenfassung und Ausblick

Ziel dieser Arbeit war es, eine Sprache zu entwickeln, mit der sich XML-Transformationen skalierbar auch für große Datenmengen ausführen lassen. Alle bisher verfügbaren Transformationsmethoden basieren entweder auf einer Baumdarstellung des gesamten XML-Dokuments oder verlangen die Programmierung auf der XML-fernen Ebene einer Programmiersprache.

Die in der vorliegenden Arbeit vorgestellte Sprache *Streaming Transformations for XML* (STX) füllt diese Lücke. STX ist eine XML-Transformationssprache, die ein XML-Dokument als Datenstrom verarbeitet. Ihr praktischer Nutzen konnte am Beispiel ausgewählter Einsatzszenarien mit Hilfe der prototypischen Implementation *Joost* gezeigt werden. Diese Implementation bietet zudem standardisierte Schnittstellen, über die sich sehr einfach STX-basierte Transformationen in Java-Anwendungen integrieren lassen.

Die Syntax von STX orientiert sich an der weit verbreiteten Sprache XSLT. So finden sich in STX viele bekannte XSLT-Konstrukte wieder. Die in STX enthaltene Pfadsprache ist eine Teilmenge der Sprache XPath 2.0, die aus pragmatischen Erwägungen um zwei Knotentypen und dazugehörige Knotentests erweitert wurde. STX ermöglicht die einfache Transformation generischer XML-Strukturen, ohne dazu eine Schemabeschreibung der Daten zu erfordern.

STX versus XSLT

STX ist als Ergänzung zu XSLT zu sehen, nicht als Ersatz. Das Wesen der jeweiligen Transformationsaufgabe bestimmt, welche der beiden Sprachen für den angestrebten Zweck geeigneter ist. Zwar sind XSLT und STX prinzipiell gleichmächtig in ihrer Berechnungsstärke, jedoch ist die Sprache STX aufgrund ihres seriellen Verarbeitungsmodells nicht das Mittel der Wahl, wenn Wissen über Knoten aus anderen entfernten Teilen des XML-Dokuments benötigt wird. Vielmehr eignet sich STX speziell für Transformationen, die

- die Struktur der Daten bis auf Umbenennungen unverändert lassen,
- Teildaten aus dem Eingabedokument herausfiltern oder
- allein Daten aus den Vorfahren des jeweiligen Kontextknotens für die Beschreibung der Transformationsregeln erfordern.

Im Allgemeinen lassen sich solche Transformationen gut durch STX realisieren, in denen die Reihenfolge der Eingabedaten zur Reihenfolge der Daten im Ergebnisdokument korrespondiert. Änderungen dieser Reihenfolge sollten nur lokal begrenzt erforderlich sein.

STX und XSLT lassen sich sehr gut miteinander kombinieren. Insbesondere bietet STX eine Schnittstelle zu externen Prozessen, über die sich auch XSLT-basierte Transformationen ausführen lassen. Auf diesem Weg können komplexe Transformationen erstellt werden, in denen sehr große Dokumente erst in Fragmente zerlegt und dann durch XSLT verarbeitet werden. Diese Vorgehensweise demonstriert, dass STX keine eigenen Sprachmittel für den Zugriff auf Baumstrukturen benötigt, da bereits vorhandene Technologien über diese Schnittstelle sehr einfach eingebunden werden können. Nutzer von STX können in besonderem Maße von einer Kombination mit

XSLT profitieren, da sie während der Entwicklung nicht zwischen stark unterschiedlichen Techniken wechseln müssen.

Bewertung von STX

Die Ähnlichkeit zwischen XSLT und STX hat des Weiteren zur Folge, dass die in Kapitel 3 formulierten Kriterien für Transformationsmethoden bis auf zwei Kriterien bei beiden Sprachen gleich bewertet werden. Unterschiede bestehen lediglich im Hinblick auf die Skalierbarkeit und die Mächtigkeit. So ist STX im Gegensatz zu XSLT sehr gut skalierbar, da der für die Transformation benötigte Hauptspeicher in erster Linie von der Tiefe des XML-Baumes abhängt, nicht jedoch von dessen Breite (siehe Kapitel 5.5). Somit lassen sich mit STX beliebig große XML-Dokumente transformieren. Dieser Verbesserung der Skalierbarkeit steht jedoch eine Verringerung der Mächtigkeit gegenüber, da komplexe Transformationen sich nicht skalierbar realisieren lassen bzw. nur sehr ineffizient auf Kosten der Skalierbarkeit umgesetzt werden können.

Umwandlung von XSLT in STX

Die Entwicklung von STX als weiterer Transformationsmethode führt angesichts des umfangreichen Fundus an XSLT-Code zu der Frage, inwieweit existierende XSLT-Stylesheets automatisch in STX-Transformations-Sheets übersetzt werden können. Aber auch die andere Richtung wird bei zunehmendem Einsatz von STX zukünftig von Interesse sein. Die Beantwortung beider Fragen erweist sich als eine nicht zu unterschätzende Herausforderung. Abgesehen von Trivialfällen muss für eine solche Übersetzung der Transformations-Algorithmus in der Regel grundlegend neu formuliert werden. Diese Aufgabe besitzt den gleichen Schwierigkeitsgrad wie das Entscheidungsproblem, ob eine gegebene XSLT-Transformation seriell ausgeführt werden kann, und bietet so Raum für weitere Forschungen.

Ausblick

Grundlage dieser Dissertation ist der Stand der STX-Entwicklung im Frühjahr 2004. Diese Entwicklung ist jedoch nicht abgeschlossen. So wurden auf der STX-Mailingliste weitere Vorschläge in die Diskussion gebracht, die in dieser Arbeit noch nicht berücksichtigt werden konnten. Ein wachsender Bekanntheitsgrad von STX wird darüber hinaus zu größeren Nutzerzahlen, breiteren Anwendungsgebieten und neuen Anforderungen führen, die sich adäquat im STX-Sprachumfang widerspiegeln sollten.

Ein weiterer wesentlicher Aspekt ist in diesem Zusammenhang die Zukunft von XSLT. Der Spezifikationsprozess von XSLT 2.0 geht derzeit in die Endphase über. Eine Revision von STX erscheint spätestens bei der Fertigstellung von XSLT 2.0 sinnvoll, da sich einige der dort eingeführten neuen Konzepte ebenfalls für STX als geeignet erweisen könnten. Nicht zuletzt muss die aus XPath 2.0 abgeleitete Pfadsprache STXPath mit der endgültigen XPath-Empfehlung abgeglichen werden.

Implementierungen

Die STX-Implementierungen bieten Raum für weitere Verbesserungen. So implementiert keiner der verfügbaren Prozessoren[1] vollständig den derzeitig beschriebenen Sprachumfang. Beispielsweise unterstützt *Joost* in der aktuellen Version vom 30. März 2004 weder reguläre Ausdrücke, noch gibt es eine Umsetzung des neuen Konzepts der in Kapitel 5.6.8 vorgestellten Fehlerbehandlung. Darüber hinaus könnte *Joost* weiter optimiert werden, um eine mit gängigen XSLT-Prozessoren vergleichbare Performance zu erreichen.

[1] Neben *Joost* existiert derzeit nur ein weiterer STX-Prozessor, der von Petr Cimprich in Perl implementierte XML::STX.

In einer zukünftigen *Joost*-Version ist die Bereitstellung einer von STX unabhängigen STXPath-Bibliothek vorgesehen. Diese wird SAX-Programmierern eine STXPath-Mustererkennung zur Verfügung stellen und damit eine zentrale Aufgabe der SAX-Programmierung – die Kontextverwaltung – lösen. Somit kann STXPath über den Einsatzbereich der Transformationen hinaus für SAX-Anwendungen eingesetzt werden.

Standardisierung

Als eines der längerfristigen Ziele kann die Fortführung der STX-Entwicklung unter dem Dach eines Standardisierungsgremiums gesehen werden. Mögliche Adressaten sind die Konsortien OASIS und W3C. So hat OASIS u.a. die Weiterentwicklung der Schemasprache Relax NG [OASIS01] übernommen. Für das W3C würde eine so genannte Notiz (*note*) von W3C-Mitgliedern der erste Schritt in Richtung Standardisierung sein. Eine solche Notiz könnte den Anstoß für die Gründung einer Arbeitsgruppe geben, die sich aufbauend auf STX mit der Entwicklung einer seriellen XML-Transformationssprache befasst. Die STX-Entwicklergemeinde hat sich damit sehr ehrgeizige Ziele gesetzt, von deren Umsetzung vor allem die Anwender von STX profitieren werden.

Anhang A
XML-Schema für STX

Das hier angegebene Schema für STX ist in der vom W3C spezifizierten Schemasprache für XML [W3C01b] formuliert. Alternativ können zur Validierung von STX-Transformations-Sheets gleichberechtigt andere Schemasprachen verwendet werden, wie z.B. RelaxNG [OASIS01].

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:stx="http://stx.sourceforge.net/2002/ns"
           xmlns="http://stx.sourceforge.net/2002/ns"
           targetNamespace="http://stx.sourceforge.net/2002/ns"
           elementFormDefault="qualified">

  <xs:group name="group-content">
    <xs:choice>
      <xs:element ref="variable"/>
      <xs:element ref="param"/>
      <xs:element ref="buffer"/>
      <xs:element ref="template"/>
      <xs:element ref="procedure"/>
      <xs:element ref="include"/>
      <xs:element ref="group"/>
      <xs:element ref="recover"/>
    </xs:choice>
  </xs:group>

  <xs:group name="top-level-content">
    <xs:choice>
      <xs:group ref="group-content"/>
      <xs:element ref="namespace-alias"/>
    </xs:choice>
  </xs:group>

  <xs:group name="text-template-content">
    <xs:choice>
      <xs:element ref="text"/>
      <xs:element ref="cdata"/>
      <xs:element ref="value-of"/>
      <xs:element ref="if"/>
      <xs:element ref="else"/>
      <xs:element ref="choose"/>
    </xs:choice>
  </xs:group>

  <xs:group name="template-content">
    <xs:choice>
      <xs:any namespace="##other" processContents="lax"/>
      <xs:element ref="call-procedure"/>
      <xs:element ref="copy"/>
      <xs:element ref="element"/>
      <xs:element ref="start-element"/>
      <xs:element ref="end-element"/>
      <xs:element ref="attribute"/>
      <xs:element ref="comment"/>
      <xs:element ref="processing-instruction"/>
```

```
    <xs:element ref="variable"/>
    <xs:element ref="param"/>
    <xs:element ref="assign"/>
    <xs:element ref="buffer"/>
    <xs:element ref="result-buffer"/>
    <xs:element ref="result-document"/>
    <xs:element ref="for-each-item"/>
    <xs:element ref="while"/>
    <xs:element ref="message"/>
    <xs:element ref="text"/>
    <xs:element ref="process-attributes"/>
    <xs:element ref="process-buffer"/>
    <xs:element ref="process-children"/>
    <xs:element ref="process-document"/>
    <xs:element ref="process-self"/>
    <xs:element ref="process-siblings"/>
    <xs:element ref="analyze-text"/>
    <xs:group ref="text-template-content"/>
  </xs:choice>
</xs:group>

<xs:simpleType name="booleanAtt">
  <xs:restriction base="xs:NMTOKEN">
    <xs:enumeration value="yes"/>
    <xs:enumeration value="no"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="ns-prefix">
  <xs:union memberTypes="xs:NCName">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration value="#default"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:union>
</xs:simpleType>

<xs:element name="transform">
  <xs:complexType>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:group ref="top-level-content"/>
    </xs:choice>
    <xs:attribute name="version" type="xs:decimal" use="required"/>
    <xs:attribute name="pass-through" use="optional">
      <xs:simpleType>
        <xs:restriction base="xs:NMTOKEN">
          <xs:enumeration value="none"/>
          <xs:enumeration value="all"/>
          <xs:enumeration value="text"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="recognize-cdata" type="booleanAtt" use="optional" />
    <xs:attribute name="strip-space" type="booleanAtt" use="optional" />
    <xs:attribute name="stxpath-default-namespace" type="xs:anyURI"
                  use="optional"/>
    <xs:attribute name="output-encoding" type="xs:string" use="optional"/>
    <xs:attribute name="output-method" type="xs:QName" use="optional"/>
    <xs:attribute name="exclude-result-prefixes" use="optional">
```

```
      <xs:simpleType>
        <xs:union>
          <xs:simpleType>
            <xs:list itemType="ns-prefix"/>
          </xs:simpleType>
          <xs:simpleType>
            <xs:restriction base="xs:string">
              <xs:enumeration value="#all"/>
            </xs:restriction>
          </xs:simpleType>
        </xs:union>
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>
</xs:element>

<xs:element name="group">
  <xs:complexType>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:group ref="group-content"/>
    </xs:choice>
    <xs:attribute name="name" type="xs:QName" use="optional"/>
    <xs:attribute name="pass-through" use="optional" default="inherit">
      <xs:simpleType>
        <xs:restriction base="xs:NMTOKEN">
          <xs:enumeration value="none"/>
          <xs:enumeration value="all"/>
          <xs:enumeration value="text"/>
          <xs:enumeration value="inherit"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="recognize-cdata" use="optional" default="inherit">
      <xs:simpleType>
        <xs:restriction base="xs:NMTOKEN">
          <xs:enumeration value="yes"/>
          <xs:enumeration value="no"/>
          <xs:enumeration value="inherit"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="strip-space" use="optional" default="inherit">
      <xs:simpleType>
        <xs:restriction base="xs:NMTOKEN">
          <xs:enumeration value="yes"/>
          <xs:enumeration value="no"/>
          <xs:enumeration value="inherit"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>
</xs:element>

<xs:element name="include">
  <xs:complexType>
    <xs:attribute name="href" type="xs:anyURI" use="required"/>
  </xs:complexType>
</xs:element>

<xs:element name="namespace-alias">
```

```
  <xs:complexType>
    <xs:attribute name="sheet-prefix" type="ns-prefix"
                  use="optional" default="#default"/>
    <xs:attribute name="result-prefix" type="ns-prefix"
                  use="optional" default="#default"/>
  </xs:complexType>
</xs:element>

<xs:element name="template">
  <xs:complexType mixed="true">
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:group ref="template-content"/>
    </xs:choice>
    <xs:attribute name="match" type="xs:string" use="required"/>
    <xs:attribute name="priority" type="xs:decimal" use="optional"/>
    <xs:attributeGroup ref="templAtts"/>
  </xs:complexType>
</xs:element>

<xs:attributeGroup name="templAtts">
  <xs:attribute name="visibility" use="optional" default="local">
    <xs:simpleType>
      <xs:restriction base="xs:NMTOKEN">
        <xs:enumeration value="local"/>
        <xs:enumeration value="group"/>
        <xs:enumeration value="global"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
  <xs:attribute name="public" type="booleanAtt" use="optional" />
</xs:attributeGroup>

<xs:element name="procedure">
  <xs:complexType mixed="true">
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:group ref="template-content"/>
    </xs:choice>
    <xs:attributeGroup ref="templAtts"/>
    <xs:attribute name="name" type="xs:QName" use="required"/>
  </xs:complexType>
</xs:element>

<xs:element name="call-procedure">
  <xs:complexType>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element ref="with-param"/>
    </xs:choice>
    <xs:attribute name="name" type="xs:QName" use="required"/>
    <xs:attribute name="group" type="xs:QName" use="optional"/>
  </xs:complexType>
</xs:element>

<xs:element name="with-param">
  <xs:complexType mixed="true">
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:group ref="text-template-content"/>
    </xs:choice>
    <xs:attribute name="name" type="xs:QName" use="required"/>
    <xs:attribute name="select" type="xs:string" use="optional"/>
  </xs:complexType>
```

```
</xs:element>

<xs:element name="param">
  <xs:complexType mixed="true">
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:group ref="text-template-content"/>
    </xs:choice>
    <xs:attribute name="name" type="xs:QName" use="required"/>
    <xs:attribute name="select" type="xs:string" use="optional"/>
    <xs:attribute name="required" type="booleanAtt" use="optional" />
  </xs:complexType>
</xs:element>

<xs:element name="copy">
  <xs:complexType>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:group ref="template-content"/>
    </xs:choice>
    <xs:attribute name="attributes" type="xs:string" use="optional"/>
  </xs:complexType>
</xs:element>

<xs:attributeGroup name="filter-atts">
  <xs:attribute name="filter-method" type="xs:string" use="optional" />
  <xs:attribute name="filter-src" use="optional">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:pattern value="url\([ ]*'[^']+'[ ]*\)" />
        <xs:pattern value="url\([ ]*&quot;[^&quot;]+&quot;[ ]*\)" />
        <xs:pattern value="url\([ ]*[^&quot;']+[ ]*\)" />
        <xs:pattern value="buffer\([ ]*.+[ ]*\)" />
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
</xs:attributeGroup>

<xs:element name="process-children">
  <xs:complexType>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element ref="with-param"/>
    </xs:choice>
    <xs:attribute name="group" type="xs:QName" use="optional"/>
    <xs:attributeGroup ref="filter-atts"/>
  </xs:complexType>
</xs:element>

<xs:element name="process-attributes">
  <xs:complexType>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element ref="with-param"/>
    </xs:choice>
    <xs:attribute name="group" type="xs:QName" use="optional"/>
  </xs:complexType>
</xs:element>

<xs:element name="process-siblings">
  <xs:complexType>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element ref="with-param"/>
    </xs:choice>
```

```
    <xs:attribute name="group" type="xs:QName" use="optional"/>
    <xs:attribute name="while" type="xs:string" use="optional"
                  default="node()"/>
    <xs:attribute name="until" type="xs:string" use="optional"
                  default="node()[false()]"/>
    <xs:attributeGroup ref="filter-atts"/>
  </xs:complexType>
</xs:element>

<xs:element name="process-self">
  <xs:complexType>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element ref="with-param"/>
    </xs:choice>
    <xs:attribute name="group" type="xs:QName" use="optional"/>
    <xs:attributeGroup ref="filter-atts"/>
  </xs:complexType>
</xs:element>

<xs:element name="analyze-text">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="match" maxOccurs="unbounded">
        <xs:complexType>
          <xs:choice minOccurs="0" maxOccurs="unbounded">
            <xs:group ref="template-content" />
          </xs:choice>
          <xs:attribute name="regex" type="xs:string" use="required"/>
          <xs:attribute name="case" use="optional" default="sensitive">
            <xs:simpleType>
              <xs:restriction base="xs:NMTOKEN">
                <xs:enumeration value="sensitive"/>
                <xs:enumeration value="insensitive"/>
              </xs:restriction>
            </xs:simpleType>
          </xs:attribute>
        </xs:complexType>
      </xs:element>
      <xs:element name="no-match" minOccurs="0">
        <xs:complexType>
          <xs:choice minOccurs="0" maxOccurs="unbounded">
            <xs:group ref="template-content"/>
          </xs:choice>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="select" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>

<xs:element name="value-of">
  <xs:complexType>
    <xs:attribute name="select" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>

<xs:element name="text">
  <xs:complexType mixed="true">
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:group ref="template-content"/>
```

```
    </xs:choice>
    <xs:attribute name="markup" use="optional" default="error">
      <xs:simpleType>
        <xs:restriction base="xs:NMTOKEN">
          <xs:enumeration value="error"/>
          <xs:enumeration value="ignore"/>
          <xs:enumeration value="serialize"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>
</xs:element>

<xs:element name="cdata">
  <xs:complexType mixed="true">
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:group ref="text-template-content"/>
    </xs:choice>
  </xs:complexType>
</xs:element>

<xs:element name="element">
  <xs:complexType>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:group ref="template-content"/>
    </xs:choice>
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="namespace" type="xs:string" use="optional"/>
  </xs:complexType>
</xs:element>

<xs:element name="start-element">
  <xs:complexType>
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="namespace" type="xs:string" use="optional"/>
  </xs:complexType>
</xs:element>

<xs:element name="end-element">
  <xs:complexType>
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="namespace" type="xs:string" use="optional"/>
  </xs:complexType>
</xs:element>

<xs:element name="attribute">
  <xs:complexType mixed="true">
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:group ref="text-template-content"/>
    </xs:choice>
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="namespace" type="xs:string" use="optional"/>
    <xs:attribute name="select" type="xs:string" use="optional"/>
  </xs:complexType>
</xs:element>

<xs:element name="processing-instruction">
  <xs:complexType mixed="true">
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:group ref="text-template-content"/>
```

```
    </xs:choice>
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="select" type="xs:string" use="optional"/>
  </xs:complexType>
</xs:element>

<xs:element name="comment">
  <xs:complexType mixed="true">
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:group ref="text-template-content"/>
    </xs:choice>
    <xs:attribute name="select" type="xs:string" use="optional"/>
  </xs:complexType>
</xs:element>

<xs:element name="if">
  <xs:complexType mixed="true">
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:group ref="template-content"/>
    </xs:choice>
    <xs:attribute name="test" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>

<xs:element name="else">
  <xs:complexType>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:group ref="template-content"/>
    </xs:choice>
  </xs:complexType>
</xs:element>

<xs:element name="choose">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="when" maxOccurs="unbounded">
        <xs:complexType>
          <xs:choice minOccurs="0" maxOccurs="unbounded">
            <xs:group ref="template-content"/>
          </xs:choice>
          <xs:attribute name="test" type="xs:string" use="required"/>
        </xs:complexType>
      </xs:element>
      <xs:element name="otherwise" minOccurs="0">
        <xs:complexType>
          <xs:choice minOccurs="0" maxOccurs="unbounded">
            <xs:group ref="template-content"/>
          </xs:choice>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="for-each-item">
  <xs:complexType>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:group ref="template-content"/>
    </xs:choice>
    <xs:attribute name="name" type="xs:QName" use="required"/>
```

```
    <xs:attribute name="select" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>

<xs:element name="while">
  <xs:complexType>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:group ref="template-content"/>
    </xs:choice>
    <xs:attribute name="test" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>

<xs:element name="process-document">
  <xs:complexType>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element ref="with-param"/>
    </xs:choice>
    <xs:attribute name="href" type="xs:string" use="required"/>
    <xs:attribute name="base" type="uri-reference" use="optional"/>
    <xs:attribute name="group" type="xs:QName" use="optional"/>
    <xs:attributeGroup ref="filter-atts"/>
  </xs:complexType>
</xs:element>

<xs:simpleType name="uri-reference">
  <xs:union memberTypes="xs:string">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration value="#input"/>
        <xs:enumeration value="#sheet"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:union>
</xs:simpleType>

<xs:element name="result-document">
  <xs:complexType>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:group ref="template-content"/>
    </xs:choice>
    <xs:attribute name="href" type="xs:string" use="required"/>
    <xs:attribute name="output-encoding" type="xs:string" use="optional"/>
    <xs:attribute name="output-method" type="xs:QName" use="optional"/>
  </xs:complexType>
</xs:element>

<xs:element name="buffer">
  <xs:complexType>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:group ref="template-content"/>
    </xs:choice>
    <xs:attribute name="name" type="xs:QName" use="required"/>
  </xs:complexType>
</xs:element>

<xs:element name="result-buffer">
  <xs:complexType>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:group ref="template-content"/>
```

```
    </xs:choice>
    <xs:attribute name="name" type="xs:QName" use="required"/>
    <xs:attribute name="clear" type="booleanAtt" use="optional"
                  default="no" />
  </xs:complexType>
</xs:element>

<xs:element name="process-buffer">
  <xs:complexType>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element ref="with-param"/>
    </xs:choice>
    <xs:attribute name="name" type="xs:QName" use="required"/>
    <xs:attribute name="group" type="xs:QName" use="optional"/>
    <xs:attributeGroup ref="filter-atts"/>
  </xs:complexType>
</xs:element>

<xs:element name="message">
  <xs:complexType mixed="true">
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:group ref="template-content"/>
    </xs:choice>
    <xs:attribute name="select" type="xs:string" use="optional"/>
    <xs:attribute name="terminate" type="booleanAtt" use="optional"/>
    <xs:attribute name="logger" type="xs:string" use="optional" />
    <xs:attribute name="level" use="optional">
      <xs:simpleType>
        <xs:restriction base="xs:NMTOKEN">
          <xs:enumaration value="trace"/>
          <xs:enumaration value="debug"/>
          <xs:enumaration value="info"/>
          <xs:enumaration value="warn"/>
          <xs:enumaration value="error"/>
          <xs:enumaration value="fatal"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>
</xs:element>

<xs:element name="recover">
  <xs:complexType mixed="true">
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:group ref="template-content"/>
    </xs:choice>
    <xs:attributeGroup ref="templAtts"/>
    <xs:attribute name="from" type="xs:QName" use="optional"/>
  </xs:complexType>
</xs:element>

<xs:element name="variable">
  <xs:complexType mixed="true">
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:group ref="text-template-content"/>
    </xs:choice>
    <xs:attribute name="name" type="xs:QName" use="required"/>
    <xs:attribute name="select" type="xs:string" use="optional"/>
  </xs:complexType>
</xs:element>
```

```
  <xs:element name="assign">
    <xs:complexType mixed="true">
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:group ref="text-template-content"/>
      </xs:choice>
      <xs:attribute name="name" type="xs:QName" use="required"/>
      <xs:attribute name="select" type="xs:string" use="optional"/>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

Anhang B
Grammatik

B.1 Modifikation von EBNF-Grammatiken

Dieses Kapitel beschreibt, welche Änderungen in einer gegebenen EBNF-Grammatik[2] vorgenommen werden können, sodass die resultierende Grammatik eine Teilmenge der ursprünglich beschriebenen Sprache erzeugt. Auf diese Weise lässt sich aus den EBNF-Produktionsregeln für XPath 2.0 die Teilsprache XPath_S konstruieren (siehe Kapitel 5.4.1), welche schließlich zu STXPath ergänzt wird.

Zunächst werden die hier verwendeten Begriffe kurz eingeführt.

Formale Sprachen

Eine formale Sprache wird durch eine Grammatik $G = (A, M, s, P)$ beschrieben, wobei A das verwendete Alphabet (Terminalsymbole), M die Menge der Metasymbole (Nichtterminalsymbole), s das Startsymbol mit $s \in M$ und schließlich P die Menge der Produktionsregeln mit $P \subseteq M \times (A \cup M)^*$ sei. Häufig sind A, M und s nur implizit gegeben und ergeben sich dann aus der Notation der Produktionsregeln in P.

Eine Produktionsregel definiert einen Ableitungsschritt, bei dem ein in einer Symbolfolge enthaltenes Metasymbol $l \in M$ durch eine Folge von Ersetzungssymbolen r_1 bis r_n mit $r_i \in (A \cup M)$ ersetzt werden kann. Solche Produktionsregeln lassen sich in Backus-Naur-Form (BNF) [Kas90] folgendermaßen notieren:

$$l \rightarrow r_1 \ldots r_n$$

Die Ausführung mehrerer Ableitungsschritte nacheinander nennt man eine Ableitung. Eine durch eine Grammatik G definierte Sprache $L(G)$ ist die Menge der Zeichenfolgen des Alphabets A, die sich aus dem Startsymbol s unter Anwendung der Produktionsregeln in P ableiten lässt.

Teilmengenbeziehungen von Sprachen

Der Umfang einer durch eine Grammatik definierten Sprache wird in erster Linie durch die gegebenen Produktionsregeln bestimmt. Wenn Q ebenfalls eine Menge von Produktionsregeln mit $Q \subseteq P$ ist, so gilt $L(A, M, s, Q) \subseteq L(A, M, s, P)$. Dies ergibt sich daraus, dass jede Zeichenfolge, die durch Anwendung von Regeln aus Q abgeleitet wurde, durch die gleichen Regeln aus P abgeleitet werden kann. Durch das Hinzufügen zusätzlicher Produktionsregeln wird eine Sprache damit im Allgemeinen erweitert, durch das Streichen von Produktionsregeln wird sie im Allgemeinen eingeschränkt.

Redundante Produktionsregeln

Eine Produktionsregel ist *redundant*, wenn jede Zeichenfolge der definierten Sprache auch ohne Anwendung dieser Regel abgeleitet werden kann. Damit kann eine redundante Produktionsregel aus der Menge P entfernt werden, ohne dass sich die definierte Sprache ändert. Umgekehrt gilt, dass das Hinzufügen einer redundanten Regel die Sprache nicht erweitert.

[2] EBNF = Erweiterte Backus-Naur-Form

Eine redundante Produktionsregel entsteht, wenn das Ergebnis einer Ableitungsfolge als zusätzliche Produktionsregel in die Grammatik aufgenommen wird.

Sei also m ein Metasymbol, das durch mehrere Ableitungsschritte unter Verwendung von Produktionsregeln in P durch die Folge der Symbole r_1 bis r_n ersetzt wurde. Die Aufnahme der zusätzlichen (und redundanten) Produktionsregel q der Form

$$m \rightarrow r_1 \ldots r_n$$

in die bestehende Grammatik ändert somit nichts an der definierten Sprache, d.h. es gilt $L(A, M, s, P) = L(A, M, s, P \cup \{q\})$.

Erweiterte Backus-Naur-Form

Die Erweiterte Backus-Naur-Form (EBNF) ermöglicht eine kompaktere Schreibweise für die Produktionsregeln P einer Grammatik G. Die hier verwendete EBNF benutzt als weitere Notationssymbole u.a. das Zeichen | zur Angabe von Alternativen sowie ?, + und *, die das optionale, mehrfache bzw. optional mehrfache Auftreten des so gekennzeichneten Symbols oder geklammerten Ausdrucks kennzeichnen. Zur Unterscheidung von BNF-Produktionsregeln benutzt EBNF darüber hinaus das Symbol ::= anstelle des BNF-Symbols →. Eine vollständige Definition der hier verwendeten EBNF-Notation enthält Kapitel 6 der XML-Spezifikation [W3C04a]. Jede EBNF-Produktionsregel kann in eine äquivalente Menge von BNF-Produktionsregeln umgeformt werden. Gegebenenfalls ist dafür die Verwendung zusätzlicher Metasymbole notwendig.

Im Folgenden werden drei Spezialfälle behandelt, bei denen die Änderung einer EBNF-Produktionsregel zu einer Teilmenge der abgeleiteten Sprache führt.

Verringerung der Alternativen

Mehrere BNF-Produktionsregeln mit der gleichen linken Seite können in EBNF als eine Produktionsregel notiert werden, wobei alle rechten Seiten dieser BFN-Produktionsregeln in der EBNF-Produktionsregel als Alternativen aufgeführt werden. In der EBNF werden diese Alternativen durch das Zeichen | voneinander getrennt.

Es gilt: *Wird auf der rechten Seite einer EBNF-Produktionsregel eine der möglichen Alternativen entfernt, ist die dadurch definierte Sprache eine Teilmenge der ursprünglichen Sprache.*

Betrachten wir dazu die beiden Produktionsregeln

$$l ::= a_1 \mid \ldots \mid a_n$$

und

$$l' ::= a_1 \mid \ldots \mid a_{i-1} \mid a_{i+1} \mid \ldots \mid a_n$$

Jede Alternative a_i steht für eine Folge von Symbolen aus $A \cup M$. Die Produktionsregel für l' entsteht durch Entfernen der Alternative a_i aus der Produktionsregel für l. Da sich die EBNF-Produktionsregel für l in eine Menge von n BNF-Produktionsregeln der Form

$$l \rightarrow a_1$$
$$\ldots$$
$$l \rightarrow a_n$$

umformen lässt, ist leicht zu sehen, dass die äquivalenten BNF-Produktionsregeln für l' genau eine Regel weniger enthalten, nämlich die mit der rechten Seite a_i.

Das Entfernen einer Alternative in der rechten Seite einer EBNF-Produktionsregel führt damit zu einer Grammatik, die eine Teilmenge der ursprünglichen Sprache definiert.

Einschränkung der Häufigkeiten

Es gilt: *Wird auf einer rechten Seite einer Produktionsregel der Stern-Operator durch den Fragezeichen-Operator ersetzt, definiert die neue Grammatik eine Teilmenge der ursprünglichen Sprache.*

Der Stern-Operator gibt an, dass sein Operand, mehrfach, einmal oder auch überhaupt nicht bei Anwendung der Produktionsregeln in der Ersetzung auftreten kann. Der Fragezeichen-Operator gibt an, dass sein Operand optional ist, also einmal oder überhaupt nicht auftreten kann.

Es ist sehr leicht einzusehen, dass eine EBNF-Regel der Form

$l ::= r_1 \ldots r_i^* \ldots r_n$

in die folgende Menge von BNF-Regeln umgewandelt werden kann:[3]

$l \rightarrow r_1 \ldots r_{i-1}\ r'\ r_{i+1} \ldots r_n$
$r' \rightarrow r'\ r_i$
$r' \rightarrow r_i$
$r' \rightarrow \varepsilon$

Die vorletzte Regel ist offenbar redundant. Ihr Sinn wird deutlich, wenn wir ganz analog eine EBNF-Produktionsregel mit Fragezeichen-Operator der Form

$l ::= r_1 \ldots r_i? \ldots r_n$

folgendermaßen äquivalent in BNF ausdrücken:

$l \rightarrow r_1 \ldots r_{i-1}\ r'\ r_{i+1} \ldots r_n$
$r' \rightarrow r_i$
$r' \rightarrow \varepsilon$

Die zweite Menge enthält genau eine Produktionsregel weniger. Das Ersetzen eines Stern-Operators durch einen Fragezeichen-Operator in einer EBNF-Produktionsregel führt damit zu einer Grammatik, die eine Teilmenge der ursprünglichen Sprache beschreibt.

Es gilt: *Wird auf einer rechten Seite einer EBNF-Produktionsregel ein optionaler Bestandteil (gekennzeichnet durch den Fragezeichen-Operator) weggelassen, definiert die neue Grammatik eine Teilmenge der ursprünglichen Sprache.*

Der Beweis dieser Aussage lässt sich analog zur vorherigen führen. Ein fehlendes Symbol auf der rechten Seite einer Produktionsregel lässt sich dabei äquivalent durch eine zusätzliche leere Produktionsregel ausdrücken. Die folgende Produktionsregel, die das Symbol r_i nicht enthält

$l ::= r_1 \ldots r_{i-1}\ r_{i+1} \ldots r_n$

entspricht damit den beiden Produktionsregeln

$l \rightarrow r_1 \ldots r_{i-1}\ r'\ r_{i+1} \ldots r_n$
$r' \rightarrow \varepsilon$

Offensichtlich sind diese beiden Produktionsregeln auch in der Menge der BNF-Produktionsregeln enthalten, die oben bereits beim Umschreiben der EBNF-Produktionsregel mit $r_i?$ in der rechten Seite entstanden waren.

[3] Das Symbol r' ist ein zusätzliches Metasymbol, das in keiner anderen Produktionsregel als den hier angegebenen vorkommt. Durch die linksrekursive BNF-Produktionsregel $r' \rightarrow r'\ r_i$ wird eine mögliche Wiederholung des Symbols r_i erreicht. Völlig gleichbedeutend für die beschriebene Sprache wäre eine rechtsrekursive Regel mit der rechten Seite $r_i\ r'$. Das Symbol ε in der letzten Produktionsregel steht für die leere Zeichenfolge.

Die Kombination der beiden letzten Aussagen ergibt, dass das Weglassen eines Symbols mit Stern-Operator auf der rechten Seite einer Produktionsregel ebenfalls zu einer Teilmenge der ursprünglich definierten Sprache führt.

B.2 STXPath-Grammatik

Die folgenden Produktionsregeln geben die Grammatik für STXPath-Ausdrücke und -Muster wieder. Die Originalregeln sind dem Entwurf der XPath-2.0-Spezifikation [W3C03a] sowie dem Entwurf der XSLT-2.0-Spezifikation [W3C03b] (ab Regel 81) entnommen. Die einzelnen Änderungen wurden im Kapitel 5.4.1 besprochen.

Die Modifikationen werden durch die folgende Notation veranschaulicht:

Entfernte Bestandteile
werden in grauer Schrift und durchgestrichen dargestellt: ~~Beispiel~~

Hinzugefügte Bestandteile
werden unterstrichen dargestellt: Beispiel

Vollständig neue Produktionsregeln werden durch zusätzliche kleine Buchstaben fortnummeriert. In dieser Grammatik betrifft das die Regeln 78a und 78b.

Für *Nichtterminalsymbole* und "`Terminalsymbole`" wird jeweils eine eigene Schriftart verwendet.

Named Terminals

[1] ExprComment ::=
"(:" (ExprCommentContent | ExprComment)* ":)"

[2] ExprCommentContent ::= Char

[3] IntegerLiteral ::= Digits

[4] DecimalLiteral ::= ("." Digits) | (Digits "." [0-9]*)

[5] DoubleLiteral ::=
(("." Digits) | (Digits ("." [0-9]*)?)) ("e" | "E") ("+" | "-")? Digits

[6] StringLiteral ::=
('"' (('"' '"') | [^"])* '"') | ("'" (("'" "'") | [^'])* "'")

~~[7] SchemaGlobalTypeName ::= "type" "(" QName ")"~~

~~[8] SchemaGlobalContext ::= QName | SchemaGlobalTypeName~~

~~[9] SchemaContextStep ::= QName~~

[10] Digits ::= [0-9]+

[11] NCName ::=
[http://www.w3.org/TR/REC-xml-names/#NT-NCName]*Names*

[12] VarName ::= QName

[13] QName ::=
[http://www.w3.org/TR/REC-xml-names/#NT-QName]*Names*

[14] Char ::= [http://www.w3.org/TR/REC-xml#NT-Char]*XML*

Non-Terminals

[15] STXPath ::= Expr?

[16] Expr ::= ExprSingle (`","` ExprSingle)*

[17] ExprSingle ::= ForExpr | QuantifiedExpr | IfExpr | OrExpr

[18] ForExpr ::= SimpleForClause `"return"` ExprSingle

[19] SimpleForClause ::=
`"for"` `"$"` VarName `"in"` ExprSingle (`","` `"$"` VarName `"in"` ExprSingle)*

[20] QuantifiedExpr ::=
(`"some"` `"$"` | `"every"` `"$"`) VarName `"in"` ExprSingle (`","` `"$"` VarName `"in"` ExprSingle)* `"satisfies"` ExprSingle

[21] IfExpr ::=
`"if"` `"("` Expr `")"` `"then"` ExprSingle `"else"` ExprSingle

[22] OrExpr ::= AndExpr (`"or"` AndExpr)*

[23] AndExpr ::= InstanceofExpr (`"and"` InstanceofExpr)*

[24] InstanceofExpr ::=
TreatExpr (`"instance"` `"of"` SequenceType)?

[25] TreatExpr ::=
CastableExpr (`"treat"` `"as"` SequenceType)?

[26] CastableExpr ::= CastExpr (`"castable"` `"as"` SingleType)?

[27] CastExpr ::= ComparisonExpr (`"cast"` `"as"` SingleType)?

[28] ComparisonExpr ::=
RangeExpr ((ValueComp | GeneralComp | NodeComp) RangeExpr)?

[29] RangeExpr ::= AdditiveExpr (`"to"` AdditiveExpr)?

[30] AdditiveExpr ::=
MultiplicativeExpr ((`"+"` | `"-"`) MultiplicativeExpr)*

[31] MultiplicativeExpr ::=
UnaryExpr ((`"*"` | `"div"` | `"idiv"` | `"mod"`) UnaryExpr)*

[32] UnaryExpr ::= (`"-"` | `"+"`)* UnionExpr

[33] UnionExpr ::=
IntersectExceptExpr ((`"union"` | `"|"`) IntersectExceptExpr)*

[34] IntersectExceptExpr ::=
ValueExpr ((`"intersect"` | `"except"`) ValueExpr)*

[35] ValueExpr ::= PathExpr | FilterStep

[36] PathExpr ::= `"/"` RelativePathExpr?
| `"//"` RelativePathExpr
| RelativePathExpr

[37] RelativePathExpr ::= StepExpr ((`"/"` | `"//"`) StepExpr)*

[38] StepExpr ::= AxisStep | FilterStep

[39] AxisStep ::= (ForwardStep | ReverseStep) Predicate

[40] FilterStep ::= PrimaryExpr Predicate*

[41] ContextItemExpr ::= `"."`

[42] PrimaryExpr ::= Literal | VarRef | ParenthesizedExpr | ContextItemExpr | FunctionCall

[43] VarRef ::= "$" VarName

[44] Predicate ::= ("[" Expr "]")?

[45] GeneralComp ::= "=" | "!=" | "<" | "<=" | ">" | ">="

[46] ValueComp ::= "eq" | "ne" | "lt" | "le" | "gt" | "ge"

[47] NodeComp ::= "is" | "<<" | ">>"

[48] ForwardStep ::= (ForwardAxis NodeTest) | AbbrevForwardStep

[49] ReverseStep ::= (ReverseAxis NodeTest) | AbbrevReverseStep

[50] AbbrevForwardStep ::= "@"? NodeTest

[51] AbbrevReverseStep ::= ".."

[52] ForwardAxis ::= "child" "::" | "descendant" "::" | "attribute" "::" | "self" "::" | "descendant-or-self" "::" | "following-sibling" "::" | "following" "::" | "namespace" "::"

[53] ReverseAxis ::= "parent" "::" | "ancestor" "::" | "preceding-sibling" "::" | "preceding" "::" | "ancestor-or-self" "::"

[54] NodeTest ::= KindTest | NameTest

[55] NameTest ::= QName | Wildcard

[56] Wildcard ::= "*" | NCName ":" "*" | "*" ":" NCName

[57] Literal ::= NumericLiteral | StringLiteral

[58] NumericLiteral ::= IntegerLiteral | DecimalLiteral | DoubleLiteral

[59] ParenthesizedExpr ::= "(" Expr? ")"

[60] FunctionCall ::= QName "(" (ExprSingle ("," ExprSingle)*)? ")"

[61] SingleType ::= AtomicType "?"?

[62] SequenceType ::= (ItemType OccurrenceIndicator?) | "empty" "(" ")"

[63] AtomicType ::= QName

[64] ItemType ::= AtomicType | KindTest | "item" "(" ")"

[65] KindTest ::= DocumentTest | ElementTest | AttributeTest | PITest | CommentTest | TextTest | AnyKindTest | CdataTest | DoctypeTest

[66] ElementTest ::= "element" "(" ((SchemaContextPath ElementName) | (ElementNameOrWildcard ("," TypeNameOrWildcard "nillable"?)?))? ")"

[67] AttributeTest ::= "attribute" "(" ((SchemaContextPath AttributeName) | (AttribNameOrWildcard ("," TypeNameOrWildcard)?))? ")"

[68] ElementName ::= QName

[69] AttributeName ::= QName

[70] ~~TypeName ::= QName~~

[71] ~~ElementNameOrWildcard ::= ElementName | "*"~~

[72] ~~AttribNameOrWildcard ::= AttributeName | "*"~~

[73] ~~TypeNameOrWildcard ::= TypeName | "*"~~

[74] PITest ::=
"processing-instruction" "(" (NCName | StringLiteral)? ")"

[75] ~~DocumentTest ::= "document-node" "(" ElementTest? ")"~~

[76] CommentTest ::= "comment" "(" ")"

[77] TextTest ::= "text" "(" ")"

[78] AnyKindTest ::= "node" "(" ")"

[78a] CdataTest ::= "cdata" "(" ")"

[78b] DoctypeTest ::= "doctype" "(" ")"

[79] ~~SchemaContextPath ::=
SchemaGlobalContext "/" SchemaContextStep "/"*~~

[80] ~~OccurrenceIndicator ::= "?" | "*" | "+"~~

Patterns

[81] Pattern ::= PathPattern
| Pattern '|' PathPattern

[82] PathPattern ::= RelativePathPattern
| '/' RelativePathPattern?
~~| '//' RelativePathPattern~~
~~| IdKeyPattern (('/' | '//') RelativePathPattern)?~~

[83] RelativePathPattern ::=
PatternStep (('/' | '//') RelativePathPattern)?

[84] PatternStep ::= PatternAxis? NodeTest Predicate

[85] PatternAxis ::= ~~('child' '::' | 'attribute' '::' |~~ '@'~~)~~

[86] ~~IdKeyPattern ::= 'id' '(' IdValue ')'
| 'key' '(' StringLiteral ',' KeyValue ')'~~

[87] ~~IdValue ::= StringLiteral | VarRef~~

[88] ~~KeyValue ::= Literal | VarRef~~

Anhang C
Quellcode der Fallbeispiele

C.1 Simulation einer Turing-Maschine

```
<?xml version="1.0"?>
<!--
 | This STX transformation runs the Turing machine (TM) that is
 | specified by the source document. The initial tape for the
 | Turing machine is specified by a global parameter named 'tape'.
 | (Thus, this STX transformation is a Universal Turing Machine.)
 |
 | The source document, which specifies a Turing machine,
 | is an XML document that conforms to the
 | Turing Machine Markup Language (TMML), see
 | http://www.unidex.com/turing/tmml.htm
 +-->
<stx:transform xmlns:stx="http://stx.sourceforge.net/2002/ns"
               version="1.0"
               pass-through="none" strip-space="yes">

  <!-- initial tape (input) -->
  <stx:param name="tape" />

  <!-- initial position on the tape -->
  <stx:variable name="pos" select="1" />

  <!-- the set of known symbols -->
  <stx:variable name="symbols" />

  <!-- the current symbol -->
  <stx:variable name="symbol" select="substring($tape, $pos, 1)" />

  <!-- the blank symbol -->
  <stx:variable name="blank" select="' '" />

  <!-- the current state of the turing machine -->
  <stx:variable name="state" />

  <!-- halt condition -->
  <stx:variable name="run" select="true()" />



  <!-- ************************ main ************************ -->

  <stx:template match="turing-machine">
    <stx:if test="$tape = ''">
      <stx:text>Missing input tape&#xA;</stx:text>
      <stx:assign name="run" select="false()" />
    </stx:if>
    <!-- a buffer which contains the instructions of our turing machine -->
    <stx:buffer name="turing">
      <stx:process-self group="store"/>
    </stx:buffer>
```

```
  <stx:while test="$run">
    <stx:process-buffer name="turing" group="process" />
  </stx:while>
  <stx:value-of select="concat('Result: ', $tape, '&#xA;')" />
</stx:template>


<!-- ********************** storing ********************** -->

<stx:group name="store" pass-through="all">

  <stx:template match="symbols">
    <stx:assign name="symbols" select="." />
    <stx:if test="@blank-symbol">
      <stx:assign name="blank" select="@blank-symbol" />
    </stx:if>
  </stx:template>

  <stx:template match="state">
    <stx:if test="@start = 'yes'">
      <stx:assign name="state" select="." />
    </stx:if>
    <stx:process-self />
  </stx:template>

</stx:group>


<!-- ********************** processing ********************** -->

<stx:group name="process">

  <!-- indicates whether the next state of the TM has been found -->
  <stx:variable name="found" />

  <stx:template match="turing-machine">
    <!-- test for error conditions -->
    <stx:if test="not(contains($symbols, $symbol)) and $symbol != $blank">
      <stx:text>Found unknown symbol: "</stx:text>
      <stx:value-of select="$symbol" />
      <stx:text>"&#xA;</stx:text>
      <stx:assign name="run" select="false()" />
    </stx:if>
    <stx:assign name="found" select="'no'" />
    <stx:process-children />
    <stx:if test="$found = 'no'">
      <stx:text>No action found for symbol "</stx:text>
      <stx:value-of select="$symbol" />
      <stx:text>" and state "</stx:text>
      <stx:value-of select="$state" />
      <stx:text>"&#xA;</stx:text>
      <stx:assign name="run" select="false()" />
    </stx:if>
  </stx:template>

  <stx:template match="state">
    <stx:if test=". = $state and @halt = 'yes'">
      <stx:assign name="run" select="false()" />
      <stx:assign name="found" select="'done'" />
    </stx:if>
```

```
    </stx:template>

    <stx:template match="mapping">
      <stx:if test="$found = 'no'">
        <stx:process-children />
      </stx:if>
    </stx:template>

    <stx:template match="from[@current-state = $state and
                              @current-symbol = $symbol]">
      <stx:assign name="found" select="'yes'" />
    </stx:template>

    <stx:template match="to[$found='yes']">
      <stx:assign name="state" select="@next-state" />
      <stx:assign name="tape" select="concat(substring($tape, 1, $pos - 1),
                                             @next-symbol,
                                             substring($tape, $pos + 1))" />
      <stx:if test="@movement = 'right'">
        <stx:assign name="pos" select="$pos + 1" />
        <stx:assign name="symbol" select="substring($tape, $pos, 1)" />
        <stx:if test="$symbol = ''">
          <stx:assign name="symbol" select="$blank" />
          <stx:assign name="tape" select="concat($tape, $blank)" />
        </stx:if>
      </stx:if>
      <stx:if test="@movement = 'left'">
        <stx:assign name="pos" select="$pos - 1" />
        <stx:assign name="symbol" select="substring($tape, $pos, 1)" />
        <stx:if test="$symbol = ''">
          <stx:assign name="symbol" select="$blank" />
          <stx:assign name="pos" select="1" />
          <stx:assign name="tape" select="concat($blank, $tape)" />
        </stx:if>
      </stx:if>
      <stx:assign name="found" select="'done'" />
    </stx:template>

  </stx:group>

</stx:transform>
```

C.2 Verarbeitung der Daten des Open Directory

```
<?xml version="1.0"?>
<!--
 | This transformation creates HTML pages from the RDF content dump
 | content.rdf.u8 of the Open Directory.
 +-->
<stx:transform xmlns:stx="http://stx.sourceforge.net/2002/ns"
               xmlns:r="http://www.w3.org/TR/RDF/"
               xmlns:d="http://purl.org/dc/elements/1.0/"
               xmlns:od="http://dmoz.org/rdf"
               xmlns="http://www.w3.org/1999/xhtml"
               version="1.0"
               exclude-result-prefixes="#all" strip-space="yes">

  <stx:param name="css" select="'dmoz.css'" />
```

```
<stx:variable name="subtopics" select="true()"/>

<stx:template match="od:RDF">
  <html>
    <head>
      <link rel="stylesheet" type="text/css" href="{$css}" />
      <title>The Open Directory</title>
    </head>
    <body>
      <h2>The Open Directory</h2>
      <p>
        This page was created from the RDF content dump (see
        <a href="http://rdf.dmoz.org/">http://rdf.dmoz.org/</a>).
      </p>
      <ul class="topics">
        <stx:process-children />
      </ul>
      <hr />
      <p class="foot">Powered by STX</p>
    </body>
  </html>
</stx:template>

<stx:variable name="has-pages" />
<stx:variable name="catid" />

<stx:template match="od:Topic">
  <stx:param name="base" />
  <stx:param name="ancestors" select="'.'" />
  <stx:param name="cssfile" select="$css" />

  <stx:variable name="eid" select="escape-uri(@r:id,false())" />
  <stx:message>
    <stx:value-of select="$eid" />
  </stx:message>

  <stx:variable name="filename"
                select="substring-after(@r:id, $base)" />
  <stx:variable name="efilename"
                select="translate(escape-uri($filename, false()),
                                  '%', '')" />

  <stx:if test="not($subtopics)">
    <stx:start-element name="ul" />
    <stx:attribute name="class" select="'topics'" />
  </stx:if>

  <li>
    <a href="{translate(escape-uri($ancestors, false()),
                        '%', '')}/{$efilename}.html">
      <stx:value-of select="$filename" />
    </a>
  </li>

  <stx:result-document href="{concat(translate($eid, '%', ''), '.html')}">
    <html>
      <head>
        <title>
          <stx:text>DMOZ: </stx:text>
```

```
    <stx:value-of select="@r:id" />
  </title>
  <link rel="stylesheet" style="text/css" href="{$cssfile}" />
</head>
<body>
  <h3>
    <stx:variable name="len" select="count($ancestors)" />
    <stx:variable name="cnt" select="$len" />

    <stx:for-each-item name="dummy" select="$ancestors">
      <stx:variable name="item"
                    select="item-at($ancestors, $cnt)" />
      <stx:if test="$len = $cnt">
        <a href="{$item}/index.html">DMOZ</a>
        <stx:text> | </stx:text>
      </stx:if>
      <stx:else>
        <a href="../{translate(escape-uri($item, false()),
                               '%', '')}.html">
          <stx:value-of select="substring($item, $cnt*3-2)" />
        </a>: <stx:text />
      </stx:else>
      <stx:assign name="cnt" select="$cnt - 1" />
    </stx:for-each-item>
    <stx:value-of select="$filename" />
  </h3>

  <stx:assign name="has-pages" select="false()" />
  <stx:process-children group="t" />
  <p class="cat">Category id: <stx:value-of select="$catid" /></p>
  <p>All of the following data are used under the
    <a href="http://rdf.dmoz.org/license.html">Open Directory
      License</a>.</p>
  <stx:if test="$has-pages">
    <dl class="pages">
      <stx:process-siblings while="od:ExternalPage | comment()" />
    </dl>
    <hr />
  </stx:if>

  <stx:assign name="subtopics" select="false()" />
  <stx:variable name="id" select="concat(@r:id, '/')" />
  <stx:variable name="apath" select="()" />
  <stx:for-each-item name="a" select="$ancestors">
    <stx:assign name="apath"
                select="($apath, concat('../', $a))" />
  </stx:for-each-item>
  <stx:process-siblings
       until="od:Topic[not(starts-with(@r:id, $id))]">
    <stx:with-param name="base" select="$id" />
    <stx:with-param name="ancestors" select="($filename, $apath)" />
    <stx:with-param name="cssfile"
                    select="concat('../', $cssfile)" />
  </stx:process-siblings>
  <stx:if test="$subtopics">
    <stx:end-element name="ul" />
    <hr />
  </stx:if>

  <div align="center">
```

```
          <!-- Attribution copied from http://dmoz.org/become_an_editor/ -->
<p><table border="0" bgcolor="#336600" cellpadding="3" cellspacing="0">
<tr>
<td>
    <table width="100%" cellpadding="2" cellspacing="0" border="0">
        <tr align="center">
        <td><font face="sans-serif, Arial, Helvetica" size="2"
            color="#FFFFFF">Help build the largest human-edited
            directory on the web.</font></td></tr>
        <tr bgcolor="#CCCCCC" align="center">
        <td><font face="sans-serif, Arial, Helvetica" size="2"> <a
            href="http://dmoz.org/cgi-bin/add.cgi?where={$eid}">Submit
            a Site</a> - <a href="http://dmoz.org/about.html"><b>Open
            Directory Project</b></a> -
            <a href="http://dmoz.org/cgi-bin/apply.cgi?where={$eid}">Become
            an Editor</a> </font>
        </td></tr>
    </table>
</td>
</tr>
</table></p>
          </div>
          <p class="foot">Powered by STX</p>
        </body>
      </html>
    </stx:result-document>
    <stx:assign name="subtopics" select="true()" />
  </stx:template>

  <stx:group name="t">
    <stx:template match="od:catid">
      <stx:assign name="catid" select="." />
    </stx:template>

    <stx:template match="od:link">
      <stx:assign name="has-pages" select="true()" />
    </stx:template>
  </stx:group>

  <stx:template match="d:Title">
    <dt>
      <strong>
        <a href="{escape-uri(../@about, false())}" target="_blank">
          <stx:value-of select="." />
        </a>
      </strong>
    </dt>
  </stx:template>

  <stx:template match="d:Description">
    <dd>
      <stx:value-of select="." />
    </dd>
  </stx:template>

</stx:transform>
```

C.3 Web Services am Beispiel Google

Datei google-request.stx

```
<?xml version="1.0"?>
<!--
 | This STX transformation sheet sends a SOAP request via the
 | HTTP POST filter method of Joost to the Google Web Service.
 | The actual query must be passed via the global parameter 'search'.
 | The result of this transformation is the corresponding SOAP response.
 +-->
<stx:transform xmlns:stx="http://stx.sourceforge.net/2002/ns" version="1.0">

  <stx:param name="search" />

  <stx:buffer name="request">
    <SOAP-ENV:Envelope
              xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
              xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
              xmlns:xsd="http://www.w3.org/1999/XMLSchema">
      <SOAP-ENV:Body>
        <ns1:doGoogleSearch xmlns:ns1="urn:GoogleSearch"
           SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
          <key xsi:type="xsd:string">UKbgLPVQFHLleftuVSIVLvqjXXXXXXXX</key>
          <q xsi:type="xsd:string">
            <stx:value-of select="$search" />
          </q>
          <start xsi:type="xsd:int">0</start>
          <maxResults xsi:type="xsd:int">10</maxResults>
          <filter xsi:type="xsd:boolean">true</filter>
          <restrict xsi:type="xsd:string"></restrict>
          <safeSearch xsi:type="xsd:boolean">false</safeSearch>
          <lr xsi:type="xsd:string"></lr>
          <ie xsi:type="xsd:string">latin1</ie>
          <oe xsi:type="xsd:string">latin1</oe>
        </ns1:doGoogleSearch>
      </SOAP-ENV:Body>
    </SOAP-ENV:Envelope>
  </stx:buffer>

  <stx:template match="/">
    <stx:process-buffer name="request"
                        filter-method="http://www.ietf.org/rfc/rfc2616.txt#POST">
      <stx:with-param name="target"
                      select="'http://api.google.com/search/beta2'" />
    </stx:process-buffer>
  </stx:template>

</stx:transform>
```

Datei google2html.stx

```
<?xml version="1.0"?>
<!--
 | This STX transformation sheet creates an HTML page from Google's
 | SOAP response by displaying only title and URL of each result.
 +-->
<stx:transform xmlns:stx="http://stx.sourceforge.net/2002/ns" version="1.0">

  <stx:template match="/">
```

```
    <html>
      <body>
        <div style="border-width: 1pt; border-style: solid;
                    width: 40%; background-color: #eed;">
          <stx:process-children />
        </div>
      </body>
    </html>
  </stx:template>

  <stx:template match="resultElements">
    <ul>
      <stx:process-children />
    </ul>
  </stx:template>

  <stx:variable name="url" />
  <stx:variable name="title" />

  <stx:template match="resultElements/item">
    <stx:process-children />
    <stx:buffer name="text">
      &lt;span&gt;<stx:value-of select="$title" />&lt;/span&gt;
    </stx:buffer>
    <li>
      <a href="{$url}">
        <stx:process-buffer name="text"
                            filter-method="http://xml.org/sax" />
      </a>
    </li>
  </stx:template>

  <stx:template match="URL">
    <stx:assign name="url" select="." />
  </stx:template>

  <stx:template match="title">
    <stx:assign name="title" select="." />
  </stx:template>

</stx:transform>
```

Anhang D
Abkürzungsverzeichnis

API	Application Programming Interface
BNF	Backus-Naur Form
CML	Chemical Markup Language
CORBA	Common Object Request Broker Architecture
CSS	Cascading Style Sheets
DCOM	Distributed Component Object Model
DDD	Deutsch Diachron Digital
DiML	Dissertation Markup Language
DOM	Document Object Model
DSSSL	Document Style Semantics and Specification Language
DTD	Document Type Definition
EBNF	Erweiterte Backus-Naur Form
fxt	Functional XML Transformation Tool
GAME	Genome Annotation Markup Elements
HTML	Hypertext Markup Language
IDL	Interface Definition Language
IRI	Internationalized Resource Identifier
ISO	International Organization for Standardization
JAXP	Java API for XML Processing
MathML	Mathematical Markup Language
OASIS	Organization for the Advancement of Structured Information Standards
ODP	Open Directory Project
OMG	Object Management Group
OQL	Object Query Language
RDF	Resource Description Framework
SAX	Simple API for XML
SGML	Standardized General Markup Language
SML	Standard Meta Language
SQL	Structured Query Language
STX	Streaming Transformations for XML
SVG	Scalable Vector Graphics

TMML	Turing Machine Markup Language
TrAX	Transformation API for XML
UML	Unified Modeling Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WAP	Wireless Application Protocol
WSDL	Web Service Definition Language
XHTML	Extensible Hypertext Markup Language
XLink	XML Linking Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language
XMLTK	XML Toolkit
XPath	XML Path Language
XSL	Extensible Stylesheet Language
XSLFO	XSL Formatting Objects
XSLT	XSL Transformations

Anhang E
Literaturverzeichnis

AGG+02 Iliana Avila-Campillo, Todd J. Green, Ashish Gupta, Makoto Onizuka, Demian Raven, Dan Suciu, *XMLTK: An XML Toolkit for Scalable XML Stream Processing* in: *Proceedings of the Workshop Programming Languages Technologies for XML (PLANX)*, Pittsburgh, Oktober 2002, http://www.cs.washington.edu/homes/suciu/XMLTK/planx.pdf.

ASFa The Apache Software Foundation, *Apache XML Project, Xerces2 Java Parser*, http://xml.apache.org/xerces2-j/.

ASFb The Apache Software Foundation, *Apache XML Project, Xalan-Java*, http://xml.apache.org/xalan-j/.

ASFc The Apache Software Foundation, *Apache Jakarta Project*, http://jakarta.apache.org/.

ASFd The Apache Software Foundation, *Apache Cocoon Project*, http://cocoon.apache.org/.

BBC03 Oliver Becker, Paul Brown, Petr Cimprich, *An Introduction to Streaming Transformations for XML*, XML.com, Februar 2003, http://www.xml.com/pub/a/2003/02/26/stx.html.

BCF03 Véronique Benzaken, Guiseppe Castagna, Alain Frisch, *CDuce: An XML-Centric General-Purpose Language* in: *Proceedings of the ACM International Conference on Functional Programming*, Uppsala, August 2003, http://www.cduce.org/papers/cduce-design.ps.gz.

Bec03a Oliver Becker, *Transforming XML on the Fly* in: *XML Europe 2003 - Conference Proceedings*, London, Mai 2003, http://www.idealliance.org/papers/dx_xmle03/papers/04-02-02/04-02-02.html.

Bec03b Oliver Becker, *Extended SAX Filter Processing with STX* in: *Extreme Markup Languages 2003 Proceedings*, Montréal, August 2003. Als erweiterte Fassung erschienen in: *interChange, Newsletter of the International SGML/XML Users' Group (ISUG)*, Dezember 2003, Vol. 9, No. 4, S. 8-13.

Bro02 David Brownell, *SAX2*, O'Reilly, 2002.

Bro03 Martina Brose, *Interoperabilität von SOAP-Plattformen und Portabilität von SOAP-Anwendungen*, Studienarbeit, Institut für Informatik, Humboldt-Universität zu Berlin, Juli 2003.

BS01 Alexandru Berlea, Helmut Seidl, *Fxt – A Transformation Tool for XML Documents* in: *XML 2001 - Conference Proceedings*, Orlando, Dezember 2001, http://www.idealliance.org/papers/xml2001/papers/html/05-03-05.html.

Bur01 Eric M. Burke, *Java and XSLT*, O'Reilly, 2001.

Castor The ExoLab Group, *The Castor Project*, http://www.castor.org/.

DCMI03 Dublin Core Metadata Initiative, DCMI Empfehlung, *Dublin Core Metadata Element Set, Version 1.1: Reference Description*, 2. Juni 2003, http://dublincore.org/documents/2003/06/02/dces/.

Des01 Arpan Desai, *Introduction to Sequential XPath* in: *XML 2001 - Conference Proceedings*, Orlando, Dezember 2001, http://www.idealliance.org/papers/xml2001papers/tm/web/05-01-01/05-01-01.htm.

Dod01 Leigh Dodds, *Does XML Query Reinvent the Wheel?*, XML.com, Februar 2001, http://www.xml.com/pub/a/2001/02/28/deviant.html.

DOM4J James Strachan, *DOM4J*, http://www.dom4j.org/.

EXSLT *Extension Functions for XSLT (EXSLT)*, http://www.exslt.org/.

FHK+03 Daniela Florescu, Chris Hillery, Donald Kossmann, Paul Lucas, Fabio Riccardi, Till Westmann, Michael J. Carey, Arvind Sundararajan, Geetika Agrawal, *The BEA/XQRL Streaming XQuery Processor* in: *Proceedings of the 29th VLDB Conference*, Berlin, September 2003, http://wwwdb.informatik.uni-rostock.de/vldb2003/papers/S30P01.pdf.

GHJ+95 Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns*, Addison-Wesley, 1995.

Gib04 Jeremy Gibbons, *Metamorphisms and Streaming Algorithms*, Workshop in Refactoring Functional Programs, University of Kent, Februar 2004, http://web.comlab.ox.ac.uk/oucl/work/jeremy.gibbons/publications/metamorphisms.pdf.

Google *Google Web APIs*, http://www.google.com/apis/.

GP03 Vladimir Gapeyev, Benjamin C. Pierce, *Regular Object Types* in: *Proceedings of the 10th workshop FOOL*, New Orleans, Januar 2003, http://www.cis.upenn.edu/~bcpierce/papers/regobj.pdf.

HaXml Malcolm Wallace, Colin Runciman, *HaXml*, http://www.cs.york.ac.uk/fp/HaXml/.

HP03 Haruo Hosoya, Benjamin C. Pierce, *XDuce: A Statically Typed XML Processing Language* in: *ACM Transactions on Internet Technology*, Volume 3, Number 2, Mai 2003, http://www.kurims.kyoto-u.ac.jp/~hahosoya/papers/xduce-toit.ps.

IEEE754 Institute of Eletrical and Electronics Engineers, *IEEE Standard for Binary Floating-Point Arithmetics*, ANSI/IEEE Std 754-1985, http://standards.ieee.org/reading/ieee/std_public/description/busarch/754-1985_desc.html.

IETF99 IETF (Internet Engineering Task Force), *RFC 2616: Hypertext Transfer Protocol – HTTP/1.1*, R. Fielding et al., Juni 1999, http://www.ietf.org/rfc/rfc2616.txt.

ISO96 ISO (International Organization for Standardization), *ISO/IEC 10179:1996, Document Style Semantics and Specification Language (DSSSL)*, Internationaler Standard, ftp://ftp.ornl.gov/pub/sgml/WG8/DSSSL/.

JDOM Jason Hunter, Brett McLaughlin, *JDOM*, http://jdom.org/.

Joost Oliver Becker, *Joost STX Processor*, http://joost.sourceforge.net/.

JSR173 Christopher Fry et al., *Java Specification Request 173: Streaming API for XML*, Sun Microsystems, http://www.jcp.org/en/jsr/detail?id=173.

Kas90 Uwe Kastens, *Übersetzerbau*, Handbuch der Informatik, Band 3.3, Oldenbourg, 1990.

Kay00 Michael H. Kay, *XSLT Programmer's Reference*, Wrox Press, 2000.

Kay01 Michael H. Kay, *XSLT Performance* in: *XSLTUK 01, Preprints*, Oxford, April 2001.

Kru03 K. Ari Krupnikov, *STnG – a Streaming Transformations and Glue framework* in: *Extreme Markup Languages 2003 Proceedings*, Montréal, August 2003, http://www.idealliance.org/papers/extreme03/html/2003/Krupnikov01/-EML2003Krupnikov01.html.

Len01 Evan Lenz, *XSLT as a query language* in: *XSLTUK 01 Preprints*, Oxford, April 2001; online verfügbar als *XQuery: Reinventing the Wheel?*, http://www.xmlportfolio.com/xquery.html.

LS75 M. E. Lesk, E. Schmidt, *Lex – A lexical analyzer generator*, *Computer Science Technical Report #39*, 1975, Bell Laboratories, Murray Hill, NJ.

MS99 Erik Meijer, Mark Shields, *XMλ – A Functional Language for Constructing and Manipulating XML Documents*, November 1999, http://www.cse.ogi.edu/~mbs/pub/xmlambda/.

Nov03 Dimitre Novatchev, *Functional programming in XSLT using the FXSL library* in: *Extreme Markup Languages 2003 Proceedings*, Montréal, August 2003, http://www.idealliance.org/papers/extreme03/html/2003/Novatchev01/-EML2003Novatchev01.html.

OASIS01 Organization for the Advancement of Structured Information Standards (OASIS), *Relax NG Specification*, James Clark, Murata Makoto, 3. Dezember 2001, http://www.oasis-open.org/committees/relax-ng/spec-20011203.html.

Par01 Bijan Parsia, *Functional Programming and XML*, XML.com, Februar 2001, http://www.xml.com/pub/a/2001/02/14/functional.html.

RDF World Wide Web Consortium, *Resource Description Framework (RDF)*, http://www.w3.org/RDF/.

SAX David Megginson, David Brownell, et al., *Simple API for XML*, http://www.saxproject.org/.

Saxon Michael H. Kay, *Saxon XSLT Processor*, http://saxon.sourceforge.net/.

Schtrn Rick Jelliffe et al., *The Schematron Assertion Language*, http://www.ascc.net/xml/schematron/.

Sei03 Uwe Seimet, *Java-Logging-APIs: Mithören erwünscht* in: *iX*, 12/2003, S. 82-85, Verlag Heinz Heise GmbH & Co KG.

SMQ03 C. M. Sperberg-McQueen, *Playing by the rules* in: *Extreme Markup Languages 2003 Proceedings*, Montréal, August 2003, http://www.idealliance.org/papers/extreme03/html/2003/Sperberg-McQueen02/-EML2003Sperberg-McQueen02.html.

StDB Anatolji Zubow, *STX Debugger*, http://joost.sourceforge.net/StDB/.

StL03 Simon St. Laurent, *What can you do with half a parser?* in: *XML Europe 2003 - Conference Proceedings*, London, Mai 2003, http://www.idealliance.org/papers/dx_xmle03/papers/03-02-02/03-02-02.html.

STX Petr Cimprich, Oliver Becker et al. *Streaming Transformations for XML (STX) Version 1.0*, Working Draft, http://stx.sourceforge.net/documents/.

Ten01 Jeni Tennison, *XSLT and XPath On The Edge*, M & T Books, 2001.

Ten03 Jeni Tennison, *Typing in transformations* in: *Extreme Markup Languages 2003 Proceedings*, Montréal, August 2003, http://www.idealliance.org/papers/extreme03/html/2003/Tennison01/-EML2003Tennison01.html.

TMML Bob Lyons, *The Turing Machine Markup Language*, http://www.unidex.com/turing/tmml.htm.

Unicode The Unicode Consortium, *The Unicode Standard*, http://www.unicode.org/unicode/standard/standard.html.

W3C00 World Wide Web Consortium, *Document Object Model (DOM) Level 2 Core Specification*, W3C Empfehlung, 13. November 2000, http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/.

W3C01a World Wide Web Consortium, *Canonical XML Version 1.0*, W3C Empfehlung, 15. März 2001, http://www.w3.org/TR/2001/REC-xml-c14n-20010315.

W3C01b World Wide Web Consortium, *XML Schema*, W3C Empfehlung, 2. Mai 2001, http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/, http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/, http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/.

W3C01c World Wide Web Consortium, *XML Base*, W3C Empfehlung, 27. Juni 2001, http://www.w3.org/TR/2001/REC-xmlbase-20010627/.

W3C01d World Wide Web Consortium, *XML Linking Language (XLink) Version 1.0*, W3C Empfehlung, 27. Juni 2001, http://www.w3.org/TR/2000/REC-xlink-20010627/.

W3C01e World Wide Web Consortium, *Extensible Stylesheet Language (XSL) Version 1.0*, W3C Empfehlung, 15. Oktober 2001, http://www.w3.org/TR/2001/REC-xsl-20011015/.

W3C02 World Wide Web Consortium, *XPointer xpointer() Scheme*, W3C Arbeitsentwurf, 19. Dezember 2002, http://www.w3.org/TR/2002/WD-xptr-xpointer-20021219/.

W3C03a World Wide Web Consortium, *XML Path Language (XPath) 2.0*, W3C Arbeitsentwurf, 12. November 2003, http://www.w3.org/TR/2003/WD-xpath20-20031112/.

W3C03b World Wide Web Consortium, *XSL Transformations (XSLT) Version 2.0*, W3C Arbeitsentwurf, 12. November 2003, http://www.w3.org/TR/2003/WD-xslt20-20031112/.

W3C03c World Wide Web Consortium, *XQuery 1.0: An XML Query Language*, W3C Arbeitsentwurf, 12. November 2003, http://www.w3.org/TR/2003/WD-xquery-20031112/.

W3C03d World Wide Web Consortium, *XQuery 1.0 and XPath 2.0 Data Model*, W3C Arbeitsentwurf, 12. November 2003, http://www.w3.org/TR/2003/WD-xpath-datamodel-20031112/.

W3C03e World Wide Web Consortium, *XQuery 1.0 and XPath 2.0 Functions and Operators*, W3C Arbeitsentwurf, 12. November 2003, http://www.w3.org/TR/2003/WD-xpath-functions-20031112/.

W3C03f World Wide Web Consortium, *XSLT 2.0 and XQuery 1.0 Serialization*, W3C Arbeitsentwurf, 12. November 2003, http://www.w3.org/TR/2003/WD-xslt-xquery-serialization-20031112/.

W3C03g World Wide Web Consortium, *XML Syntax for XQuery 1.0 (XQueryX)*, W3C Arbeitsentwurf, 19. Dezember 2003, http://www.w3.org/TR/2003/WD-xqueryx-20031219/.

W3C04a World Wide Web Consortium, *Extensible Markup Language (XML) 1.0*, (Third Edition), W3C Empfehlung, 4. Februar 2004, http://www.w3.org/TR/2004/REC-xml-20040204/.

W3C04b World Wide Web Consortium, *Extensible Markup Language (XML) 1.1*, W3C Empfehlung, 4. Februar 2004, http://www.w3.org/TR/2004/REC-xml11-20040204/.

W3C04c World Wide Web Consortium, *XML Information Set*, (Second Edition), W3C Empfehlung, 4. Februar 2004, http://www.w3.org/TR/2004/REC-xml-infoset-20040204/.

W3C04d World Wide Web Consortium, *Namespaces in XML 1.1*, W3C Empfehlung, 4. Februar 2004, http://www.w3.org/TR/2004/REC-xml-names11-20040204/.

W3C99a World Wide Web Consortium, *Namespaces in XML*, W3C Empfehlung, 14. Januar 1999, http://www.w3.org/TR/1999/REC-xml-names-19990114/.

W3C99b World Wide Web Consortium, *XML Path Language (XPath) Version 1.0*, W3C Empfehlung, 16. November 1999, http://www.w3.org/TR/1999/REC-xpath-19991116.

W3C99c World Wide Web Consortium, *XSL Transformations (XSLT) Version 1.0*, W3C Empfehlung, 16. November 1999, http://www.w3.org/TR/1999/REC-xslt-19991116.

Weg99 Ingo Wegener, *Theoretische Informatik, eine algorithmenorientierte Einführung*, Teubner, 1999.

Wil03 Stephen D. Williams, *Efficiency structured XML (exXML)* in: *Extreme Markup Languages 2003 Proceedings*, Montréal, August 2003, http://www.esxml.org/.

XmlPull Stefan Haustein, Aleksander Slominski et al., *Common API for XML Pull Parsing*, http://www.xmlpull.org/.

XSLS Paul Tchistopolskii, *XSLScript*, http://www.pault.com/XSLScript/.

XST DecisionSoft Limited, *XML Script*, http://www.xmlscript.org/.

Zub02 Anatolij Zubow, *Transformation API for XML*, Studienarbeit, Institut für Informatik, Humboldt-Universität zu Berlin, November 2002.

Index

T

U

V

W

X

Z

Erklärung

Ich erkläre hiermit, dass

- ich die vorliegende Dissertationsschrift »Serielle Transformationen von XML. Probleme, Methoden, Lösungen« selbstständig und ohne unerlaubte Hilfe angefertigt habe,
- ich mich nicht bereits anderwärts um einen Doktorgrad beworben habe oder einen solchen besitze,
- mir die Promotionsordnung der Mathematisch-Naturwissenschaftlichen Fakultät II der Humboldt-Universität zu Berlin bekannt ist.

1. Juli 2004

Oliver Becker