Adaptive anwendungsspezifische Verarbeitung von XML-Dokumenten

Dissertation

zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
(Dr.-Ing.)

der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel

Daniel Fötsch

Kiel 2008

1. Gutachter	Prof. Dr. Andreas Speck
2. Gutachter	Prof. Dr. Wilhelm R. Rossak
Datum der mündlichen Prüfung	16.02.2010

Kurzfassung

Tranformationskomponenten sind ein wichtiger Bestandteil innerhalb der XML-Infrastruktur, um XML-Daten in das gewünschte Format zu überführen und dadurch bspw. den Austausch zwischen Systemen zu erlauben. Eine Möglichkeit der Umsetzung ist die Verwendung von spezifischen XML-Transformationssprachen. Spezifische XML-Transformationssprachen, wie z. B. XSLT, stellen spezielle Operatoren bereit, mit denen XML-Dokumente verarbeitet werden können. Die bereitgestellten Operatoren sind bzgl. dieser Domäne elementar und universell einsetzbar, um ein möglichst großes Spektrum der XML-Transformationsdomäne abzudecken. Aufgrund der feingranularen Operatoren kann die Beschreibung von XML-Transformationen umfangreich werden. Gegenwärtige XML-Transformationssprachen sind monolithische Sprachen. Sie stellen keine Mittel zur Verfügung, um neue Sprachkonstrukte hinzuzufügen, die bspw. problemspezifisch für eine bestimmte Zielsprache sind. Dies hat zu proprietären Spracherweiterungen geführt, die jedoch aufwendig mit einem Sprachprozessor implementiert werden müssen. Diese Ergänzungen werden deshalb nicht oder nur teilweise unterstützt.

In dieser Arbeit wird ein Konzept vorgeschlagen, mit dem neue höhere Operatoren auf der Grundlage existierender Operatoren aufgebaut werden können, ohne jedoch in die Sprachdefinition der Basistransformationssprache einzugreifen. Hierdurch kann die monolithische Struktur von XML-Transformationssprachen aufgebrochen und deren Nachteile überwunden werden. Durch das Zusammenfassen von immer wieder auftretenden Operatorkombinationen durch höhere Operatoren können bspw. Transformationsbeschreibungen kürzer und verständlicher werden. Höhere Operatoren können aber auch für eine ganz bestimmte Zielsprache entwickelt werden. Aufgrund der Spezifität können bspw. angemessene Prüf- und Fehlermechanismen realisiert werden.

Zur Umsetzung des Konzeptes ist die Ausführungsumgebung XTC entstanden. XTC koordiniert den Ablauf, um höhere Operatoren in niedrigere, letztendlich elementare Operatoren einer Basistransformationssprache zu überführen. XTC wurde so realisiert, dass über standardisierte Schnittstellen verschiedene spezifische XML-Transformationssprachen integriert werden können. Somit können die integrierten XML-Transformationssprachen einerseits erweitert werden und andererseits selbst für die Erweiterung genutzt werden. Neben XTC wird das Generatorsystem XOpGen entwickelt, welches den Implementierungsaufwand für die neuen höheren Operatoren weiter verringert. Das Potential von höheren Operatoren wird an der vom W3C standardisierten XML-Transformationssprache XSLT demonstriert. XSLT wird mit verschiedenen, sowohl universellen als auch problemspezifischen, Operatoren erweitert. Bei der Anwendung des Konzeptes in unterschiedlichen Forschungsprojekten ist u. a. die zielspezifische Transformationssprache XML2DSV entstanden.

Vorwort

Meinen besonderen Dank für die Betreuung und Unterstützung meiner Arbeit durch kritisch-konstruktive Diskussionen und hilfreiche Anregungen möchte ich Prof. Dr. A. Speck und Prof. Dr. W. Rossak aussprechen. Insbesondere für die Möglichkeit zur Forschung in den BMBF-Projekten Integration Engineering und OrViA sowie dem Thüringer Projekt VbSI danke ich Prof. Dr. A. Speck. Ebenso möchte ich mich bei Prof. Dr. B. Thalheim und Prof. Dr. R. von Hanxleden für die Unterstützung bedanken.

Das Gelingen dieser Arbeit war nur durch die beständige und tatkräftige Unterstützung meiner Betreuer, meiner KollegInnen in Kiel, Leipzig und Jena, meiner ProjektpartnerInnen, meiner Freunde, meiner Familie sowie durch den intensiven Austausch mit zahlreichen Kolleginnen und Kollegen im In- und Ausland möglich. Ihnen allen gehört mein herzlichster Dank!

Kiel, 1. August 2008

Daniel Fötsch

Inhaltsverzeichnis

Kurzfassung iii

Vorwort v

1 Einführung 1
- 1.1 Problem- und Zielstellung . . . 2
- 1.2 Beitrag . . . 3
- 1.3 Aufbau . . . 5

I Grundlagen 9

2 XML 11
- 2.1 Ursprung und Entwicklung . . . 12
- 2.2 Strukturelle Grundkonzepte . . . 14
  - 2.2.1 Aufbau . . . 15
  - 2.2.2 Strukturierung . . . 17
  - 2.2.3 Korrektheit . . . 17
  - 2.2.4 Namensraum . . . 17
- 2.3 Schemasprachen . . . 18
  - 2.3.1 Dokumenttyp-Definition . . . 19
  - 2.3.2 XML Schema . . . 20
  - 2.3.3 Schematron . . . 21
- 2.4 Abstrakte Datenmodelle . . . 23
- 2.5 Alternativer, sprachbasierter Zugang . . . 25
  - 2.5.1 Sprachen und Metasprachen . . . 25
  - 2.5.2 Modelle und Metamodelle . . . 27
- 2.6 Einordnung der Arbeit in den XML-Technologieraum . . . 31

3 Verarbeitung von XML-Dokumenten 33
- 3.1 Repräsentationsformen . . . 34
  - 3.1.1 Text . . . 35
  - 3.1.2 Datenstrom . . . 37
  - 3.1.3 Baum . . . 39
- 3.2 Phasen . . . 41

3.2.1 Analyse . . . 41
3.2.1.1 Lexikalische Analyse . . . 41
3.2.1.2 Syntaktische Analyse . . . 42
3.2.2 Transformation . . . 43
3.2.2.1 Transformationsregeln . . . 45
3.2.2.2 Regelausführungskontrolle . . . 49
3.2.2.3 Organisation von Regeln . . . 51
3.2.2.4 Input und Output . . . 52
3.2.2.5 Transformationsrichtung . . . 53
3.2.2.6 Mechanismus zur Transformationsverfolgung . . . 54
3.2.2.7 Programmierparadigma . . . 54
3.2.2.8 Syntax . . . 55
3.2.2.9 Kategorien von Transformationsmethoden . . . 55
3.2.3 Serialisierung . . . 59
3.3 Implementierungsmethoden . . . 59
3.3.1 Interne DSL . . . 59
3.3.2 Externe DSL . . . 63
3.3.3 Bewertung . . . 63
3.4 Einordnung der Arbeit in die Verarbeitung von XML-Dokumenten . . . 65

II Framework 67

4 Konzept 69
4.1 Anwendungsszenarien . . . 69
4.1.1 Umstrukturierung . . . 70
4.1.2 Überführung in ein Darstellungsformat . . . 73
4.2 Vision . . . 76
4.3 Anforderungen und Systemeigenschaften . . . 77
4.4 Systemstruktur . . . 80

5 Operatorhierarchie 83
5.1 Einführung . . . 84
5.2 Detaillierte Beschreibung . . . 88
5.3 Realisierung . . . 103
5.3.1 Core System . . . 103
5.3.2 Level Library . . . 104
5.3.3 Integration der Sprachprozessoren . . . 105
5.3.4 Funktionsweise . . . 109
5.3.5 Experimentelle Untersuchungen . . . 111

6 Generative Techniken 121
6.1 Einführung . . . 122
6.2 Detaillierte Beschreibung . . . 125
6.3 Realisierung . . . 130
6.3.1 Generatoren für die Schemaüberführungen . . . 131
6.3.2 Generatoren zur Erzeugung der Validierungskomponenten . . . 134
6.3.3 Infrastruktur, Ebenentransformator und Integration . . . 136

7 Entwicklungsvorgehen 139
7.1 Top-down-Vorgehensmodell . . . 139
7.2 Bottom-up-Vorgehensmodell . . . 151
7.3 Rollenverteilung . . . 156

III Evaluation 161

8 Anwendung 163
8.1 Erweiterung von XSLT . . . 163
8.1.1 Universelle Operatoren . . . 164
8.1.2 Domänenspezifische Operatoren . . . 176
8.2 Fallbeispiele . . . 177
8.2.1 Datenaustausch am Beispiel von Truitions XML-Export – Die Transformationssprache XML2DSV . . . 177
8.2.2 Weitere Fallbeispiele . . . 184
8.3 Schlussbemerkungen . . . 186
8.3.1 Vergleich mit XSLT 1.0 . . . 187
8.3.2 Vergleich mit XSLT 2.0 . . . 188

9 Diskussion 191
9.1 Nutzen . . . 191
9.1.1 Verständlichkeit und Kompaktheit . . . 191
9.1.2 Prüfbarkeit . . . 194
9.1.3 Wiederverwendbarkeit . . . 197
9.1.4 Adaptivität . . . 198
9.1.5 Agilität . . . 198
9.2 Kosten . . . 200
9.2.1 Auswirkungen auf die Codemenge . . . 200
9.2.2 Auswirkungen auf den Gesamtaufwand . . . 202
9.2.2.1 Entwicklungskosten der Sprachkomponente . . . 202
9.2.2.2 Startup- und Overhead-Kosten . . . 203
9.2.2.3 Sonstige Kosten . . . 204
9.2.3 Schlussbemerkungen . . . 204

10 Zusammenfassung und Ausblick **207**

A Quellcode der Anwendungsbeispiele **211**

A.1 Definition des `function`- und `call-function`-Operators 211

A.2 Definition der Transformationssprache XML2DSV 214

B Sprachmerkmale von XSLT **219**

Literaturverzeichnis **225**

Abkürzungsverzeichnis **245**

Index **249**

Abbildungsverzeichnis

1.1 Die Top 25 der meistgenannten IT-Begriffe im Jahr 2006 in den deutschsprachigen Printmedien und Agenturmeldungen 2
1.2 Überblick über Aufbau und Kapitelabhängigkeiten 5

2.1 Vereinfachte schematische Darstellung des Infosets 24
2.2 Sprachbestandteile von XML 26
2.3 Hierarchie von Sprachebenen im Kontext von XML 27
2.4 Elemente der Modellbildung 28
2.5 Linguistische Modellhierarchie im Kontext von XML basierend auf der XML-Spezifikation 30
2.6 Modellhierarchie im Kontext von XML basierend auf XML Schema 30
2.7 Modellhierarchie im Kontext von XML basierend auf MOF 31

3.1 Repräsentationsformen von XML 35
3.2 Merkmaldiagramm auf der obersten Ebene 45
3.3 Merkmale der Transformationsregeln 46
3.4 Merkmale der Typisierung 48
3.5 Merkmale der Regelausführungskontrolle 50
3.6 Merkmale der Regelorganisation 51
3.7 Merkmale der Inputs und Outputs 53
3.8 Merkmale der Transformationsrichtung 54
3.9 Merkmale der Transformationsverfolgung 54

4.1 Erzeugte XHTML-Datei im Browser 75
4.2 Struktur des Frameworks 80

5.1 Einordnung von XTC in die Gesamtstruktur 83
5.2 Grobeinteilung von Operatorhierarchien 87
5.3 Schematische Beispieloperatorhierarchie mit Sprachkomponenten 92
5.4 Mögliche Sprachen einer schematischen Beispieloperatorhierarchie 94
5.5 Beispielanwendung des `first`-Ebenentransformationsprozessalgorithmus 96
5.6 Untere Schranke des `first`-Algorithmus 97
5.7 Beispielanwendung des `maxrank`-Ebenentransformationsalgorithmus 99
5.8 Beispielanwendung des `sort`-Ebenentransformationsalgorithmus 101
5.9 Struktur von XTC 104
5.10 Integration von Sprachprozessoren 108

5.11 Zusammenspiel der Komponenten von XTC 109
5.12 Beispielablauf eines Transformationsprozesses in XTC 110
5.13 Laufzeiten der Ebenentransformationsprozessalgorithmen `first`, `maxrank` und `sort` im Vergleich (1. Experiment) 114
5.14 Laufzeiten der Ebenentransformationsprozessalgorithmen `first`, `maxrank` und `sort` im Vergleich (2. Experiment) 116
5.15 Laufzeiten der Ebenentransformationsprozessalgorithmen `first`, `maxrank` und `sort` im Vergleich (3. Experiment) 119

6.1 Einordnung des Generatorsystems XOpGen in die Gesamtstruktur 121
6.2 Konzept von XOpGen 124
6.3 Generationsprozess vom grammatikbasierten Schema zu den Validierern 125

7.1 Das Top-down-Vorgehensmodell 140
7.2 Das Bottom-up-Vorgehensmodell 152
7.3 Spezifische Rollen bei der (Weiter-)Entwicklung von Sprachen 157

8.1 Zusätzliche Sprachkomponenten für XSLT 163
8.2 Übersicht über Ergebnisse des Erstellungsprozesses 166
8.3 JavaDoc-ähnliche Dokumentation der `control`-Sprachkomponente 172
8.4 Ebenentransformation für `control`-Sprachkomponente 176
8.5 Truitions Commerce Management System (CMS) 178
8.6 Merkmale von XML2DSV 180
8.7 Angepasster Editor für XML2DSV 185

9.1 Verhältnis von XSLT-Code zu XML2DSV-Code 194
9.2 Verhältnis von generischen zu manuell erstellten und generierten Artefakten am Beispiel der universellen `control`-Sprachkomponente 199
9.3 Verhältnis von generischen zu manuell erstellten und generierten Artefakten am Beispiel der domänenspezifischen Transformationssprache XML2DSV 200

Tabellenverzeichnis

3.1 Gegenüberstellung von Analysekomponenten und unterstützte Analyseschritten . 43

4.1 Ziel-/Anforderungsmatrix 79

5.1 Strukturelle Unterscheidungsmerkmale von XML-basierten Operatoren 86
5.2 Beispiele von XML-basierten Operatoren 86
5.3 Unterscheidungsmerkmale einer Operatorebenentransformation 90
5.4 Uneingeschränkte Anwendbarkeit der Algorithmen `first`, `maxrank` und `sort` bzgl. der Unterscheidungsmerkmale von Operatorebenentransformationen 102
5.5 Eingaben für die Beispielmessungen (1. Experiment) 112
5.6 Übersicht der Einzelwertmessungen der Ebenentransformationen für n=3 der Algorithmen `first`, `maxrank` und `sort` (1. Experiment) 113
5.7 Eingaben für die Beispielmessungen (2. Experiment) 115
5.8 Übersicht der Einzelwertmessungen der Ebenentransformationen für n=5 der Algorithmen `first`, `maxrank` und `sort` (2. Experiment) 117
5.9 Eingaben für die Beispielmessungen (3. Experiment) 118
5.10 Übersicht der Einzelwertmessungen für n=5 der der Algorithmen `first`, `maxrank` und `sort` (2. Experiment) 118

6.1 Schematische Abbildung von Sprachkonstrukten des XML Schemas nach Schematron 132
6.2 Übersicht über die wichtigsten Hooks des Schema-Generators 133
6.3 Abbildung von Sprachkonstrukten des Schematrons nach XSLT 135
6.4 Übersicht über die wichtigsten Hooks des Code-Generators 136

7.1 Aktivität 1: Definieren der Anwenderzielgruppe 141
7.2 Aktivität 2: Erfassen von Terminologien und Konzepten 142
7.3 Aktivität 3: Eingrenzen der Quell- und Zielsprachen 142
7.4 Aktivität 4: Bündeln und Aufbereiten des gesammelten Wissens 143
7.5 Aktivität 5: Definieren der domänenspezifischen Sprachkomponente 144
7.6 Aktivität 6: Definieren des Referenztransformationsszenarios 145
7.7 Aktivität 7: Erstellen der Referenztransformationsdefinition 145
7.8 Aktivität 8: Erstellen der Referenzimplementierung auf der Plattform 146
7.9 Aktivität 9: Erweitern der Plattform 147
7.10 Aktivität 10: Ableiten des Ebenentransformators 148
7.11 Aktivität 11: Generieren des Ebenen- und Inputvalidierers 149

7.12 Aktivität 12: Generieren der Dokumentation 149
7.13 Aktivität 13: Erstellen des Editors 150
7.14 Aktivität 14: Integrieren in die Ebenentransformationsinfrastruktur 150
7.15 Aktivität 15: Test, Installation und Nutzung 151
7.16 Aktivität 1: Identifizieren potentieller Operatoren 152
7.17 Aktivität 2: Definieren neuer Operatoren 153
7.18 Aktivität 3: Anpassen vorhandener Operatoren 153
7.19 Aktivität 4: Ableiten des Ebenentransformators 154
7.20 Aktivität 5: Generieren des Ebenen- und Inputvalidierers 154
7.21 Aktivität 6: Generieren der Dokumentation 155
7.22 Aktivität 7: Erstellen des Editors 155
7.23 Aktivität 8: Integrieren in die Ebenentransformationsinfrastruktur 155
7.24 Aktivität 9: Test, Installation und Nutzung 156
7.25 Aktivitäten/Rollen-Matrix des Top-down-Vorgehensmodells und des Bottom-up-Vorgehensmodells 160

9.1 Analyse des `move-after`-Operators und des `swap`-Operators auf Codeebene . . . 202

B.1 Vergleich von XSLT 1.0, XSLT 1.0 mit Erweiterungen und XSLT 2.0 anhand der aufgestellten Sprachmerkmale 223

Code-Verzeichnis

2.1 Glossar mit XML-Markup. 14
2.2 Schema des Glossars mit DTD. 19
2.3 Schema des Glossars mit XML Schema. 21
2.4 Schema des Glossars mit Schematron. 22
4.1 Liste von Organisationen. 70
4.2 Verschieben von Teilbäumen mit XUpdate. 71
4.3 Austauschen von Teilbäumen mit XUpdate. 72
4.4 Erzeugen von XHTML mit XSLT. 74
4.5 CSS für XHTML. 75
5.1 Ebenentransformation für das Verschieben und Austauschen von Teilbäumen mit XUpdate. 89
5.2 Aufruf einer Transformation in TrAX. 106
5.3 Dynamische Auswahl der integrierten Sprachprozessoren. 108
6.1 Mögliche Validierungsregeln für den `move-after`-Operator. 123
6.2 Syntaxdefinition des `move-after`-Operators in XML Schema. 126
6.3 Zusicherungen für den `move-after`-Operator in Schematron. 128
6.4 Zusicherungen für den Input des `tr`-Operators in Schematron. 130
6.5 Generische Infrastrukturkomponente von XOpGen. 137
6.6 Integration der einzelnen Komponenten und der Infrastruktur. 137
6.7 Ebenentransformator bei analoger Syntax von höheren Transformationsoperatoren und Sprachkonstrukten der Zielsprache. 138
8.1 Ausschnitt aus dem Schema (`control.xsd`) – das `with-param`-Element des `call-function`-Operators. 168
8.2 Ebenentransformationsdefinition des `function`- und `call-function`-Operators. . 169
8.3 Ausschnitt aus dem Ebenenvalidierer (`level-validator.xslt`) – das `with-param`-Element des `call-function`-Operators. 171
8.4 Beispielregistrierung der `control`-Sprachkomponente in XTC. 173
8.5 Beispieltransformationsaufgabe. 174
8.6 Nutzung des `function`- und `call-function`-Operators. 175
8.7 Anwendung der XML2DSV-Transformationssprache. 181
8.8 Ausschnitt der Implementation der Referenztransformationsdefinition in XSLT unter der Verwendung der `control`-Sprachkomponente. 183
8.9 Registrierung des Schemas von XML2DSV beim Editor. 184
9.1 Vergleich des `for`-Operators der `control`-Sprachkomponente und dessen Implementierung in XSLT. 192

A.1 Ausschnitt aus dem Schema (`control.xsd`) – der `function`-Operators und der `call-function`-Operators. 211
A.2 Schema der Transformationssprache XML2DSV. 214

Kapitel 1

Einführung

Das Akronym XML ist zu seinem zehnjährigen Bestehen einer der meistgenannten IT-Begriffe innerhalb der Medien (vgl. letzte aktuelle Untersuchung in Abbildung 1.1). In nahezu jedem Anwendungsgebiet der Informationstechnik hält XML nicht nur als plakatives Schlagwort Einzug. Vielmehr existieren fast keine neuen IT-Entwicklungen oder Produkte, welche XML nicht in irgendeiner Form nutzen oder unterstützen. Aber was macht dieses einfache und zugleich sehr flexible Format zu dem, was Henry S. Thompson in [Tho00] als ASCII des 21. Jahrhunderts bezeichnet?[1] Zumal XML ursprünglich lediglich für das Publizieren von elektronischen Inhalten innerhalb des Webs entworfen wurde.

Die Stärke von XML besteht darin, dass XML ermöglicht, Inhalte durch zusätzliche Informationen anzureichern und zu strukturieren. Mit Hilfe der Strukturierung durch sogenannte Textauszeichnungen oder Markups können Inhalte einfacher gefunden, verändert oder anderweitig automatisiert weiterverarbeitet werden. XML legt dabei lediglich die erlaubten Typen von Textauszeichnungen und deren syntaktischen Aufbau fest. Basierend auf diesen Konventionen besteht die Möglichkeit, eigene, konkrete Markup-Sprachen für die individuellen Ansprüche und Anforderungen innerhalb der verschiedenen Anwendungsgebiete zu formulieren.

Aufgrund der Festlegung des syntaktischen Aufbaus von Textauszeichnungen sowie der möglichen Beziehungen zwischen den einzelnen Typen von Textauszeichnungen können, unabhängig von den konkreten Markup-Sprachen, verschiedenste Sprachen, Konzepte, Methoden und Werkzeuge (z. B. Parser, Transformatoren, Validierer, Serialisierer) modular entwickelt werden. Diese vereinfachen das Speichern, Exportieren, Verarbeiten oder Importieren von XML-Dokumenten der anwendungsbezogenen Markup-Sprachen und gestalten es in vielerlei Hinsicht effizienter. Die Plattformunabhängigkeit von XML bezüglich einer konkreten Hardware, eines Betriebssystems oder einer Programmiersprache zum einen und die Lizenzkostenfreiheit zum anderen bilden den Zugang zu der stetigen Weiterentwicklung dieser Infrastruktur.

Neben den Eigenschaften von XML macht diese Infrastruktur XML besonders als Austauschformat interessant. Ohne die Festlegung auf XML als Serialisierungsformat müssen zum Datenaustausch zwischen zwei Applikationen die jeweiligen proprietären Quellformate mit einem individuellen Parser eingelesen und analysiert werden, entsprechende Transformationen vorgenommen und anschließend wieder gemäß dem proprietären Zielformat serialisiert werden. Ohne

[1] Zur richtigen Einordnung dieser Aussage sei erwähnt, dass ASCII das fundamentale Austauschproblem für flache Textdokumente löste, indem es in [Int91] definierte, welche Zeichen durch welche Bits kodiert werden.

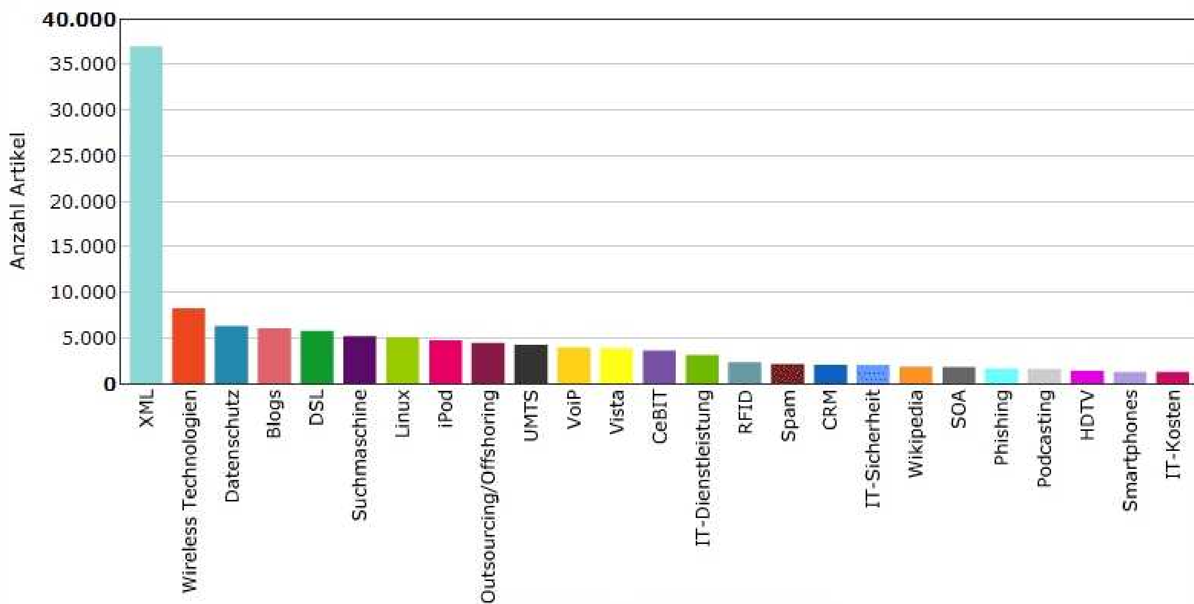


ABBILDUNG 1.1: Die Top 25 der meistgenannten IT-Begriffe im Jahr 2006 in den deutschsprachigen Printmedien und Agenturmeldungen [Com07].

XML kann i. d. R. nicht auf eine Vielzahl von Werkzeugen für die einzelnen Schritte zurückgegriffen werden. Gerade deswegen stellt XML aus heutiger Sicht die Standardtechnologie für die Datendarstellung und den Datenaustausch dar. Man geht davon aus, dass rund 40 Prozent der weltweit täglich erstellten Daten im offenen XML-Format abgespeichert werden [Com07].

1.1 Problem- und Zielstellung

Die alleinige Vereinbarung der Nutzung einer XML-Schnittstelle für den Datenaustausch genügt allerdings bei weitem nicht, um diesen sicherzustellen. Voraussetzung dafür ist vielmehr, dass zwei Applikationen zwischen denen der Datenaustausch stattfinden soll, sich auf eine gemeinsame Markup-Sprache mit gleichem Vokabular und gleicher Semantik einigen. Diese gemeinsame Kommunikationsbasis könnte eine Standardauszeichnungssprache innerhalb des Anwendungsgebietes sein. In vielen Anwendungsgebieten muss sich jedoch erst noch *die* Standardsprache etablieren und verbreiten. Einerseits existieren alternative Vorschläge, die nicht selten in Konkurrenz stehen und oftmals nur auf bestimmte Märkte, Länder oder Ländergruppen zugeschnitten sind. Andererseits kann eine Standardsprache in manchen Anwendungsgebieten die individuellen Anforderungen der Applikationen nur ungenügend befriedigen. In beiden Situationen wird deshalb entweder der Standard durch herstellerspezifische Anpassungen (z. B. für zusätzliche graphische Informationen für Modellelemente bei Editoren) verändert bzw. erweitert oder auf eine komplett individuelle, werkzeugspezifische Markup-Sprache gesetzt. Damit in einem solchen Fall die Applikationen nicht verschiedenste Vokabulare exportieren oder importieren müssen,

sind innerhalb einer XML-Infrastruktur Transformationssysteme ein wichtiger Bestandteil, die die Daten in das jeweils gewünschte XML-Vokabular übersetzen.

Existierende XML-Transformationssysteme bieten nur elementare Sprachkonstrukte zur Beschreibung der Transformationsdefinitionen an. Elementare Sprachkonstrukte sind Operatoren, die abhängig von der konkreten Transformationssprache Informationseinheiten von XML erzeugen, ändern oder löschen können. Solche Transformationssysteme sind mächtig, da sie jegliche auf XML-basierende Markup-Sprache erzeugen können und deswegen für die gesamte XML-Transformationsdomäne eingesetzt werden können. Andererseits führen die elementaren Sprachkonstrukte zu feingranularen und umfangreichen Transformationsbeschreibungen. Obwohl der Anwendungsbereich bzgl. der Quell- und/oder Zielsprache weiter eingeschränkt werden kann, wird die Transformationslogik in elementaren Sprachkonstrukten „versteckt“. Gegenwärtige XML-Transformationssprachen bieten keine Mittel, um bspw. Sprachkonstrukte für eine bestimmte Klasse von XML-Dokumenten einer Zielsprache hinzuzufügen. Infolgedessen ist der Code schwerer verständlich, prüfbar und dadurch fehleranfälliger. Dies wird besonders deutlich bei Transformationssprachen, die neben XML auch andere Formate (z. B. PDF, RTF, HTML) erzeugen können. Für die Beschreibung dieser Transformationen werden die Zielformate durch Aneinanderreihen von Zeichenketten gebildet. Jegliche Unterstützung für eine Validierung fehlt.

In dieser Arbeit sollen spezifischen XML-Transformationssprachen weiterentwickelt werden. Die zentrale Zielsetzung ist, ein Framework aufzubauen, mit dem die Nachteile der starren, monolithischen Struktur von gegenwärtigen XML-Transformationssprachen überwunden werden können. Das Framework soll die Möglichkeit schaffen, XML-Transformationssprachen adaptiv und modular erweitern zu können. Das Framework soll so organisiert sein, dass einzelne Sprachkonstrukte, umfassendere Sprachkomponenten oder gänzlich eigenständige Transformationssprachen basierend auf einer XML-Basistransformationssprache flexibel implementiert werden können. Entsprechende Granularitäten muss das zugrundeliegende Konzept unterstützen. Das Konzept soll außerdem so strukturiert sein, dass möglichst wenige Einschränkungen bzgl. der neuen Sprachkonstrukte erfolgen. Mit dem Framework soll es bspw. möglich sein, sowohl technische, allgemein wiederverwendbare Sprachkonstrukte als auch abstrakte, anwendungsspezifische Sprachkonstrukte zu entwickeln. Zudem soll das Framework für eine Vielzahl von spezifischen XML-Transformationssprachen anwendbar sein. Die Detaillierung der Zielstellung durch Ableitung von Anforderungen an das Framework wird in Abschnitt 4.3 vorgenommen.

1.2 Beitrag

Für die Anreicherung von XML-Transformationssprachen mit zusätzlichen Wissen existieren verschiedene Ansätze. Es werden bspw. wiederverwendbare Funktionen (u. a. Funktionen zur Berechnung von Datums- und Zeitangaben, Mathematik- und Mengenfunktionen, zusätzliche Zeichenkettenfunktionen) zur Verfügung gestellt, die mit entsprechenden Sprachkonstrukten der Transformationssprache importiert und anschließend entsprechend aufgerufen werden können. Diese Form der Wissenssammlung ist vergleichbar mit einer klassischen Programmbibliothek,

die selbstständige, in der selben Sprache formulierte Programmkomponenten für eine Wiederverwendung bereitstellt. Beispielsweise kommuniziert die EXSLT-Initiative [EXS08] verschiedene Funktionen für die vom W3C standardisierte Transformationssprache XSLT 1.0 [Cla99a]. FunctX [Fun08] bietet ebenso eine Sammlung von Funktionen für XSLT. Solche Funktionen können bei Bedarf mit dem in dieser Arbeit entwickelten Konzept durch eigene Sprachkonstrukten beschrieben werden. Es kann dementsprechend eine direkte Repräsentation in den Transformationsdefinitionen erfolgen, welche im Allgemeinen in einer kompakteren und verständlicheren Darstellung resultiert. Mit Hilfe von höheren Sprachkonstrukten können darüber hinaus komplexere Modularisierungen als mit dem Funktionsmechanismus realisiert werden.

Aus diesem Grund werden ebenso Spracherweiterungen zum Standard vorgeschlagen. Diese Erweiterungen können im Gegensatz zu den vorherigen Funktionssammlungen nicht mehr von standardkonformen Sprachprozessoren ausgeführt werden. Infolgedessen haben sich Inkompatibilitäten zwischen den Sprachprozessoren entwickelt: manche bleiben beim Standard, andere unterstützen nur einen Teil der vorgeschlagenen Erweiterungen und wiederum andere bieten zusätzlich eigene proprietäre Sprachkonstrukte an. Insgesamt ist somit ein Austausch der Sprachprozessoren an viele Voraussetzungen geknüpft. Für XSLT 1.0 bspw. hat die EXSLT-Initiative Erweiterungen angeregt. Letztlich haben einige dieser sowie proprietäre Erweiterungen (z. B. von Saxon [Kay08]) Einzug in die neue Version von XSLT 2.0 [Kay07] gefunden. Die Historie von XSLT zeigt, dass besonders vergleichsweise junge Sprachen ständigen Weiterentwicklungen unterworfen sind. Unser Konzept ermöglicht u. a. die Erweiterungen ohne einen Eingriff in die Sprachdefinition von XSLT oder anderen Transformationssprachen, so dass beliebige vorhandene und zukünftige standardkonforme Sprachprozessoren genutzt werden können.

Der in den Bibliotheken der XML-Transformationssprachen enthaltene wiederverwendbare Code ist i. d. R. unabhängig von einem konkreten Anwendungskontext. Für spezifische Anwendungsprobleme sind bisher eigenständige Transformationssprachen entwickelt worden. Sie werden üblicherweise mit universellen Programmiersprachen mit den entsprechenden APIs wie DOM [CBNW98] oder dessen Derivaten (JDOM [HM07], dom4j [Str05], etc.) direkt implementiert. Ein Beispiel dafür ist das Kompositionskonzept von XOpT [PS04]. XOpT ist ein Sprachvorschlag, der Java-Code aspektorientiert weiterverarbeiten kann. Der Java-Code muss zu diesem Zweck zuvor in die XML-basierte Markup-Sprache JavaML überführt und nach der Verarbeitung wieder serialisiert werden (siehe dazu auch [Bad00]). Wiederum einige andere Sprachvorschläge (z. B. [PZB00, Lei03, Pan04]) werden auf spezifische XML-Transformationssprachen (meistens XSLT) abgebildet. In den genannten Beispielen wird die dafür notwendige Überführung mit universellen Programmiersprachen vorgenommen.

Unser Konzept leistet ebenso für die Neuentwicklung einer XML-Transformationssprache einen Beitrag. Es erlaubt, für die Überführung zu einer XML-Basistransformationssprache selbst Transformationssprachen zu nutzen. Hierdurch kann die monolithische Sprachentwicklung aufgebrochen werden. Es können Sprachkomponenten konstruiert und anschließend kombiniert werden. So ist es neben einer kompletten Neuentwicklung auch möglich, eine Basistransformationssprache mit problemangepassten Operatoren zusammengefasst in einer Sprachkomponente modular und adaptiv zu erweitern. Auf der einen Seite kann durch die Modularisierung eine

höhere Wiederverwendbarkeit, bessere Anpassungsfähigkeit und kürzere Entwicklungszeit erzielt werden. Auf der anderen Seite können die auf die gesamte XML-Domäne spezialisierten Sprachkonstrukte einer Basistransformationssprache durch spezifische Sprachkonstrukte, die sich bspw. auf die Erzeugung eines bestimmten Zielformates beschränken und deshalb die Konzepte und Terminologien des Zielformates explizit verkörpern können, angereichert werden.

1.3 Aufbau

Abbildung 1.2 gibt einen Überblick über den Aufbau der Arbeit und die Abhängigkeiten der Kapitel. Die Arbeit ist in drei Teile gegliedert – Grundlagen, Framework und Evaluation.

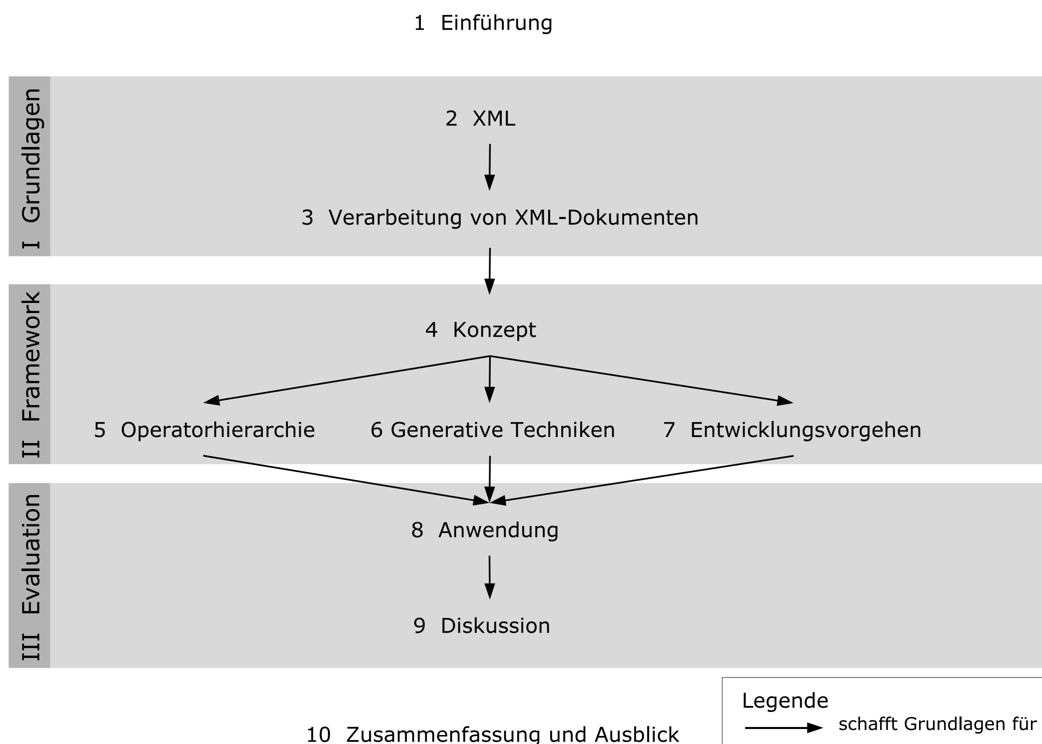


Abbildung 1.2: Überblick über Aufbau und Kapitelabhängigkeiten.

Im ersten Teil werden die Grundlagen vorgestellt, die für das Verständnis der vorliegenden Arbeit bzgl. XML im Allgemeinen und der Verarbeitung von XML-Dokumenten im Besonderen notwendig sind. Dementsprechend ist der erste Teil unterteilt in zwei Kapitel.

Kapitel 2 gibt eine Einführung in wesentliche Basiskonzepte von XML. Neben den grundlegenden syntaktischen Konstrukten von XML werden Schemasprachen zur Definition von individuellen XML-Vokabularen sowie ein abstraktes Datenmodell, bei dem gegenüber der Syntax die Informationen im Vordergrund stehen, beschrieben. Nach dieser eher intuitiven Einführung von

XML erfolgt eine systematische, sprachbasierte Definition der Begriffe sowie der Begriffsbeziehungen. Das Kapitel schließt mit einer Einordnung der Arbeit in den XML-Technologieraum.

Kapitel 3 führt in die Verarbeitung von XML-Dokumenten ein. Es werden zu Beginn verschiedene Verarbeitungsmodelle betrachtet, bevor auf die einzelnen Verarbeitungsphasen eingegangen wird. Die Phase der Transformation wird dabei im Detail aufgearbeitet. Es wird eine umfassende Domänenanalyse vorgenommen, in der die Merkmale von diversen Transformationsansätzen untersucht werden. Diese Merkmale werden anschließend aufgegriffen, um Kategorien von grundsätzlichen Transformationsmethoden, denen i. d. R. mehrere Transformationsansätze zugeordnet werden können, zu bilden. Nachfolgend werden unterschiedliche Implementierungsaspekte diskutiert. Abschließend wird die Arbeit bzgl. der XML-Verarbeitung eingeordnet. Es werden dabei erste Ansatzpunkte für Verbesserungspotentiale aufgezeigt.

Der zweite Teil bildet den Schwerpunkt dieser Arbeit und ist in vier weitere Kapitel untergliedert. Hier werden die Konzepte zur Erreichung der Zielsetzung entwickelt und realisiert.

Im **Kapitel 4** werden zwei typische Anwendungsszenarien für XML-Transformationen und deren Probleme gezeigt. Auf dieser Basis werden die Ziele der Arbeit konkretisiert und die Anforderungen herausgearbeitet. Die Struktur des gesamten Frameworks wird anhand der aufgestellten Anforderungen abgeleitet. Die einzelnen Module des Frameworks werden in den beiden folgenden Kapiteln erklärt.

Kapitel 5 geht auf das Fundament des Frameworks ein. In das Grundprinzip der dafür notwendigen Operatorhierarchie wird zunächst kurz eingeführt, bevor das Konzept umfassend beschrieben wird. Hiernach folgt die Erläuterung der technischen Realisierung.

Im **Kapitel 6** wird ein Generatorsystem vorgeschlagen, welches die Erstellung von neuen Sprachkonstrukten unterstützt. Erneut wird zu Beginn das grundlegende Prinzip kurz dargestellt und daraufhin ausführlich erläutert. Den Abschluss des Kapitels bildet die Schilderung der technischen Realisierung. Zum besseren Verständnis wird der Generationsprozess dabei beispielhaft mit Hilfe der beiden Anwendungsszenarien demonstriert.

Im **Kapitel 7** werden zwei Vorgehensmodelle vorgestellt, die beschreiben, welche Aktivitäten in welcher Reihenfolge und mit welchen Rollen durchgeführt werden. Die Vorgehensmodelle legen u. a. auch fest, wie und wann die entworfenen Module des Frameworks angewendet werden.

Der dritte Teil der Arbeit evaluiert die entwickelten Konzepte und deren Realisierung. Er teilt sich in zwei weitere Kapitel auf.

Im **Kapitel 8** wird die Anwendung des entwickelten Frameworks exemplarisch an XSLT gezeigt. Es wird sowohl die Erweiterung der Basistransformationssprache mit zusätzlichen Sprachkomponenten als auch der Aufbau einer neuen, eigenständigen Transformationssprache basierend auf XSLT demonstriert.

Kapitel 9 diskutiert den Nutzen und die Kosten des vorgeschlagenen Frameworks. In die Kosten/Nutzen-Analyse werden die zuvor betrachteten Anwendungsbeispiele einbezogen, um Argumente zu veranschaulichen sowie quantitative Messungen vornehmen zu können.

Die Arbeit schließt mit dem **Kapitel 10**. Dieses Kapitel fasst die wichtigsten Ergebnisse dieser Arbeit zusammen. Zum anderen wird ein Ausblick auf mögliche Anknüpfungspunkte und weiterführende Arbeiten gegeben.

Teil I

Grundlagen

Der erste Teil dieser Arbeit behandelt die wesentlichen Grundkonzepte von XML. Es werden u. a. die XML-Syntax, verschiedene Schemasprachen sowie abstrakte und konkrete Datenmodelle beschrieben. Anschließend wird gesondert auf die eigentliche Transformationsphase eingegangen. Es werden Merkmale unterschiedlicher Transformationssysteme gesammelt und kategorisiert. Ebenso werden Implementierungstechniken untersucht und diskutiert.

Kapitel 2
XML

Die im Februar 1998 verfasste XML-Spezifikation zur Erstellung strukturierter, maschinen- und menschenlesbarer Dokumente legte aus heutiger Sicht den Grundstein zur Entwicklung einer Standardtechnologie für den Datenaustausch. XML wird nicht direkt in den verschiedensten Applikationen verarbeitet, sondern die jeweiligen problemspezifischen, XML-basierten Dokumente und Sprachen. XML stellt jedoch die grundlegenden syntaktischen Konventionen zur Verfügung, die festlegen, wie Markups abgegrenzt werden und an welchen Stellen eines Textes diese hinzugefügt werden dürfen; mehr allerdings auch nicht. Schemasprachen ermöglichen darüber hinaus eine zusätzliche Einschränkung durch genaue Festlegung, welches konkrete Markup-Vokabular und in welcher Reihenfolge verwendet werden darf. Diese Unterscheidung ermöglicht es, Werkzeuge zu entwickeln, die die allgemeine XML-Struktur effizient lesen, verarbeiten und speichern können. Diese Werkzeuge können ebenso für die spezifischen XML-Sprachen in den verschiedenen Anwendungsgebieten genutzt werden.

XML kann somit nicht als isoliertes Konzept oder Technik betrachtet werden. Vielmehr ergibt das Zusammenspiel mit anderen XML-Standards und die Einbettung in unterschiedlichste Applikationen eine gemeinsame Infrastruktur. Auf der anderen Seite stellt XML aber auch kein geschlossenes monolithisches System dar. Die einzelnen Vertreter der Sprachfamilie sind derart modular aufgebaut, dass sie beständig und unabhängig voneinander weiterentwickelt werden. Oftmals stehen dem Anwender verschiedene Alternativen zur Lösung konkreter Problemen bereit, die wahlweise herangezogen werden können.

Rückblickend auf den Erfolg von XML lässt sich zusammenfassend sagen, dass

- XML einerseits standardisiert genug ist, um eine Vielzahl von Werkzeugen zur Verarbeitung von XML-Dokumenten einsetzen zu können,
- XML andererseits flexibel genug ist, um jedem Anwenderkreis zu ermöglichen, eine eigene, anwendungsspezifische XML-Sprache zu definieren und deren Instanzen zu nutzen.

Dieses Kapitel bietet im Weiteren einen Einstieg in die für das Verständnis der vorliegenden Arbeit wesentlichen Basiskonzepte von XML. Der Abschnitt 2.1 stellt die historische Entwicklung der expliziten Textauszeichnung dar und motiviert zugleich deren Verwendung. Im Abschnitt 2.2 wird zunächst auf die grundlegenden syntaktischen Konstrukte von XML eingegangen, bevor im Abschnitt 2.3 Möglichkeiten vorgestellt werden, wie mit Hilfe von Schemasprachen eigene XML-Vokabulare definiert werden können. Im Abschnitt 2.4 wird von der konkreten XML-Syntax

abstrahiert und ein abstraktes Modell vorgestellt, bei dem die Daten im Vordergrund stehen. Abschließend werden die bis dahin intuitiven Begriffe definiert und in einem Begriffsnetz in Beziehung gesetzt.

2.1 Ursprung und Entwicklung

Der Ursprung von Markups im weiteren Sinne geht auf die Zeit der Erfindung des Buchdruckes zurück. In dessen Anfängen wurde das Blockdruckverfahren eingesetzt, bei dem jeweils eine komplette Seite aus Holztafeln geschnitten und abgezogen wurde. Der klassische Buchdruck, dessen Entwicklung in Europa Mitte des 15. Jahrhunderts Johannes Gutenberg zugeschriebenen wird, ist ein Verfahren, welches mit beweglichen Lettern oder Typen arbeitet. Diese Lettern wurden so aneinandergereiht, dass sie eine Seite ergaben. Die Lettern wurden anschließend mit Farbe bestrichen und auf Papier mit Druck übertragen (daher auch der geläufige Name Hochdruckverfahren). Im Gegensatz zu den bis dahin eingesetzten Druckplatten aus Holz, die Unikate darstellten und nur für eine Buchseite genutzt werden konnten, konnten die einzelnen Lettern wiederverwendet werden. Diese Form des Buchdruckes hielt sich bis Anfang des 20. Jahrhunderts in fast unveränderter Form. Später wurden automatische Schriftgießmaschinen eingeführt, die einzelne Lettern (z. B. Monotype) oder ganze Zeilen (z. B. Linotype, Ludlow, Typograph) gossen und nach deren Gebrauch wieder einschmelzen konnten. Diese Verbesserung mechanisierte die Arbeit des Setzers und beschleunigte somit den traditionellen Ablauf. Allerdings änderte sich das Prinzip der Bleisatz-Technik nicht. Erst Mitte des 20. Jahrhunderts wurde es durch den Fotosatz, der mit Hilfe von optischen Belichtungsverfahren i. d. R. die Vorlage für den Offsetdruck bildet, abgelöst. Während beim Offsetdruck noch Druckplatten produziert werden müssen, verzichtet man beim Digitaldruck völlig auf die Herstellung von Druckvorlagen.

In allen Epochen innerhalb der Geschichte des Buchdruckes wurden Markups (Textauszeichnungen) eingesetzt. So wurden in der Phase des Bleisatzes die Manuskripte (lat. *manu scriptum* = von Hand Geschriebenes) mit Formatierungshinweisen durch den Autor bzw. einen Textgestalter versehen, damit der Schriftsetzer die Druckplatten sowie später die Belichtungsvorlagen nach bestimmten typographischen Richtlinien wie Schriftart, Zeilenabstand, Zeilenbreite, aber auch Fett- oder Kursivschrift usw. erstellen konnte. Diese Gestaltungsmerkmale wurden zunächst handschriftlich am Text vermerkt. Später, mit der fotomechanischen Umsetzung des Druckes sowie mit dem Einzug erster Großrechner in den 1960er und frühen 1970er Jahren, ermöglichten Lochstreifen eine rudimentäre Seitenbeschreibungssprache für eine detailliertere Erfassung von Texten. Mit dem weiteren Fortschritt der EDV-Systeme wurde in den 1980er Jahren das heute unter dem Namen *Desktop Publishing* (DTP) bekannte Verfahren eingeführt, welches elektronische Markups, ja sogar komplette Seitenbeschreibungssprachen wie PostScript® [Inc99] zur Formatierung von Texten und Dokumenten einsetzt.

Die Suche nach einer Definition von Markups führt bei IT-Wissen, einem Online-Lexikon für Informationstechnologie [IT-08], zu folgendem interessanten Ergebnis:

> „Ein Markup ist eine Auszeichnung, eine Markierung. Markups, das sind Folgen von Symbolen und Zeichen, die an bestimmten Stellen in einem Dokument eingefügt werden, um die Form der Darstellung, des Druckes oder der Struktur der Dokumente zu beschreiben. Das können auch Kommentare sein, Trennzeichen und Verarbeitungsanweisungen. [...]
> Die einzelnen voneinander getrennten Markup-Elemente nennt man Tags.“

Diese Definition von **Markups** greift zwei verschiedene Aufgaben der Textauszeichnung auf. Zum einen wird die Textauszeichnung entsprechend ihrer ursprünglichen Funktion zur Darstellung von bestimmten Wörtern, Sätzen oder Abschnitten eines Textes, u. a. am Drucker oder Bildschirm, verwendet. Da diese Form des Markups die formatorientierte Verarbeitung von bestimmten Textbestandteilen beschreibt, wird es in der Literatur oftmals als **prozedurales Markup** bezeichnet. Zum anderen können Markups Texte mit Informationen ergänzen, die strukturelle und logische Aspekte eines Dokumentes (z. B. die Einteilung in Überschriften, Kapiteln, Abschnitte, usw.) beschreiben, ohne jedoch eine genaue Repräsentation oder eine andere Verarbeitung der Inhalte festzulegen. Aus diesem Grund werden diese Textauszeichnungen häufig unter dem Begriff **deskriptives** oder auch **generisches Markup** zusammengefasst.

Generisches Markup

Die wesentliche Idee, den Informationsgehalt eines Dokumentes von seiner äußeren Form zu trennen, geht auf William Tunnicliffe Ende der 1960er Jahre zurück [Gol90]. Auf Basis dieser Idee, welche unter dem Namen *generic coding* veröffentlicht wurde, entwickelten Charles Goldfarb, Edward Mosher und Raymond Lorie die GML (*Generalized Markup Language*). GML definiert eine einheitliche Notation für Markups, die jedoch deren Bestandteile, wie z. B. die Namensgebung der Tags, flexibel lässt. Darüber hinaus enthielt GML erstmals das Konzept eines formal definierten Dokumenttyps mit einer verschachtelten Struktur. GML und dessen Dokumenttypkonzept wurden im Laufe der Jahre stetig weiterentwickelt und letztendlich 1986 als ISO 8879 unter dem Namen SGML (*Standard Generalized Markup Language*) standardisiert [Int86].

In SGML erfolgt die Beschreibung eines Dokumenttyps innerhalb der DTD (*Document Type Definition*). Die DTD spezifiziert, welche konkreten Markups ein Text enthalten darf, welche Markups unbedingt vorhanden sein müssen und in welcher Reihenfolge diese auftreten. D. h., es ist mit SGML möglich, eine Menge von Markups in Form einer individuellen **Markup-Sprache**, auch als Auszeichnungssprache bezeichnet, zu definieren. Die Markup-Sprache weist dabei dem Vokabular an Textauszeichnungen eine feste Semantik zu. Die konkrete Benennung der Textauszeichnungen kann an die Terminologie der Anwendungsdomäne, in der die Markup-Sprache zum Einsatz kommt, angelehnt werden. Bekannte Beispiele aus SGML definierter Markup-Sprachen sind HTML [RHJ99] (Standard zur Strukturierung von Web-Seiten) oder DocBook [WM06] (Standard zur Erstellung von technischen Dokumenten).

Aufgrund der Komplexität durch die große Flexibilität beschränkte sich das Einsatzgebiet von SGML im Wesentlichen auf die professionelle Verarbeitung technischer Dokumente. Erst die Vereinfachung von SGML zu XML (*Extensible Markup Language* - standardisiert im Jahre 1998 [BPS+06a], [BPS+06b]) durch die Fokussierung auf Funktionen, die für Dokumente im Internet

benötigt werden, führte zu einer starken Verbreitung im Web, später aber auch auf anderen Anwendungsfeldern. Heute ist XML der Standard zur Erstellung von strukturierten, maschinen- und menschenlesbaren Dokumenten. Darüber hinaus entstanden im Umfeld von XML zahlreiche Sprachen, die häufig benötigte allgemein auftretende Aufgaben bzgl. XML anbieten (wie etwa Transformation von XML-Dokumenten, Adressierung von Informationen in XML-Dokumenten, APIs für XML-Datenstrukturen usw.), aber auch Parser und Werkzeuge, die diese Sprachen sowie APIs implementieren. Die durch die XML-Familie erzeugte Infrastruktur ist die Grundlage für eine weiterreichende Nutzung von XML in noch nicht erschlossenen Anwendungsgebieten und Domänen.

2.2 Strukturelle Grundkonzepte

XML (*Extensible Markup Language*) stellt ein gegenüber SGML vereinfachtes Regelwerk zur Verfügung, mit dem es ermöglicht wird, konkrete, individuelle Markup-Sprachen zu definieren. XML hält zu diesem Zweck syntaktische Konventionen bereit, die festlegen, aus welcher Zeichenfolge ein Markup besteht und an welchen Stellen innerhalb eines Textes ein Markup hinzugefügt werden darf. Eine genauere Beschreibung des XML-Vokabulars einer Markup-Sprache kann mit Hilfe einer aus der SGML vereinfachten DTD spezifiziert werden. Eine DTD schränkt durch bestimmte grammatische Regeln die Markups (z. B. die Namensgebung oder deren Auftretensreihenfolge) ein.

Der folgende Beispielcode 2.1 zeigt eine mögliche Textauszeichnung mit XML eines Glossars in der bspw. die Definition von Markup (siehe Seite 12) eingetragen wird. Mit Hilfe der expliziten Textkennzeichnung durch XML-Markups wird es Software-Applikationen ermöglicht, Zeichendaten zu unterscheiden, zu suchen und auszuwerten.

```
1  <?xml version="1.1" encoding="ISO-8859-1"?>
2  <!DOCTYPE glossar SYSTEM "glossar-beispiel.dtd">
3  <glossar quelle="http://www.itwissen.info">
4    <eintrag lokation="/definition/lexikon//_markup_markup.html" kategorie="Datentyp">
5      <begriff>Markup</begriff>
6      <definition>
7       Ein Markup ist eine Auszeichnung, eine Markierung. Markups, das sind Folgen
8       von Symbolen und Zeichen, die an bestimmten Stellen in einem Dokument eingefügt
9       werden, um die Form der Darstellung, des Druckes oder der Struktur der Dokumente
10      zu beschreiben. Das können auch Kommentare sein, Trennzeichen und
11      Verarbeitungsanweisungen.
12       ...
13      Die einzelnen voneinander getrennten Markup-Elemente nennt man Tags.
14     </definition>
15   </eintrag>
16   <!--Weitere Einträge folgen.-->
17 </glossar>
```

Code 2.1: Glossar mit XML-Markup.

2.2.1 Aufbau

Sämtliche im Beispielcode 2.1 fett dargestellten Zeichen des Dokumentes bilden das XML-Markup. Die restlichen Zeichen des Textes, die kein Markup sind, werden als Zeichendaten bezeichnet. Prinzipiell handelt es sich bei dem Beispielcode um ein reines Textdokument ohne jegliche Formatierungen. So wird bspw. die Fett-Darstellung lediglich zur visuellen Kennzeichnung des Markups verwendet.

Ein XML-Dokument kann mit einem Prolog beginnen. Ein Prolog kann aus einer XML-Deklaration, einer Dokumenttyp-Deklaration sowie weiteren Kommentaren und Verarbeitungsanweisungen bestehen. Eine XML-Deklaration gibt, wie im Beispielcode 2.1 in Zeile 1, die verwendete XML-Version sowie die verwendete Zeichenkodierung an. Weitere Informationen sind möglich. Eine Dokumenttyp-Deklaration besteht entweder aus einem Verweis auf eine externe Teilmenge (wie in Zeile 2), die die Markup-Deklarationen enthält, oder aus einer internen Teilmenge von Markup-Deklarationen oder aus beidem. Beide Teilmengen zusammen bilden eine Grammatik für eine Klasse von XML-Dokumenten.

Wie das Beispiel zeigt, verwendet XML für die Markups im Prolog sowie auch für die meisten anderen Markups spitze Klammern (< und >), um sie syntaktisch von den eigentlichen Zeichendaten des Textes zu trennen. In Zeile 4 wird bspw. der Glossareintrag Markup explizit mit <begriff>Markup</begriff> ausgezeichnet. Dadurch kann der Begriff von der eigentlichen Definition unterschieden werden. Die Zeichen <begriff> und </begriff> sind Bausteine des wichtigsten Markups innerhalb von XML-Dokumenten – den Elementen. Der erste Baustein, er wird auch Start-Tag genannt, und der zweite Baustein, das End-Tag, grenzen das Element nach außen und innen ab. Der abgegrenzte Text zwischen Start- und End-Tag bildet den Inhalt eines Elementes. Der Inhalt kann wiederum Zeichendaten, (Unter-)Elemente oder andere Markups enthalten. Er kann aber auch leer sein. Diese Struktur von XML bedingt, dass ein End-Tag immer zu dem zuletzt geöffneten Start-Tag passen muss. Bei einem korrekt gebildeten XML-Dokument entsteht dadurch eine strikt hierarchische Elementstruktur. Das äußerste Element wird Wurzelelement genannt.

Ein Element kann durch Attribute mit zusätzlichen Informationen angereichert werden. In der Zeile 2 bspw. enthält das Element glossar ein Attribut quelle="http://www.itwissen.info", welches Informationen über die Quelle des Glossars festhält. Ein Attribut besteht immer aus einem Name-Wert-Paar: dem Attributnamen und dem Attributwert. Attribute werden im Start-Tag des Elementes abgelegt. Ein Attribut darf dabei, im Gegensatz zu möglichen Unterelementen, nur einmal im abgelegten Element auftreten und unterliegt keiner Reihenfolge. Die Groß- und Kleinschreibung der Namen für Elemente und Attribute in den Elementen ist sensitiv und muss daher beachtet werden.

Im Beispielcode 2.1 ist ein weiteres Markup – der Kommentar – enthalten (siehe Zeile 16). Kommentare sind durch die Zeichenfolge <!-- und --> begrenzt. Kommentare sind Annotationen, die Hinweise für den Textbearbeiter und Nutzer geben. Sie sollen die Lesbarkeit und Nachvollziehbarkeit eines XML-Dokumentes erhöhen. Kommentare können (aber müssen nicht) von einer Applikation verarbeitet werden.

Weitere Markups, die mit eckigen Klammern begrenzt sind, aber nicht im Beispielcode 2.1 auftreten, sind Verarbeitungsanweisungen (<? und ?>) und CDATA-Abschnitte (<![CDATA[und]]>). Verarbeitungsanweisungen sind spezielle Anweisungen für Applikationen, die z. B. Informationen zur Darstellung der XML-Daten enthalten. Die Applikationen werden durch das Ziel, mit der jede Verarbeitungsanweisung beginnt, identifiziert. Das Ziel steht dabei für einen symbolischen Namen, den die adressierte Applikation kennen muss. Die restlichen Zeichen innerhalb der Verarbeitungsanweisung sind die Daten, die der Applikation übergeben werden. Häufig werden diese Daten als Pseudoattribute abgelegt, die ähnlich wie ein Attribut aussehen, aber nicht als solches von einem Parser behandelt werden. Obwohl die XML-Deklaration die gleichen Trennzeichen verwendet, ist sie keine Verarbeitungsanweisung. Aus diesem Grund darf das Ziel einer Verarbeitungsanweisung auch nicht aus den Literalen xml oder XML gebildet werden.

Einige Zeichen und Zeichenfolgen, z. B. eine beginnende eckige Klammer, sind innerhalb der Zeichendaten nicht erlaubt, da sie als XML-Markup falsch interpretiert werden könnten. Eine Möglichkeit diese Zeichenfolgen zu maskieren, bieten CDATA-Abschnitte. Per Definition dürfen in diesen CDATA-Abschnitten alle Zeichen verwendet werden, die nicht das Ende dieses Abschnittes (]]>) kennzeichnen.

Treten einzelne solcher Zeichen innerhalb von Zeichendaten auf, so können diese ebenso mit Hilfe von sogenannten Entity-Referenzen maskiert werden. Eine Entity-Referenz verweist auf den Inhalt eines benannten Entitys. Dieser Inhalt wird, vergleichbar mit einem einfachen Textersetzungsmechanismus, an die Stelle der Referenz eingesetzt. Da Entity-Referenzen, im Gegensatz zu CDATA-Abschnitten, auch in Attributwerten eingesetzt werden dürfen, verwenden sie eine andere Syntax zur Kennzeichnung des Markups, um falsche Interpretationen zu vermeiden. Referenzen zu analysierten allgemeinen Entities bestehen aus dem et-Zeichen (&), dem Namen des referenzierten Entitys und dem Semikolon (;). So bedeutet bspw. die Notation <, dass diese Referenz durch den Inhalt des vordefinierte Entitys lt, in diesem Fall durch das Zeichen <, ersetzt wird. Die Zeichen & und ; sind Trennzeichen zur Abgrenzung der Entity-Referenz. Soll das et-Zeichen nicht als Trennzeichen verwendet werden, muss es selbst maskiert werden. XML stellt dafür das vordefinierte Entity amp bereit. Weitere vordefinierte Entities sind bspw. lt (<), apos (’) und quot (”). Neben den vordefinierten können ebenso benutzerdefinierte Entities einem XML-Dokument hinzugefügt werden. Die Entities werden in der Dokumenttyp-Deklaration definiert. Beispielsweise definiert die Notation <!ENTITY land ”049”> ein Entity land mit dem Inhalt 049.

Für spezielle Zeichen, die bspw. nicht direkt über die Tastatur in den Text eingegeben oder die nicht in der Kodierung des Dokumentes repräsentiert werden können, stellt XML Zeichenreferenzen zur Verfügung. Eine Zeichenreferenz wird aus der Zeichenfolge (&#), einem entsprechenden Zeichencode und einem Semikolon (;) gebildet. Der Zeichencode verweist auf eine atomare Texteinheit gemäß der Spezifikation von ISO/IEC 10646 [Int93, Int00]. Damit sind Zeichen des gesamten Unicode-Bereiches [Uni96, Uni00] mit einigen wenigen Ausnahmen innerhalb der Zeichendaten erlaubt. Beispielsweise steht die Zeichenkodierung α (dezimal) als auch α (hexadezimal) für das kleine griechische Zeichen α. Obwohl Zeichenreferenzen eine

ähnliche Notation wie Entity-Referenzen besitzen, zählen sie nicht zum Markup.

2.2.2 Strukturierung

Der Beispielcode 2.1 zeigt nur eine mögliche Strukturierung eines Glossars. In Abhängigkeit der Verwendung des Dokumentes kann eine regelmäßige oder aber auch eine unregelmäßige Strukturierung vorgenommen werden. Eine wie in unserem Beispiel sehr regelmäßige Struktur, in der Elemente entweder nur Elemente oder nur Zeichendaten enthalten dürfen, wird **datenzentrierte** Struktur genannt. Dem gegenüber wird eine unregelmäßige Strukturierung mit Elementen die sowohl Zeichendaten als auch Elemente enthalten können, unter dem Begriff **dokumentzentriert** zusammengefasst. Da XML zum einen beide Strukturierungsmöglichkeiten und zum anderen keine oder offene Schemabeschreibungen erlaubt, werden die mit XML-Markups gekennzeichneten Zeichendaten als **semistrukturierte** Daten bezeichnet.

2.2.3 Korrektheit

XML-Dokumente werden in wohlgeformte und gültige Dokumente unterschieden. Genügt ein XML-Dokument den syntaktischen Regeln für XML, so wird es als **wohlgeformt** (*well-formed*) bezeichnet. Werden darüber hinaus weitere Einschränkungen eingehalten, die in einem dazugehörigen Schema (z. B. in einer DTD oder einer anderen Schemasprache) formuliert sind, wird von einem **gültigen** (*valid*) XML-Dokument gesprochen. Ein gültiges Dokument setzt immer ein wohlgeformtes Dokument voraus und ist damit ein stärkeres Korrektheitskriterium.

Neben dem XML-Dokument wird in der Literatur oftmals eine weichere Form genannt – das XML-Fragment. Ein XML-Fragment muss, im Gegensatz zum XML-Dokument, kein äußerstes Wurzelelement besitzen, welches alle anderen Elemente umschließt. Ein XML-Fragment ist dann bereits wohlgeformt, wenn durch das Hinzufügen eines umschließenden Elementes ein wohlgeformtes XML-Dokument entstehen würde.

2.2.4 Namensraum

Elemente und Attribute in XML-Fragmenten oder -Dokumenten können, obwohl sie aus verschiedenen Anwendungskontexten kommen und eine unterschiedliche Semantik besitzen, identische Bezeichner verwenden. Namensräume bieten eine Möglichkeit, potentielle Namenskonflikte bspw. bei einer Mischung von Dokumenten zu vermeiden. Ein **Namensraum** (*namespace*) erlaubt es, Elementen und Attributen einem bestimmten Geltungsbereich, der über die Dokumentgrenzen hinausgeht, zuzuweisen, der sie eindeutig und damit unterscheidbar macht. Als Geltungsbereich wird eine URI (*Uniform Resource Identifier*) eingesetzt. Dieser Verweis dient lediglich dazu, die Eindeutigkeit sicherzustellen. Hinter der URI kann sich aber auch eine sinnvolle Ressource verbergen, die interpretiert werden kann.

Bevor ein Namensraum innerhalb eines XML-Dokumentes benutzt werden kann, muss er deklariert werden. Für diese Deklaration stellt XML spezielle Attribute zur Verfügung, deren Namen mit dem reservierten xmlns: beginnen. Die restlichen Zeichen des Attributnamens vereinbaren ein Kürzel, dem im Attributwert eine URI zugeordnet wird. Beispielsweise deklariert xmlns:xs="http://www.w3.org/2001/XSLSchema" das Kürzel xs für den Namensraum http://www.w3.org//2001/XSLSchema. Das Kürzel kann in Attributen und Elementen als Präfix vorangestellt werden, um sie an einen bestimmten Namensraum zu binden. Der Name, der durch das Voranstellen des Präfixes entsteht, wird als qualifizierter Name bezeichnet. So wird bspw. im qualifizierten Elementnamen xs:schema der Namensraum von xs an den lokalen Elementnamen schema gebunden.

Da sich eine explizite Angabe des definierten Namensraumpräfixes für jedes Element und dessen Unterelemente u. U. als platzraubend und umständlich erweisen kann, bietet XML eine kompaktere Variante, in der kein Kürzel deklariert wird. Hierzu wird das reservierte Attribut xmlns ohne den separierenden Doppelpunkt und ohne nachfolgendes Kürzel verwendet. Der so deklarierte Namensraum wird Vorgabenamensraum (*default namespace*) genannt. Dieser umfasst neben dem Element, welches die Namensraumdeklaration beinhaltet, auch alle Unterelemente. Eine Ausnahme hiervon bilden die untergeordneten Elemente, in denen eine Redeklaration des Vorgabenamensraums erfolgt oder die durch einen expliziten Präfix an einen anderen Namensraum gebunden werden. Die Namensräume auf Elementebene werden nicht an die Attribute propagiert. Attribute können nur explizit einem Namensraum zugeordnet werden. Entscheidend zur Identifikation des Namensraums ist allein die jeweilige URI; unabhängig davon, ob die Zuordnung durch einen Vorgabenamensraum oder durch eine explizite Präfixangabe erfolgte. Ein Namensraum wird dann als gleich angesehen, wenn alle Zeichen dieser zugeordneten URI identisch sind. Dieses Konzept der Namensräume wurde nachträglich zu XML hinzugefügt und wird in einer eigenen Spezifikation beschrieben [BHLT06].

2.3 Schemasprachen

Schemasprachen ermöglichen mit Hilfe einer Grammatik oder bestimmter Regeln, eine Klasse von XML-Dokumenten – eine Markup-Sprache – formal zu definieren. Je nach Schemasprache können dabei sowohl die erlaubte Elementstruktur sowie deren zugehörige Attribute als auch die darin verwendeten Typen festgelegt werden. Mit der Festlegung eines gültigen Vokabulars für XML-Dokumente wird immer, wenn auch implizit, eine dazugehörige Semantik definiert, die den Sinn und Zweck der Elemente und Attribute beschreibt. Widerspricht ein XML-Dokument dieser semantischen Verwendung, wird dies als Tag-Missbrauch (*tag abuse*) bezeichnet. Ein solcher Verwendungsmissbrauch kann fast nicht automatisiert erkannt werden, da die Bedeutung der Elemente und Attribute mittels XML nicht explizit ausgedrückt werden kann. Nichtsdestotrotz sind Schemasprachen die Voraussetzung, um auf ein eigenes gemeinsames XML-Vokabular aufzubauen und somit Grundlage des vielfältigen Einsatzes von XML.

2.3.1 Dokumenttyp-Definition

XML selbst stellt eine Sprache zur Formulierung von XML-Grammatiken zur Verfügung – die Dokumenttyp-Definition (DTD). Die DTD wird durch eine Menge von Markup-Deklarationen gebildet, die intern, extern oder durch eine Kombination aus beidem spezifiziert werden. Innerhalb einer DTD werden vier Arten von Markup-Deklarationen unterstützt. Mit Hilfe der Elementtyp-Deklaration (<!ELEMENT ...>) wird beschrieben, welche Typen von Elementen es gibt und welche Inhalte diese besitzen. Der Inhalt kann dabei sowohl aus Zeichendaten als auch aus Kindelementen sowie aus einer Mischung aus beidem bestehen. Eine nähere Definition des textuellen Inhaltes durch Festlegung von zulässigen Datentypen ist jedoch nicht möglich. Kindelemente können jedoch durch eine EBNF-angelehnte Notation gruppiert sowie als Sequenz, Option, Alternative oder Wiederholung gekennzeichnet werden. Die Attribute der Elementtypen werden durch Attributlisten-Deklarationen (<!ATTLIST ...>) festgelegt. Eine Attributliste enthält einerseits eine Referenz auf einen deklarierten Elementtyp und andererseits Name-Wert-Paare, denen ein Existenzkriterium (zwingend oder optional) zugewiesen wird. Zudem können zum Attributtypen passende Vorgabewerte angegeben werden. Als Attributtypen innerhalb einer DTD werden Zeichendaten (`CDATA`) und Aufzählungen sowie Typen für Eindeutigkeits- sowie Referenzmechanismen unterstützt. Der Beispielcode 2.2 (glossar.dtd) zeigt ein Schema in DTD zum Beispiel 2.1 von Seite 14.

```
<?xml version="1.1" encoding="ISO-8859-1"?>          1
<!ELEMENT glossar (eintrag*)>                        2
<!ELEMENT eintrag (begriff,definition)>              3
<!ELEMENT begriff (#PCDATA)>                         4
<!ELEMENT definition (#PCDATA)>                      5
<!ATTLIST glossar                                    6
    quelle CDATA #required>                          7
<!ATTLIST eintrag                                    8
    lokation CDATA #required>                        9
```

Code 2.2: Schema des Glossars mit DTD.

Neben den bisher vorgestellten Markup-Deklarationen ermöglicht DTD ebenso die Deklaration von Entities und Notationen. Entity-Deklarationen (<!ENTITY ...>) sind mit einem Namen versehene Textbausteine, auf die mit Parameter-Entity-Referenzen innerhalb eines DTD-Schemas sowie aus den Instanzen (XML-Dokumente) verwiesen werden kann. Die Schreibweise der Parameter-Entity-Referenzen unterscheidet sich von den Entity-Referenzen, die nur in den Instanzen zulässig sind, dadurch, dass statt des et-Zeichen (&) nun ein Prozentzeichen (%) als linkes Abgrenzungszeichen dient. Die Textbausteine der Entities können sowohl intern als auch extern definiert werden. Ebenso wird in analysierte (Textbausteine mit XML) und nicht-analysierte Entities (Textbausteine ohne XML-codierten Inhalt wie bspw. Binärdaten) unterschieden. Notationen geben in diesem Zusammenhang zusätzliche Informationen über nicht-analysierte Entities. Die Notation <!NOTATION TEX PUBLIC "ISBN 0-201-13448-9::/NOTATION TeX//EN"> beschreibt bspw. die Art eines Textformates.

2.3.2 XML Schema

Aufgrund der historischen Entwicklung, die DTD wurde ursprünglich für SGML entwickelt, besitzt diese Schemasprache insbesondere für den datenzentrierten Einsatz einige Beschränkungen und Schwächen. Wie bereits erwähnt, bietet DTD kaum Unterstützung für Datentypdefinitionen sowie keinerlei Möglichkeiten, Wertebereiche einzuschränken. Darüber hinaus werden nur Strukturierungsmechanismen einer EBNF unterstützt. Diese erlauben bspw. nicht, die exakte minimale und maximale Anzahl von Elementtypen festzulegen. Namensräume werden ebenfalls nicht mit einbezogen. Zur Behebung dieser Schwächen wurden alternative Ansätze zur DTD vorgeschlagen. Eine von diesen ist die vom W3C standardisierte Sprache XML Schema.

XML Schema erweitert die dokumentzentrierte Sichtweise von DTD durch zusätzliche Sprachbestandteile zur Beschreibung von Dateninhalten. So bietet XML Schema 19 Datentypen sowie diverse Ableitungen dieser an. Die Datentypen lassen sich einschränken und erweitern. Zu den einfachen Datentypen gehören bspw. String, Boolean, verschiedene Ganzzahl- und Gleitkommazahltypen, URI und verschiedene Datums- und Zeittypen. Alle Datentypen werden im zweiten Teil der XML Schema-Spezifikation beschrieben [BM04]. Darüber hinaus besteht die Möglichkeit, komplexe Datentypen zu definieren und mittels Referenzmechanismen wieder zu verwenden. Ebenso lassen sich genaue Kardinalitäten bzgl. Elementtypen angeben. Sämtliche Sprachbestandteile, mit denen Strukturen formuliert werden, sind in [TBMM04] spezifiziert. Durch die Unterstützung von Namensräumen können innerhalb von XML-Dokumenten Elemente, deren Elementtypen in verschiedenen Schemas festgelegt werden, verwendet werden.

Ein Schema in XML Schema wird selbst in XML-Syntax notiert und kann deshalb mit den gleichen Werkzeugen wie andere XML-Dokumente bearbeitet werden. Das XML-Vokabular, welches zur Erstellung eines Schemas bereitgestellt wird, ist wiederum durch ein konkretes Schema in XML Schema formal definiert (siehe Abbildung 2.6). Der Code 2.3 zeigt eine Schemadefinition mit XML Schema zum Beispiel 2.1 von Seite 14. Jedes Schema besteht aus einem Container-Element `schema`, welches sämtliche weiteren Elemente zur Beschreibung einer Klasse von XML-Dokumenten enthält. Von Zeile 4–11 wird bspw. die Struktur des `glossar`-Elementes festgelegt. Das `glossar`-Element darf demnach eine Sequenz aus `eintrag`-Elementen (mindestens eines und maximal zehn) als Kindelementen haben und muss ein `quelle`-Attribut besitzen. Der Typ des `eintrag`-Elementes wird wiederum über eine Referenz in Zeile 13–19 definiert. Es besteht dementsprechend aus den Kindelementen `begriff` und `definition` sowie einem `lokation`-Attribut.

Wie der Beispielcode 2.3 veranschaulicht, hat die Fülle von Möglichkeiten, die Struktur und Datentypen von XML-Dokumenten zu definieren, auch Nachteile. Schemadefinitionen in XML Schema sind im Vergleich zu Schemadefinitionen in DTD sehr viel größer, unübersichtlicher und weniger intuitiv verständlich.

Neben den vorgestellten Schemasprachen wurden weitere vielversprechende Alternativen wie z. B. XDR [LJM+98], SOX [DFH+99] oder RELAX NG [Int03] entwickelt. Trotz der verschiedensten Sprachvorschläge kommt jedoch dem XML Schema vom W3C die größte praktische Bedeutung zu. Insgesamt bildet das XML Schema zusammen mit der XML-Spezifikation und den

Namensräumen die Basis aller weiteren W3C-XML-Sprachstandards. Aufgrund des vollkommen anderen Ansatzes soll an dieser Stelle abschließend die regelbasierte Schemasprache Schematron vorgestellt werden. Ein Vergleich zwischen den vorgestellten und weiteren Schemasprachen kann z. B. [LC00], [Jel01] oder [Rog06] entnommen werden.

```
<?xml version="1.1" encoding="ISO-8859-1"?>                                                    1
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">                                       2
                                                                                               3
  <xs:element name="glossar">                                                                  4
    <xs:complexType>                                                                           5
      <xs:sequence>                                                                            6
        <xs:element name="eintrag" type="eintragT" minOccurs="1" maxOccurs="10"/>              7
      </xs:sequence>                                                                           8
    </xs:complexType>                                                                          9
    <xs:attribute name="quelle" type="xs:string" use="required"/>                              10
  </xs:element>                                                                                11
                                                                                               12
  <xs:complexType name="eintragT">                                                             13
    <xs:sequence>                                                                              14
      <xs:element name="begriff" type="xs:string"/>                                            15
      <xs:element name="definition" type="xs:string"/>                                         16
    </xs:sequence>                                                                             17
    <xs:attribute name="lokation" type="xs:string" use="required"/>                            18
  </xs:complexType>                                                                            19
                                                                                               20
</xs:schema>                                                                                   21
```

CODE 2.3: Schema des Glossars mit XML Schema.

2.3.3 Schematron

Mit der Schemasprache Schematron [Int06] wird die erlaubte Struktur von XML-Dokumenten nicht auf Basis eines grammatikbasierten Schemas, sondern durch explizite Regeln beschrieben. Diese Regeln spezifizieren Abhängigkeiten zwischen Elementen und Attributen, die erfüllt sein müssen. Durch explizite Regeln können Einschränkungen definiert werden, die weit über die Mächtigkeit von XML Schema hinausgehen. Beispielsweise können mit vergleichsweise einfachen Mitteln gegenseitige Ausschlüsse von Elementen und Attributen oder andere komplexere Validierungsoperationen beschrieben werden. Ebenso gibt es die Möglichkeit, Fehlermeldungen zu formulieren, falls die Regeln nicht erfüllt werden.

Da allerdings sämtliche Beziehungen (z. B. auch einfache Vater-Kind-Beziehungen) sowie Datentypen explizit spezifiziert werden müssen, kann ein solches regelbasiertes Schema schnell anwachsen. Insbesondere die fehlenden vordefinierten Datentypen und die fehlende Unterstützung zur Erstellung von benutzerdefinierten, komplexeren Datentypen ist ein erheblicher Nachteil im Gegensatz zu grammatikbasierten Ansätzen. Der Code 2.4 zeigt dieses Problem für das einfache Beispiel 2.1 deutlich.

Ein Schema in Schematron besteht aus dem Wurzelelement `schema`. Dieses kann u. a. Regelmuster (`pattern`), die mehrere Regeln gruppieren, enthalten. Die gruppierten Regeln in den

Regelmustern können bspw. in verschiedenen Validierungsphasen aktiviert oder deaktiviert werden. Ein Regel (`rule`) selbst besteht aus mehreren Ausnahmen (`report`) sowie Zusicherungen oder Annahmen (`assert`), die für einen bestimmten Kontext der Regel (XPath-Ausdruck [CD99] des `context`-Attributes) gelten sollen. Wird eine Ausnahme bzgl. dieses Kontextes erfüllt oder eine Annahme nicht erfüllt (der boolesche Ausdruck im Attribut `test` ist respektive wahr oder falsch), wird die jeweilige Fehlermeldung im Inhalt der Elemente ausgegeben.

```
1  <sch:schema xmlns:sch="http://purl.oclc.org/dsdl/schematron">
2
3   <sch:pattern name="glossar-regeln">
4     <sch:rule context="/">
5       <sch:assert test="glossar">Das Wurzelelement muss 'glossar' sein.</sch:assert>
6     </sch:rule>
7     <sch:rule context="glossar">
8       <sch:assert test="eintrag">Das 'glossar'-Element muss mindestens ein 'eintrag'-Element als
          Kindelement besitzen.</sch:assert>
9       <sch:assert test="count(eintrag)&lt;11">Ein 'glossar'-Element darf nicht mehr als zehn Einträge
          besitzen.</sch:assert>
10      <sch:report test="count(*[not(name()='eintrag')])&gt;0">Andere Elemente als 'eintrag' sind im Element
          'glossar' nicht erlaubt.</sch:report>
11      <sch:assert test="@quelle">Ein 'glossar'-Element muss ein Attribut 'quelle' besitzen.</sch:assert>
12      <sch:report test="count(@*[not(name()='quelle')])&gt;0">Andere Attribute als 'quelle' sind im 'glossar
          '-Element nicht erlaubt.</sch:report>
13    </sch:rule>
14    <sch:rule context="eintrag">
15      <sch:assert test="*[position()=1 and name()='begriff']">Das 'eintrag'-Element muss als erstes
          Kindelement ein 'begriff'-Element besitzen.</sch:assert>
16      <sch:assert test="*[position()=2 and name()='definition']">Das 'eintrag'-Element muss als zweites
          Kindelement ein 'definition'-Element besitzen.</sch:assert>
17      <sch:assert test="count(*)=2">Es sind nur zwei Kindelemente im 'eintrag'-Element erlaubt.
          </sch:assert>
18      <sch:assert test="@lokation">Ein 'eintrag'-Element muss ein 'lokation'-Attribut haben.</sch:assert>
19      <sch:assert test="count(@*[not(name()='lokation')])=0">Andere Attribute als 'lokation' sind im '
          eintrag'-Element nicht erlaubt.</sch:report>
20    </sch:rule>
21    <sch:rule context="begriff">
22      <sch:assert test="count(*)=0">Das 'begriff'-Element muss ein leeres Element sein.</sch:assert>
23      <sch:assert test="count(@*)=0">Das 'begriff'-Element darf keine Attribute besitzen.</sch:assert>
24    </sch:rule>
25    <sch:rule context="definition">
26      <sch:assert test="count(*)=0">Das 'defintion'-Element muss ein leeres Element sein.</sch:assert>
27      <sch:assert test="count(@*)=0">Das 'definition'-Element darf keine Attribute haben.</sch:assert>
28    </sch:rule>
29   </sch:pattern>
30
31 </sch:schema>
```

Code 2.4: Schema des Glossars mit Schematron.

Beispielsweise wird in den Zeilen 4–7 im Beispielcode 2.4 eine Regel für den Dokumentknoten (`/`) aufgestellt. Die Annahme im Inneren der Regel sichert, dass das einzige Kindelement des Dokumentknotens – das Wurzelelement – den Namen `glossar` trägt. Die Fehlermeldung bei Nichterfüllung steht im Inhalt dieser Annahme. Auf vergleichbare Weise werden für alle Elemente entsprechende Regeln formuliert, die die erlaubte Struktur eines XML-Dokumentes festlegen.

2.4 Abstrakte Datenmodelle

Die im Kapitel 2.2 vorgestellten Grundkonzepte von XML basieren auf dessen Spezifikation [BPS+06a, BPS+06b], in der die konkrete Syntax von XML-Dokumenten definiert wird. Diese Syntax, die bspw. genau festlegt, wie Attribute mit Hilfe von Markups gekennzeichnet werden, ist beim Einlesen sowie Ausgeben der XML-Dokumente relevant. Für die Verarbeitung der enthaltenen Informationen spielt diese Syntax eine untergeordnete Rolle. Hier ist vielmehr eine syntaxunabhängige Struktur der Daten von Bedeutung, die von bestimmten, rein syntaktischen Eigenschaften (z. B. wird der Wert eines Attributes mit Apostroph (') oder mit Anführungszeichen (") begrenzt) abstrahiert.

Ein solches mögliches abstraktes Datenmodell wird vom W3C als XML Information Set (kurz: Infoset) spezifiziert [CT04]. Das Infoset eines XML-Dokumentes besteht aus (mehreren) Informationseinheiten. Eine **Informationseinheit** (*information item*) gibt eine abstrakte Beschreibung eines bestimmten Teils innerhalb eines XML-Dokumentes wider. Insgesamt unterscheidet das Infoset in elf Typen von Informationseinheiten. Informationseinheiten besitzen eine Menge mit ihnen verbundener benannter **Eigenschaften**. Die unterstützten Eigenschaftsarten sind abhängig vom Typ dieser Informationseinheiten. Innerhalb der Spezifikation wird das Infoset in einer modifizierten Baumstruktur modelliert, wobei die Informationseinheiten die Knoten des Baums repräsentieren. Es wird allerdings ausdrücklich darauf hingewiesen, dass dies jedoch keine Anforderung darstellt und alternative bspw. ereignis- oder anfragebasierte Schnittstellen ebenfalls mögliche Informationseinheiten bereitstellen können. Das Infoset selbst enthält oder favorisiert keine Schnittstelle. Die Abbildung 2.1 zeigt ein Infoset des Beispielcodes 2.1 von Seite 14 in der Notation eines UML-Objektdiagramms. Dabei sind die einzelnen Knoten des Infosets als Objekte und die Eigenschaften der Knoten als Attribute dargestellt.

Die schematische Darstellung des Infosets in Abbildung 2.1 ist hinsichtlich zweierlei Dinge vereinfacht. Zum einen werden Zeichendateninformationseinheiten, die ausschließlich Leerraum enthalten, nicht in das Infoset aufgenommen. Solche Zeichendaten treten im zugehörigen Beispielcode 2.1 bspw. zwischen benachbarten Elementen auf. Zum anderen werden Eigenschaften der Informationseinheiten, die mit keinem Wert oder nur mit einem Vorgabewert belegt sind, aus Gründen der Übersichtlichkeit vernachlässigt. Das Beispiel zeigt trotz der Vereinfachung die wesentliche Baumstrukur des Infosets. Die Wurzel des Baums bildet immer der Dokumentknoten, der einen Container für das gesamte Dokument darstellt und deshalb entsprechende Eigenschaften bereithält. Die eigentliche hierarchische Struktur entsteht durch die Elementknoten, da sie Kindknoten haben dürfen. Alle anderen Knotenarten, mit Ausnahme des Dokumentknotens, sind Blätter des Baumes. Eine Sonderstellung unter den Elementknoten nimmt der Wurzelelementknoten ein, da er als einziger Elementknoten eine Kante zum Dokumentknoten besitzt. Alle anderen Elementknoten sind Kinder oder weitere Nachkommen des Wurzelelementknotens. Eine weitere Besonderheit im Beispiel stellen die Attribut- und Namensraumknoten dar. Sie besitzen gegenüber den anderen Knotenarten eine eigene Kante (auch als Achse bezeichnet). Diese Kante unterstützt keine Sortierung im Gegensatz zu den sonstigen Vater-Kind-Kanten. Dementsprechend wird die Reihenfolge der Attribute und der Namensräume innerhalb von Elementen

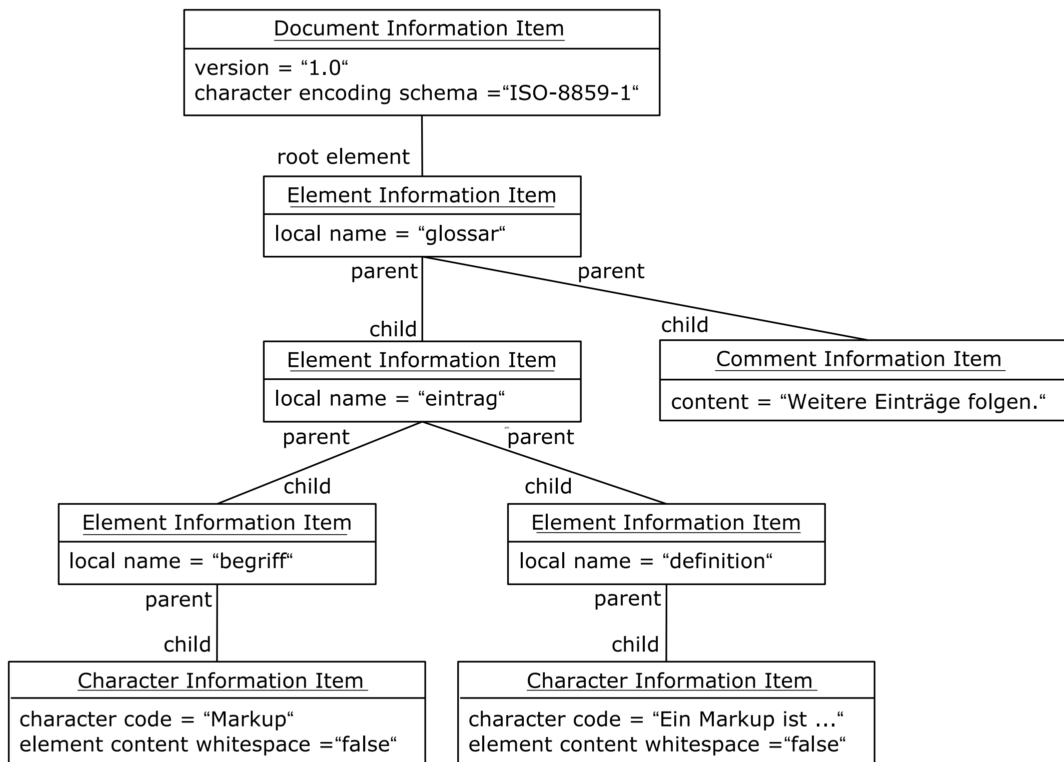


ABBILDUNG 2.1: Vereinfachte schematische Darstellung des Infosets.

im Infoset nicht unterschieden. Neben dieser Einschränkung wird von weiteren Informationen in der aktuellen Version des Infosets abstrahiert. Beispielsweise existieren keine Informationseinheiten für CDATA-Abschnitte, Entity- oder Zeichenreferenzen. Diese werden aufgelöst und in Form von Zeichendateninformationseinheiten repräsentiert. Aufgrund der Abstraktion gehen diese Informationen verloren.

Applikationen, die gemäß dem Infoset ein XML-Dokument verarbeiten, können eine textuelle Darstellung von XML auswählen bzw. eine Repräsentation durch eine andere Repräsentation ersetzen (z. B. die Apostrophe durch Anführungszeichen bei den Attributen). Vielmehr sind andere Formatierungsformate (z. B. effizientere Binärdarstellung von XML-Daten [Wil03] oder [GMNR05]), die dem Infoset entsprechen, im gewissen Sinne XML, obwohl sie nicht mehr in der Syntax übereinstimmen. Sämtliche Daten, die strukturell auf das Infoset abgebildet werden können, können aber in XML-Syntax überführt werden. Damit stehen auch für diese Daten anderer Datenmodelle die große Anzahl von XML-Werkzeugen zur Verfügung. Das Infoset bildet somit den Grundstein zur Interoperabilität zwischen Applikationen.

2.5 Alternativer, sprachbasierter Zugang

Bisher wurden in diesem Kapitel die wesentlichen Basiskonzepte von XML, wie dessen grundlegende syntaktische Sprachkonstrukte, Schemasprachen zur Definition von individuellen Markup-Sprachen sowie abstrakte Datenmodelle, vorgestellt. Die Einführung basierte dabei auf einem intuitiven Begriffsverständnis von Sprachkonstrukten, Sprachen, Metasprachen und Modellen. Im Folgenden soll eine präzise Definition dieser Begriffe nachgeholt sowie ihrer Begriffsbeziehungen aufgezeigt werden.

2.5.1 Sprachen und Metasprachen

Sprache im Allgemeinen bezeichnet die wichtigste Kommunikationsform überhaupt. Unter einer **(formalen) Sprache** wird in dieser Arbeit eine künstliche Sprache in Schriftform verstanden, die im Gegensatz zur natürlichen, also historisch entstandenen, Sprache vornehmlich die Darstellungsfunktion erfüllt. Andere von Bühler aufgeführte Funktionen einer Sprache, wie Appell oder Auslösen von Verhaltenstreue sowie Kundgabe oder Ausdruck von Gefühlen werden in künstlichen Sprachen vernachlässigt (vgl. [Büh34]). Innerhalb der schriftlichen, künstlichen Sprachen kann zwischen textuellen, deren (atomare) Notationselemente textuelle Zeichen oder Symbole sind, und graphischen Sprachen unterschieden werden. Die (atomaren) Notationselemente von graphischen Sprachen sind geometrische und topologische Figuren oder Symbole, wie Linien, Pfeile, geschlossene Kurven und Boxen u. ä. sowie Kompositionsmechanismen, wie Verbundenheit, Partitionierung, Durchschnitt oder Enthaltensein [HR04]. Bestimmte Sprachen, wie z. B. Modellierungssprachen, können sowohl textuelle als auch graphische Notationselemente enthalten. Die (endliche) Menge aller (atomaren) Notationselemente wird als **Alphabet** bezeichnet. Die Menge der nach bestimmten Regeln aus diesen Notationselementen eines Alphabets zusammengesetzten Sprachkonstrukte ergibt eine konkrete formale Sprache.

Die erlaubten Notationselemente sowie Regeln für mögliche Zusammensetzungen werden durch die Syntax der Sprache bestimmt. Zum einfacheren Verständnis wird oftmals neben der konkreten Syntax einer Sprache, die die konkreten, zur Verfügung stehenden Notationselemente zur Darstellung von Daten definiert, eine abstrakte Syntax beschrieben, die die zugrundeliegende Struktur der Notationselemente durch Konzepte, Beziehungen und Integritätsbedingungen festlegt [KSLB03]. Um die in den Notationselementen abgelegten Daten interpretieren zu können, muss zusätzlich eine Semantik der Sprache spezifiziert werden. Ansonsten können zwei gleiche Datensätze für verschiedene Leute oder Applikationen eine andere Bedeutung haben und deshalb für sie unterschiedliche Informationen darstellen. Ebenso ist es grundsätzlich möglich das zwei unterschiedliche Daten eine gleiche Information verkörpern können (in Anlehnung an [HR04]). Entsprechend erfolgt eine Zuordnung der konkreten Sprachkonstrukte zu Konstrukten der abstrakten Syntax sowie eine Zuordnung dieser syntaktischen Konstrukte auf den Gegenstandsbereich, auch als semantische Domäne bezeichnet. Fehlt die Definition einer abstrakten Syntax, muss jedem einzelnen konkreten Sprachkonstrukt eine Bedeutung zugeordnet werden. Wie in Abbildung 2.2 am Beispiel der Sprache XML exemplarisch gezeigt wird, kann zwischen

den einzelnen Sprachbestandteilen – konkrete Syntax, abstrakte Syntax und Semantik – jeweils eine eigene Zuordnung erfolgen. Allerdings muss bei einer solchen „Überspezifikation“ die Konsistenz zwischen den einzelnen syntaktischen sowie semantischen Abbildungen sichergestellt werden.

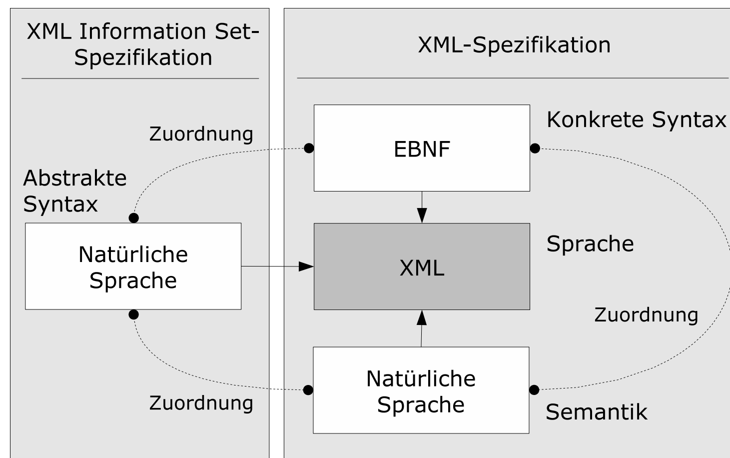


Abbildung 2.2: Sprachbestandteile von XML.

Für die Definition der Sprachbestandteile sind jeweils adäquate Repräsentationsformen erforderlich. Die konkrete Syntax der Sprachen kann bspw. mit grammatikbasierten Ansätzen (z. B. EBNF [Int96b], Graph-Grammatiken [Ehr79]) und die abstrakte Syntax durch modellbasierte Ansätze (z. B. Metamodelle wie Entity-Relationship-Diagramme oder Klassenmodelle, Datenschemata) definiert werden. Die Wahl der Repräsentationsform zur Definition der semantischen Bestandteile hängt stark vom Anwendungsbereich der Sprache ab. Die Möglichkeiten reichen von natürlichsprachlichen Beschreibungen für intuitiv verständliche Aspekte bis hin zur exakten Spezifikation durch Logiken (z. B. algebraische Spezifikationssprachen oder andere mathematische Ansätze [HR04]). Innerhalb der XML-Spezifikation [BPS+06a, BPS+06b] wird bspw. eine vereinfachte EBNF (erweiterte Backus-Naur-Form) zur Beschreibung der konkreten XML-Syntax und eine natürliche Sprache für die Beschreibung der Semantik der einzelnen XML-Notationselemente verwendet. Die abstrakte XML-Syntax wird ebenso in natürlicher Sprache innerhalb der Spezifikation des XML Information Sets [CT04] beschrieben (siehe Abbildung 2.2).

Mit Sprachen können Aussagen über einen zu untersuchenden Gegenstandsbereich (z. B. Modelle, Dokumente, Daten, aber auch Transformationen) formuliert werden. Werden Aussagen nicht nur über außersprachliche Gebilde, sondern auch über sprachliche Gebilde selbst gemacht, ist es zweckmäßig, in verschiedene Sprachebenen, auch semantische Stufen genannt, zu unterscheiden. Die Sprache, die selbst der Gegenstand der Untersuchung ist, wird als Objektsprache und die Sprache, in der die Untersuchung erfolgt, als Metasprache bezeichnet. Die Unterscheidung zwischen Objektsprache und Metasprache dient nicht der Klassifikation von Sprachen, sondern lediglich der Bildung von Bezugsebenen. Die Präfixe „Objekt“ (lat. *objectum* = Ziel, Gegenstand [Dud07]) und „Meta“ (griech. $\mu\epsilon\tau\alpha$ = zwischen, mit, um, nach [Dud07]) helfen dabei, die Sprachen den Ebenen zuzuordnen. Die Eigenschaft, eine Metasprache zu sein, ist in

diesem Sinne keine absolute Eigenschaft dieser Sprache, sondern vielmehr eine relative bzgl. einer anderen Sprache. Sie beschreibt die Rolle einer Sprache im Rahmen einer Untersuchung. Folglich ist das Prinzip des Bildens einer Metasprache zu einer Objektsprache rekursiv anwendbar, denn eine Sprache, die im Rahmen einer Untersuchung die Rolle einer Metasprache bzgl. einer Objektsprache einnimmt, kann wiederum zum Gegenstand der Untersuchung werden und somit ebenso die Rolle einer Objektsprache einnehmen. Bei der in dieser Untersuchung verwendeten Sprache handelt es sich um die Metasprache einer Metasprache oder um die Metametasprache bzgl. der ursprünglichen Objektsprache. Grundsätzlich ist dieses Prinzip fortführbar, so dass eine ganze Hierarchie von Sprachebenen gebildet werden kann. Um die einzelnen Sprachen auf den unterschiedlichen Sprachenebenen unterscheiden zu können, spricht man im Allgemeinen von Metasprachen i-ter Stufe bzw. in Kurzform M^iS. Entsprechend dieser Konvention steht S bzw. M^0S für die Objektsprache, MS bzw. M^1S für die Metasprache und MMS bzw. M^2S für die Metametasprache.

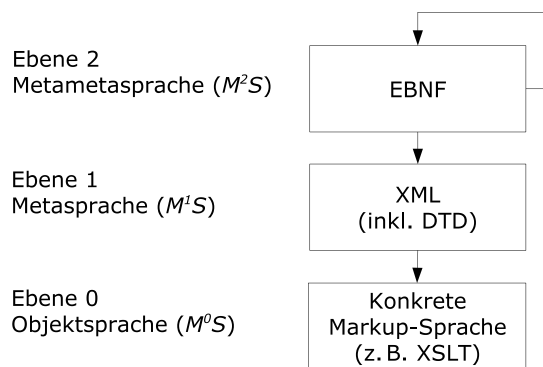


ABBILDUNG 2.3: Hierarchie von Sprachebenen im Kontext von XML.

Eine solche Sprachhierarchie entsteht bei künstlichen Sprachen erstaunlich häufig, da bereits die Definition einer künstlichen Sprache in einer Metasprache erfolgen muss. Beispielsweise wird zur Beschreibung der XML-Syntax eine formale Grammatik, eine vereinfachte EBNF, verwendet. Die XML-Spezifikation definiert selbst eine Sprache, die Dokumenttyp-Definition (DTD), mit der man die Syntax konkreter, individueller Markup-Sprachen beschreiben kann. Aus diesem Grund wird XML auch als Meta-Markup-Sprache bezeichnet ([Hol00] oder [HM04]). Die Sprachhierarchie in Abbildung 2.3 betrachtet ausschließlich die jeweiligen syntaktischen Sprachbestandteile. Die Semantik der Notationselemente in XML sowie der exemplarischen konkreten Markup-Sprache XSLT (*Extensible Stylesheet Language Transformation*) sind in natürliche Sprache beschrieben.

2.5.2 Modelle und Metamodelle

In unseren alltäglichen Leben benutzen wir Modelle bspw. wenn wir über Sachen oder Probleme nachdenken, wenn wir über sie kommunizieren oder wenn wir versuchen für sie einen Lösungsansatz zu konstruieren [Lud03]. Schon eine einfache sprachliche Beschreibung eines betrachteten

Gegenstandsbereiches kann als Abbildung dieses Bereiches auf ein System von Sprachkonstrukten verstanden werden. Sind Abbild und Gegenstandsbereich strukturgleich oder strukturerhaltend, so spricht man von einer isomorphen bzw. homomorphen Abbildung, die auch als Modell bezeichnet wird.

Im Allgemeinen ist ein **Modell** (abgeleitet vom lat. *modulus*) ein Abbild oder Vorbild eines Gegenstandes oder Gebildes [Bro00], wobei der Gegenstand oder das Gebilde auch als Original bezeichnet wird (vgl. Abbildung 2.4). Neben diesem Abbildungsmerkmal werden in [Sta77] noch zwei weitere Merkmale genannt: das Verkürzungsmerkmal und das pragmatische Merkmal. Das Verkürzungsmerkmal betont, dass Modelle nicht das Original in allen Details abbilden, sondern von bestimmten Details abstrahieren, die nicht in das Modell einfließen, um somit wesentliche Eigenschaften hervorzuheben und dem Modellbenutzer eine bessere Handhabbarkeit gegenüber dem Original zu bieten. Das pragmatische Merkmal verdeutlicht, dass Modelle den Originalen nicht eindeutig zugeordnet sind, sondern eine Ersetzungsfunktion für bestimmte Modellbetrachter, in einer bestimmten Zeit und unter Einschränkung auf bestimmte gedankliche oder tatsächliche Operationen erfüllen.

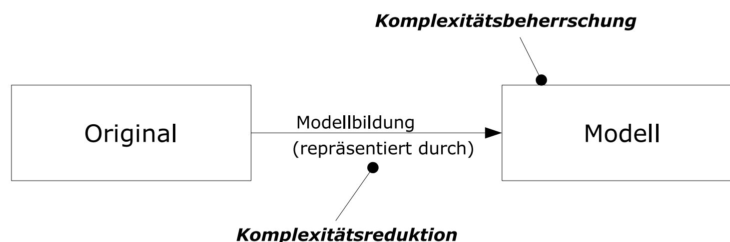


ABBILDUNG 2.4: Elemente der Modellbildung.

Werden Modelle im Alltag eher unbewusst eingesetzt, so ist diese Situation in der Forschung und Entwicklung insbesondere in der Informatik anders. In diesem Anwendungsgebiet werden Modelle explizit zur Komplexitätsreduktion sowohl von bestehenden als auch zu entwickelnden Systemen, den Originalen, erstellt.

Unabhängig davon können Modelle, trotz der Abstraktion von nichtrelevanten Eigenschaften des Originals, komplex werden. Viele Modellierungsansätze bieten deshalb Möglichkeiten zur Komplexitätsbeherrschung an, ohne jedoch auf die Abbildung weiterer Eigenschaften zu verzichten und dadurch die Komplexität weiter zu reduzieren. Beispiele dafür sind

- die Nutzung von Architekturen zur Strukturierung eines Gesamtmodells in mehrere Modellebenen sowie die Verknüpfungen dieser Ebenen durch definierte Beziehungen und Metamodelle, wodurch das Zusammenwirken der Elemente des Gesamtmodells gesichert wird,
- die Verfügbarkeit von Modellsichten, so dass der Modellnutzer das Gesamtmodell unter verschiedenen Blickwinkeln und Szenarien sieht,
- die hierarchische Zerlegung von Modellen und ihren Modellelementen in Teilmodelle, um Modelle auf unterschiedlichen Abstraktionsebenen analysieren und optimieren zu können,

- die Verwendung von Metaphern, Analogien und Mustern [Lud03].

Welche Eigenschaften des Originals wie in einem Modell abgebildet werden, ist immer von der Zielgruppe, von einer bestimmten Zeitspanne und dem Modellierungszweck abhängig [Sta77], vgl. auch [Ste93]. Aus diesem Grund kann ein Original mit einer Vielzahl von verschiedenen Sprachen, die von sehr formalen bis hin zu eher informalen Beschreibungsformen reichen, modelliert werden. Formale Modelle werden bspw. mit Gleichungssystemen und informale durch Bilder oder unstrukturierten Text beschrieben.

Wird die abstrakte Syntax einer Sprache, also deren Notationselemente sowie Beziehungen zwischen den Notationselementen, wiederum in einem Modell abgebildet, so spricht man von einem **Metamodell**. Dieses Metamodell ist in dem Sinne ein Modell eines Modells, als dass es ein Modell der dort zur Modellierung eingesetzten Sprache ist. Es wird daher auch als linguistisches Metamodell bezeichnet [AK03]. Das Metamodell wird wiederum in einer Sprache definiert. Da dies die Sprache ist, mittels der die ursprüngliche Objektsprache in Form eines Modells beschrieben ist, handelt es sich um eine Metasprache bzgl. der Objektsprache. Wird diese Sprache wiederum in ein Modell abgebildet, so handelt es sich dabei um das Metamodell eines Metamodells oder um das Metametamodell bzgl. des ursprünglichen Modells. Dieses Prinzip kann weiter fortgesetzt werden, so dass neben der Sprachhierarchie eine ganze Hierarchie von Modellebenen gebildet werden kann [Str98].

Neben dem Zweck der syntaktischen Sprachdefinition bei linguistischen Metamodellen wird Metamodellierung ebenso eingesetzt, um Konzepte und deren Eigenschaften in einem bestimmten Kontext oder einer bestimmten Domäne durch Typisierung zu beschreiben. Diese Modelle von Modellen werden ontologische Metamodelle genannt [AK03]. Weitere Metaisierungsprinzipien wie bspw. der prozessbasierte Metamodellbegriff sind möglich [Str98, Jec00]. Grundsätzlich ist jedoch nicht jedes Modell eines Modells zugleich ein Metamodell. Beispielsweise stellt eine Karte von einer Stadt im Maßstab 1:25 000 ein Modell dar. Von diesem Modell kann erneut eine Karte mit dem Maßstab 1:50 000 erstellt werden. Das ursprüngliche Modell stellt keine Instanz des zweiten Modells dar, d. h. das zweite Modell repräsentiert das ursprüngliche Modell sowie die originale Stadt und sollte deshalb nicht als Metamodell bezeichnet werden (in Anlehnung an [Béz05] oder [Küh05]). Gleiches gilt für die Sprachhierarchie von Programmiersprachen oder für die Operatorhierarchie (siehe Kapitel 5).

Im Kontext der Sprache XML kann ausgehend von der Sprachhierarchie der Abbildung 2.3 eine linguistische Modellhierarchie wie in Abbildung 2.5 aufgebaut werden. Die konkrete EBNF-Grammatik für XML (xml.ebnf) kann dabei als Modell angesehen werden, da diese zwar die konkrete Syntax von XML wiedergibt, aber von jeglicher Semantik abstrahiert. In gleicher Weise ist ein konkretes Schema in DTD, z. B. xslt .dtd, ein Modell einer Markup-Sprache, respektive der Transformationssprache XSLT. Das Modell xslt .dtd muss dabei Instanz vom Metamodell xml.ebnf sein. Da XML bzw. das Modell xml.ebnf neben der Möglichkeit ein Schema in DTD ebenso eine konkrete XML-Dokument-Instanz zu erstellen erlaubt, ist diese linguistische Modellhierarchie keine strikte Hierarchie, in der jedes Metamodell immer die Modellelemente eines Modells beschreibt, welches genau eine Ebene tiefer innerhalb der Hierarchie einzuordnen ist. Ein Dokument ist dann und nur dann ein XML-Dokument, wenn es eine Instanz des Modells

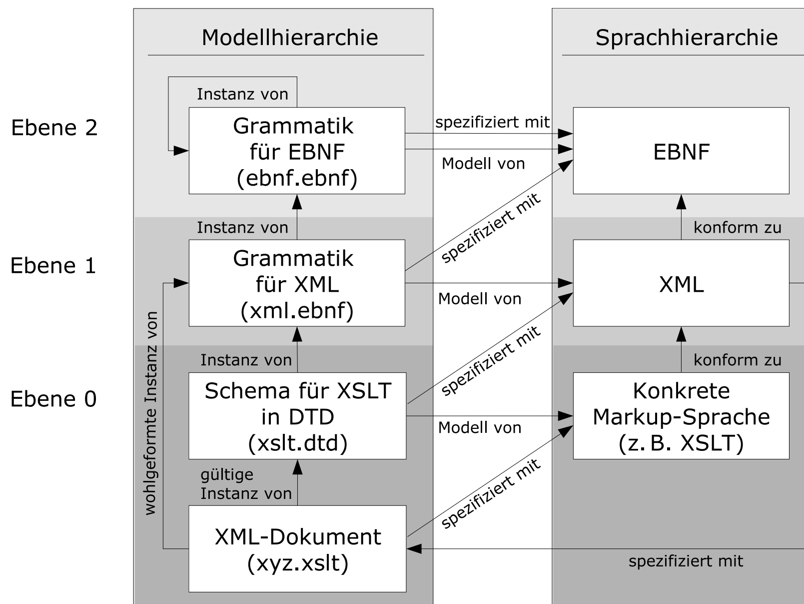


ABBILDUNG 2.5: Linguistische Modellhierarchie im Kontext von XML basierend auf der XML-Spezifikation.

xml.ebnf verkörpert und dadurch die Eigenschaft der Wohlgeformtheit erfüllt. Ist ein XML-Dokument darüber hinaus noch eine Instanz des Modells xslt.dtd, so stellt dieses eine gültige XSLT-Transformationsdefinition dar.

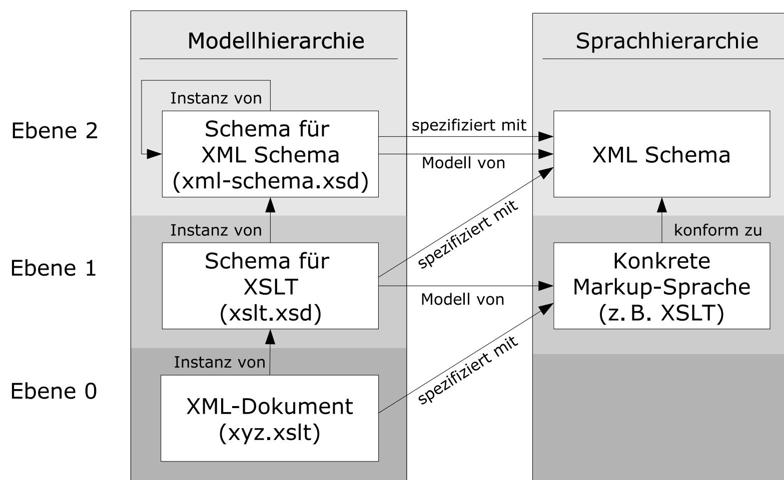


ABBILDUNG 2.6: Modellhierarchie im Kontext von XML basierend auf XML Schema.

Die Bezugsebenen der Modellhierarchie sind keine festen Eigenschaften der Modelle, sondern vielmehr relativ zu anderen Modellen. Beispielsweise handelt es sich bei XML-Dokumenten wie XSLT-Transformationsdefinitionen wiederum um Modelle bzgl. einer konkreten Transformationsfunktion [FN05], welche eine erneute Modellebene nach sich ziehen würden. Selbst mit XML-Dokumenten kann, wie die Abbildung 2.6 zeigt, eine eigenständige Modellhierarchie aufge-

baut werden. So modelliert ein konkretes XML Schema die abstrakte Syntax einer individuellen Markup-Sprache, z. B. wieder die Transformationssprache XSLT[1]. Das XML Schema selbst kann durch ein eigenes XML Schema-Modell beschrieben werden und schließt damit die Modellhierarchie nach oben ab.

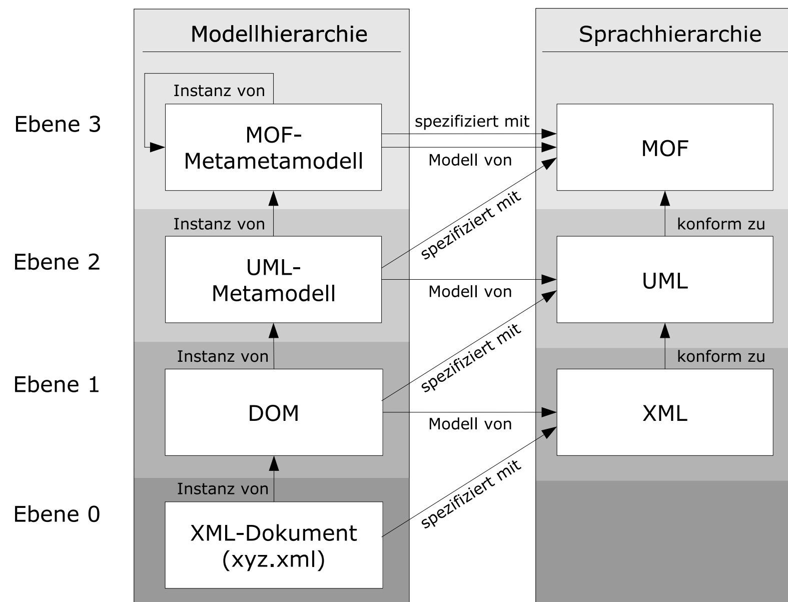


Abbildung 2.7: Modellhierarchie im Kontext von XML basierend auf MOF.

Die bisher vorgestellten Modelle von XML und XML-Dokumenten waren ausschließlich in textueller Sprache verfasst. Es ist aber ebenso möglich, die abstrakte XML-Syntax in graphischer Form bspw. mit der von der OMG (*Object Management Group*) vorgeschlagenen Metasprache MOF (*Meta Object Facility*) [OMG06] zu modellieren. Das XML-Metamodell in Abbildung 2.7 definiert mit Hilfe von MOF-Modellelementen die Modellelemente und Beziehungsarten zwischen Modellelementen, die für XML-Modelle auf einer Modellebene tiefer erlaubt sind. Eine solche Definition kann sich bspw. an dem *XML Information Set* [CT04] orientieren, welches die abstrakte XML-Syntax in textueller Form spezifiziert.

2.6 Einordnung der Arbeit in den XML-Technologieraum

Diese Arbeit beschäftigt sich mit den Transformationskomponenten innerhalb der XML-Infrastruktur. Sie konzentriert sich vor allem auf die Weiterentwicklung bestehender Technologien bzw. die Neuentwicklung basierend auf bestehenden Technologien.

Transformationskomponenten sind Programme, mit denen ein Input in einen modifizierten Output überführt wird. Das oder die Dokumente, die transformiert werden, müssen in Form einer

[1]Dies zeigt implizit, dass ein Original, hier die Markup-Sprache, auf viele unterschiedliche Weisen modelliert werden kann, hier durch DTD oder XML Schema.

bestimmten formalen Sprache vorliegen, die die Struktur und die Semantik der Dokumente beschreibt. Die Struktur erlaubt es, den Input maschinell zu transformieren. Die Semantik gibt die Möglichkeit, einerseits den Input mit dem Output zu vergleichen und andererseits die Transformation selbst zu überprüfen. Die Transformationskomponenten können zur Definition der Transformationen ganz unterschiedliche Transformationsansätze und -sprachen verwenden. Im kommenden Kapitel 3 werden einige von ihnen vorgestellt und anschließend eine genauere Einordnung bzgl. der Vielzahl an Transformationsansätzen vorgenommen.

Kapitel 3

Verarbeitung von XML-Dokumenten

XML besitzt aufgrund seiner Flexibilität bzgl. des festgelegten Strukturierungsgrades ein enormes Spektrum von Einsatzmöglichkeiten. Die zusätzliche Unabhängigkeit von einem bestimmten Betriebssystem, einer konkreten Programmiersprache oder sonstigen Plattformen und Herstellern macht XML-Dokumente insbesondere als Austauschformat für Daten jeglicher Art interessant. Im Gegensatz zu speziellen, an bestimmte Software gebundenen Datenformaten stehen somit die Daten der gesamten Öffentlichkeit zur Verfügung.

Die alleinige Verständigung auf XML als Basis für den Datenaustausch genügt jedoch nicht, dass jede Applikation die in einem beliebigen XML-Vokabular ausgedrückten Daten richtig interpretieren kann. Damit der Sender oder der Empfänger nicht verschiedenste Vokabulare beherrschen muss, sind innerhalb einer solchen XML-basierten Infrastruktur Transformationskomponenten wichtige Bestandteile, die die Daten in das jeweils gewünschte XML-Vokabular übersetzen. Diese Komponenten müssen sich leicht erstellen und anpassen lassen.

Neben der Transformation zum Zwecke des Datenaustausches zählt die Überführung von in XML gespeicherten Daten in eine formatierte Darstellung als zweites Hauptanwendungsgebiet. Das ursprüngliche generische Markup wird bei dieser Transformation durch ein prozedurales Markup ersetzt. Mögliche Darstellungsformen sind dabei bspw. Web-Dokumente in XHTML, Seitenbeschreibungen in XSLFO, Vektorgrafiken in SVG oder multimediale Präsentationen in SMIL. Das Zielformat muss hierbei nicht wie in den zuvor genannten Anwendungsbeispielen zwingend auf XML basieren. Andere Formate wie z. B. PDF, LaTeX oder RTF als Beispiele für Seitenbeschreibungssprachen sind ebenso möglich.

Um eine Transformation durchführen zu können, wird der XML-Text in eine interne Zwischenrepräsentation überführt. Diese reicht von einfachem Text bis hin zu komplexen Baumstrukturen, die lediglich XML-Daten enthalten. Abhängig von der Form dieser Zwischenrepräsentation, müssen entsprechende Analysephasen der eigentlichen Transformationsphase vorgeschaltet werden. Bei der Transformation auf Textebene kann auf die Analysephase gänzlich verzichtet werden. In der eigentlichen Transformationsphase wird aus diesen XML-Daten anhand der Transformationsdefinition eine andere interne Zwischenrepräsentation (automatisiert) konstruiert[1]. Das

[1] Wenn von nun an von einem XML-Text und XML-Daten die Rede sein wird, so bezieht sich dieser Begriff explizit auf die konkrete bzw. abstrakte Syntax. Der Begriff XML-Dokument wird im Folgenden als Oberbegriff verwendet, der sich nicht auf eine spezielle Repräsentation bezieht.

Ergebnis der Transformationsphase wird in der Serialisierungsphase in ein physisches Darstellungsformat überführt. Die Serialisierungsphase ist optional. Einige Transformationsmethoden, z. B. die Template-basierte Transformationsmethode (siehe Abschnitt 3.2.2.9), benötigen keine Serialisierungsphase, da sie direkt die konkrete Syntax der Zielsprachen erzeugen.

Die Beschreibung einer Transformation kann auf unterschiedliche Weise durchgeführt werden. Sie kann in einer universellen Programmiersprache erfolgen, die entsprechend für das Verarbeiten von XML-Dokumenten erweitert oder angepasst wird (intern). Sie kann aber auch in einer spezifischen XML-Sprache, die mglw. selbst auf einer universellen Programmiersprache basiert, durchgeführt werden (extern). Sprachen, die für nur eine bestimmte, abgegrenzte Menge von Aufgaben definiert werden, werden in der Literatur oftmals als domänenspezifische Sprachen (kurz DSL, *Domain Specific Language*) bezeichnet. Ebenso sind Synonyme wie anwendungsspezifische Sprachen [Cle88], Very-High-Level-Sprachen oder Mikrosprachen [Ben86] zu finden. Beide Implementierungsansätze – interne vs. externe DSL – haben ihr Für und Wider.

In den folgenden Abschnitten werden mögliche interne Repräsentationsformen erläutert und die Aufgaben der einzelnen Verarbeitungsphasen, die durchlaufen werden, um ein XML-Dokument zu transformieren, vorgestellt. Auf die Transformationsphase wird dabei im Abschnitt 3.2.2 detailliert eingegangen, wobei zunächst keine getrennte Untersuchung für Ansätze mit universellen Sprachen und Ansätze mit eigenständigen, spezifischen Sprachen vorgenommen wird. Vielmehr wird eine Domänenanalyse durchgeführt, bei der die Transformationskonzepte im Vordergrund stehen und deshalb die beide Implementierungsansätze gleichzeitig betrachtet werden. Anschließend werden Kategorien gebildet, denen die unterschiedlichen Sprachen und Konzepten zugeordnet werden. Auf die Aufstellung von Bewertungskriterien und eine Evaluation jeder einzelnen Sprache wird verzichtet (siehe dazu bspw. die Arbeiten [Bec04] oder [Hau06]). Abschließend wird in diesem Kapitel auf die Vor- und Nachteile der einzelnen Implementierungsansätze eingegangen und die weitere Arbeit in das Gebiet der XML-Verarbeitung eingeordnet.

3.1 Repräsentationsformen

Die Art der internen Repräsentation eines XML-Dokumentes hat entscheidenden Einfluss auf die einzelnen Verarbeitungsphasen. Aus diesem Grund wird das interne Verarbeitungsmodell als Erstes betrachtet. Es kann zwischen zwei wesentlichen Repräsentationsformen unterschieden werden:

- **Repräsentation auf lexikalischer Ebene**
  Wird auf die lexikalische und syntaktische Analyse gänzlich verzichtet, kann ein XML-Dokument lediglich als Text intern repräsentiert werden. Es wird somit nicht zwischen Markup und den eigentlichen Zeichendaten unterschieden. Alle lexikalischen Eigenschaften sind sichtbar und können für die Transformation genutzt werden. Es können allerdings nur einfache Textersetzungsmechanismen zur Umsetzung der Transformation eingesetzt werden.

- **Repräsentation durch ein Verarbeitungsmodell**
  Alternativ kann von der konkreten Syntax eines XML-Dokumentes abstrahiert werden. Der XML-Text wird mit Hilfe eines Parsers in der Analysephase auf spezifische Datenstrukturen abgebildet, die mehr oder weniger den Informationseinheiten des XML Infosets entsprechen. Zum Zugriff und zur Verarbeitung der XML-Daten werden entsprechende Schnittstellen bereitgestellt. In Abhängigkeit von der während der Analysephase aufgebauten Datenstruktur und deren zusätzlichen Zugriffsfunktionen lassen sich unterschiedliche Verarbeitungsmodelle identifizieren.

Abbildung 3.1 zeigt einige Repräsentationsformen und deren Ausprägungen. Sie werden in den folgenden Unterabschnitten näher vorgestellt.

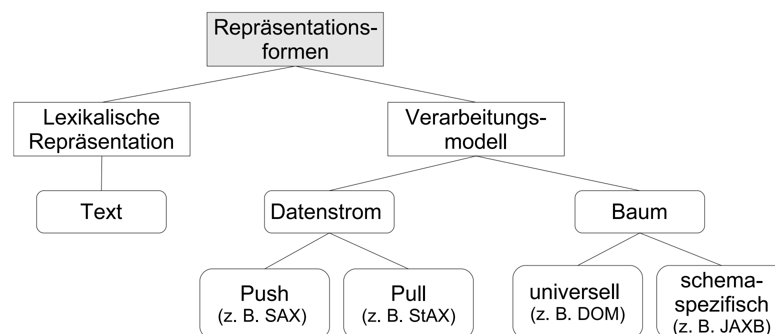


Abbildung 3.1: Repräsentationsformen von XML.

3.1.1 Text

XML-Dokumente bestehen aus reinem, mit Markups ausgezeichnetem Text. Es können somit allgemeine Textersetzungsmechanismen für die Verarbeitung der XML-Dokumente angewendet werden, die eine einfache Zeichenmanipulation erlauben. Zur Nutzung dieser Repräsentationsform von XML-Dokumenten ist keine vorgelagerte lexikalische und syntaktische Analyse notwendig. Jedoch erfordert die Unterscheidung zwischen Markup und eigentlichen Zeichendaten einen zusätzlichen Aufwand für die Beschreibung der gewünschten Transformation. Würde dies nicht beachtet, könnten womöglich unbeabsichtigt Markups verändert werden, obwohl nur bestimmte Zeichendaten ersetzt werden sollten oder umgekehrt.

Textersetzungsmechanismen werden immer dann benötigt, wenn die genaue Zeichenrepräsentation eine Rolle spielt, weil z. B. die syntaktische Textform eines XML-Dokumentes gewahrt werden soll. Dies ist bspw. eine Anforderung bei der Transformation von digitalen Signaturen, da Signaturen eine eindeutige textuelle Darstellung festlegen. Andere Verarbeitungsmodelle sind in diesem Anwendungsfall weniger gut geeignet, da sie von der konkreten XML-Syntax abstrahieren. Dadurch gehen die syntaktischen Informationen verloren und sind entsprechend nicht mehr reproduzierbar.

Ebenso werden bei der Überführung vom Text in eine dem XML Infoset entsprechende Repräsentation Entity- und Zeichenreferenzen durch deren referenzierten Inhalt ersetzt. Die Referenz ist nicht mehr sichtbar. Das heißt, auch in diesem Fall wird von Informationen abstrahiert, die später nicht mehr rekonstruierbar sind. Insbesondere bei den Entity-Referenzen erweist sich dies als nachteilig. So lässt sich bspw. ein wiederkehrender Text innerhalb eines internen Entity definieren, auf den mehrfach mit einer Entity-Referenz verwiesen wird. Bei einer Änderung muss der wiederkehrende Text nur an einer Stelle, gerade im Entity, angepasst werden. Werden diese Referenzen aufgelöst, besteht diese Möglichkeit nicht mehr. Bei der Verwendung von externen Entities, die es bspw. ermöglichen ein Dokument in mehrere Dateien aufzuteilen, wird der erhebliche Informationsverlust noch deutlicher.

Um sicher zu stellen, dass bei der Überführung in ein Verarbeitungsmodell solche Referenzen erhalten bleiben, können sie mit Hilfe von Textersetzungsmechanismen maskiert werden. Dadurch werden sie vom Parser nicht mehr als Referenz erkannt und entsprechend nicht aufgelöst. Dies kann bereits erreicht werden, indem das et-Zeichen (&) durch eine spezielle Zeichenfolge (z. B. !Referenz!) ersetzt wird, die als Platzhalter dient und im Dokument sonst nicht auftreten darf. Nach der Verarbeitung, bei der vorausgesetzt wird, dass sie den Platzhalter nicht ändert, wird dieser wieder auf das ursprüngliche et-Zeichen zurückgesetzt. Das folgende Beispiel zeigt exemplarisch eine mögliche Realisierung innerhalb einer UNIX-Shell.

```
sed 's/&/!Referenz!/g' §XML-Quelldatei§ | §XML-Transformation§ | sed 's/!Referenz!/\&/g'
```

Dabei werden genau diese drei Schritte in einem Verarbeitungsprozess miteinander verknüpft. Die jeweilige Ersetzungen von & zu !Referenz! und wieder zurück übernimmt das kleine Programm sed. *XML−Transformation* steht im oberen Beispiel für ein beliebiges, fiktives Programm, welches einen XML-Text von der Standardeingabe liest und das transformierte Ergebnis auf die Standardeingabe schreibt.

Bewertung

Wie die oberen Anwendungsfälle veranschaulichen, wird die Verarbeitung durch Textersetzungsmechanismen insbesondere dann benötigt, wenn die Manipulation von Zeichen im Vordergrund steht. Da dafür keine interne Repräsentation aufgebaut werden muss, kann eine Verarbeitung auf lexikalischer Ebene für beliebig große XML-Texte vorgenommen werden. Muss bei der Transformation jedoch zwischen Markup und Zeichendaten unterschieden werden, so erfordert diese Herangehensweise einen unverhältnismäßig hohen Programmieraufwand. Darüber hinaus ergibt sich ein zusätzlicher Aufwand bei der Sicherstellung, dass die definierten Zeichenersetzungen ein syntaktisch korrektes XML-Dokument als Ergebnis erzeugen. Der Anwender bekommt diesbezüglich keinerlei Unterstützung durch Fehlerkennungsmechanismen. Nichtsdestotrotz stellt diese Herangehensweise zu den übrigen, die auf einer eigenen Repräsentationsform basieren, eine notwendige und sinnvolle Ergänzung dar.

3.1.2 Datenstrom

Bei einem datenstrombasierten Verarbeitungsmodell werden die XML-Bestandteile, Markups und Zeichendaten, in kontinuierlichen Abfolgen von Datensätzen, deren Ende nicht vorauszusehen ist, bereitgestellt. Dieser Datenstrom erlaubt nur einen sequentiellen Zugriff auf die einzelnen Datensätze und kann daher nur fortlaufend verarbeitet werden. In Abhängigkeit wie der Parser den Zugriff auf die Datensätze zur Verfügung stellt, kann in

- Push-Modell und
- Pull-Modell

unterschieden werden. Im Folgenden werden beide Modellarten kurz vorgestellt.

Push-Modell

Das Push-Modell basiert auf einer ereignisgetriebenen Verarbeitung. Der Parser definiert eine Reihe von Ereignissen, die entsprechend der Struktur des Quelldokumentes sequentiell ausgelöst werden. Durch das Überschreiben von vordefinierten Ereignisbehandlungsroutinen kann ein gewünschtes Verhalten für das jeweilige Ereignis festgelegt werden. Die Applikation arbeitet dabei immer passiv. Sie wartet auf die serielle Aktivierung der verschiedenen Ereignisse durch den Parser.

Der bekannteste Vertreter des Push-Modells ist SAX (*Simple API for XML*) [Meg04]. SAX wurde ursprünglich in Java geschrieben. Mittlerweile existieren aber verschiedene Versionen für andere Programmiersprachen. Seit der Version 1.4 der Java-Standardedition (J2SE) wurde SAX von SUN in das Paket JAXP (*Java API for XML Processing*) aufgenommen. Die Operationen zur Behandlung der Ereignisse in SAX orientieren sich an den Informationseinheiten des XML Infosets. Eine eineindeutige Zuordnung existiert allerdings nicht. So unterstützt SAX bspw. nicht den Zugriff auf die Kodierung eines XML-Dokumentes. Umgekehrt bietet SAX durch eine standardisierte Erweiterung die Möglichkeit Entity-Referenzen und CDATA-Abschnitte zu erkennen, die wiederum im Infoset nicht vorgesehen sind.

Pull-Modell

Einen alternativen Ansatz zur passiven Verarbeitung eines Datenstroms ermöglicht das Pull-Modell. Das Pull-Modell erlaubt der Applikation, im Gegensatz zum Push-Modell, in die Verarbeitungsreihenfolge einzugreifen. Die Applikation kann aktiv Informationen aus einem XML-Dokument extrahieren, wenn diese benötigt werden. Der Zugriff erfolgt allerdings hierbei nicht wahlfrei, sondern fortlaufend entlang der Reihenfolge der einzelnen Datensätze im Datenstrom, die der Parser durch die syntaktische Analyse des Quelldokumentes zur Verfügung stellt. Dieses Vorgehen kann mit dem linearen Vorrücken eines Zeigers verglichen werden.

Ein Vertreter des zeitlich jüngeren Pull-Modells ist StAX (*Streaming API for XML*) in Java [Ben03]. StAX ist eine Weiterentwicklung des Vorgängers XPP (*XML Pull Parsing*) [Slo05] und wurde von SUN seit der Java Enterprise Edition 5 (Java EE 5.0) als Bestandteil aufgenommen.

Neben dem grundsätzlich anderen Ausführungsmodell sind die bereitgestellten Informationen von StAX und SAX vergleichbar. StAX bietet z. B. ebenso wie SAX die Möglichkeit, falls erwünscht, Entity-Referenzen und CDATA-Abschnitte nicht aufzulösen, so dass sie erkannt und für die Transformation verwendet werden können. Unterschiede zwischen beiden sind nur im Detail zu finden. So kann bspw. in StAX auf die Kodierung eines XML-Dokumentes zugegriffen werden. Wiederum Informationen über XML-Deklarationen innerhalb der DTD werden durch SAX besser unterstützt.

Bei beiden Ausführungsmodellen – aktiv und passiv – muss die Applikation eine eigene Struktur für die XML-Daten aufbauen. Somit liegt die Speicherung und Verarbeitung der Daten im Verantwortungsbereich der Applikation. Dies bedeutet zusätzlichen Aufwand. Andererseits kann die Applikation dynamisch in Abhängigkeit einer bestimmten Aufgabe entscheiden, welche Daten wie in einer Datenstruktur gespeichert und verarbeitet werden. So kann sie bspw. genau die Daten, die im aktuellen Transformationsschritt benötigt werden, im Speicher halten und nicht mehr benötigte Daten wieder freigeben. In manchen Sonderfällen, wenn keine oder nur wenige Kontextinformationen aus anderen Teilen des Quelldokumentes notwendig sind, muss sogar keine oder nur eine minimale Datenstruktur aufgebaut werden. Ein typischer solcher Anwendungsfall ist die einfache Umbenennung von benannten Markups.

Neben dem Aufbau einer eigenen Datenstruktur muss sich die Applikation bei dem datenstrombasierten Verarbeitungsmodell ebenfalls um die Serialisierung kümmern. SAX bietet als der Vertreter des Push-Modells keine Funktionen zur Generierung von XML. Hier gibt StAX mehr Unterstützung. StAX stellt Ausgabeoperationen zur Verfügung, die bspw. Elemente, Attribute und Zeichendaten erzeugen und bei der Maskierung bestimmter Zeichen helfen. Die Korrektheit des Outputs bzgl. der XML-Spezifikation wird aber nur minimal unterstützt. So ist es bspw. möglich, mehrere Wurzelelemente oder Elementnamen mit Leerzeichen zu generieren, die gegen das Wohlgeformtheitskriterium verstoßen. Eine Lösung ist die Benutzung von ergänzenden APIs, die korrekte XML-Dokumente erzeugen und Verstöße gegen die Anforderungen der Wohlgeformtheit erkennen. In diesem Fall muss die Applikation das entsprechende Format der Serialisierungskomponente produzieren.

Bewertung

Beide Ausführungsmodelle der datenstrombasierten Verarbeitung unterstützen prinzipiell beliebig große XML-Dokumente, da keine zusammenhängende, allgemeine Repräsentation der Quelldokumente erzeugt wird. Vielmehr kann abhängig vom konkreten Anwendungsfall entschieden werden, was für eine Datenstruktur für die zu speichernden XML-Daten aufgebaut und verwaltet wird. Die Applikation bestimmt, wie umfangreich eine solche Datenstruktur ausfällt, wie auf sie zugegriffen wird und wie sie manipuliert wird. Dies bedeutet insgesamt einen generell höheren Programmieraufwand und kann bei nichttrivialen Transformationsaufgaben zu komplexem und unübersichtlichem Code führen. Darüber hinaus handelt es sich beim datenstrombasierten Verarbeitungsmodell zumeist um APIs, die nur das Parsen der Quelldokumente unterstützen und keine oder nur rudimentäre Hilfe für die Serialisierung von XML-Daten anbieten. STX

[CBN+07] ist eine spezifische XML-Transformationssprache, die auf einem datenstrombasiertes Verarbeitungsmodell arbeitet.

3.1.3 Baum

Bei einem baumbasierten Verarbeitungsmodell wird während der Analysephase eine komplette Datenstruktur des Quelldokumentes aufgebaut. Diese Datenstruktur stellt dem Entwickler im Gegensatz zum Datenstrom eine Baumansicht des gesamten Quelldokumentes zur Verfügung und ermöglicht es, über bereitgestellte Schnittstellen wahlfrei lesend und schreibend auf alle enthaltenen Informationen zuzugreifen. Abhängig von der Universalität der erzeugten Baumstruktur lassen sich zwei baumbasierte Verarbeitungsmodelle unterschieden:

- Aufbau einer generischen Struktur, die im weitesten Sinne dem XML Infoset entspricht (universelles Modell), und
- Aufbau einer spezifischen Struktur, die durch die Analyse eines Schemas erzeugt werden kann (schemaspezifisches Modell).

Beide prinzipielle Herangehensweisen werden im Folgenden kurz erläutert.

Universelles Modell

Das universelle Modell orientiert sich strukturell stark an den Informationselementen des XML Infosets. Ähnlich wie beim XML Infoset wird eine generische Struktur erzeugt, die Knoten für jeden Typ von Markup sowie für die eigentlichen Zeichendaten bereithält. Die Struktur ist damit unabhängig von einer bestimmten Klasse von XML-Dokumenten bzw. spezieller Markup-Sprachen. Neben der generischen Struktur werden ebenso Schnittstellen definiert, mit denen Änderungen an dem bereitgestellten Baum vorgenommen werden können. Es können neue Knoten hinzugefügt, andere entfernt oder Eigenschaften der Knoten geändert werden. Unterscheidet sich das gewünschte Ergebnis einer Transformation strukturell stark vom Quelldokument, so kann es günstiger sein, einen komplett neuen Baum aufzubauen. Die anschließende Serialisierung von XML-Text aus einer solchen Datenstruktur übernehmen i. d. R. entsprechende Schnittstellenfunktionen.

Der bekannteste Vertreter des baumbasierten, universellen Modells ist das DOM (*Document Object Model*) [HHW+04]. DOM definiert eine abstrakte, programmiersprachenunabhängig formulierte Schnittstelle zum lesenden und schreibenden Zugriff auf wohlgeformte XML-Dokumente sowie eine Reihe weiterer Formate (z. B. HTML). Die Schnittstellen werden in IDL (*Interface Definition Language*) der OMG spezifiziert. Das durch die Schnittstellen beschriebene Objektmodell entspricht stark dem des XML Infosets. Einige Unterschiede gibt es jedoch. So sieht DOM u. a. eigene Knotentypen für CDATA-Abschnitte sowie für zusammenhängende Zeichendaten vor. DOM ist das Standard-API für den Zugriff und die Manipulation von XML-Dokumenten und wurde auf etliche Programmiersprachen abgebildet. Da DOM allerdings keine sprachspezifischen Konstrukte der jeweiligen Programmiersprache berücksichtigt und daher an manchen

Stellen schwerfällig und ineffizient wirkt, wurden inzwischen einige Alternativen zu DOM entwickelt. So wurden bspw. für Java die sprachspezifischen APIs JDOM [HM07] und DOM4J [Str05] entworfen. Beide erlauben eine effizientere Programmierung durch Java-typische Klassen. Dessen ungeachtet hat DOM als W3C-Standard eine große Verbreitung in der Praxis gefunden.

Schemaspezifisches Modell

Einen alternativen Ansatz stellen spezifische Modelle dar, die von einem konkreten Schema abgeleitet werden. Während beim universellen Modell entsprechend den Informationselementen des XML Infosets ein generischer Baum modelliert wird, ermöglichen sogenannte *Data Binding*-Frameworks (wie z. B. JAXB [VF06] oder Castor [Cas08]) die Erzeugung von vokabularspezifischen Speicher- und Verarbeitungsstrukturen. Dazu werden aus einer in DTD oder XML-Schema vorliegenden Grammatik programmiersprachliche Konstrukte, etwa Klassendefinitionen und darauf operierende Methoden, automatisiert generiert, die wiederum in eine eigene Applikation eingebunden werden können. In einem solchen Baum existiert anstelle des generischen Knotentyps Element für jeden XML-Elementtyp ein eigener Knotentyp, der nur solche Kindknoten zulässt, die im Schema erlaubt werden.

Der Programmieraufwand gestaltet sich somit geringer, da zum einen notwendige Typkonversionen wie bspw. beim DOM wegfallen und zum anderen anwendungsspezifische Datenstrukturen einen leichteren und zudem typisierten Zugriff auf XML-Daten ermöglichen. Darüber hinaus wird durch die von einem Schema abgeleitete Datenstruktur erreicht, dass nur gültige XML-Dokumente erzeugt werden können. Als Kehrseite bedingt diese restriktive Struktur, dass lediglich Manipulationen der Zeichendaten auf diesem Baum erlaubt sind. Eine einfache Elementumbenennung stellt bereits eine strukturelle Transformation dar, die den Aufbau eines vollständigen neuen Ergebnisbaums erfordert. Soll somit ein Quelldokument in ein anderes XML-Vokabular überführt werden, so wird für den Output ebenso eine Grammatik benötigt. Des Weiteren ist zu beachten, dass auf alle XML-Markups, die nicht in einem Schema beschrieben werden, nicht zugegriffen werden kann. Es können bspw. deshalb weder Verarbeitungsanweisungen noch Kommentare in eine Transformation einbezogen werden. CDATA-Abschnitten und Entity-Referenzen können ebenfalls nicht erkannt und entsprechend verarbeitet werden.

Bewertung

Die Repräsentation des gesamten XML-Dokumentes als Baumstruktur ermöglicht einen wahlfreien lesenden und schreibenden Zugriff auf alle enthaltenen Informationen. Der Aufbau einer kompletten Datenstruktur eines Quelldokumentes bedingt aber auch, dass nur XML-Dokumente mit begrenzter Größe auf diese Weise verarbeitet werden können. Schemaspezifische Modelle nutzen gegenüber universellen Modellen diesbezüglich effizientere Speicher- und Verarbeitungsstrukturen, die zudem entsprechend eines Schemas nur angepasste Operationen zum einfacheren, typisierten Zugriff auf XML-Daten erlauben. Für eine pure inhaltliche, kontextabhängige Veränderung der XML-Daten eignet sich dieses Verarbeitungsmodell deshalb sehr gut. Müssen jedoch

kontextabhängige, strukturelle Transformationen vorgenommen werden oder muss auf Informationen zurückgegriffen werden, die nicht in einem Schema repräsentiert sind, sollten universelle Modelle bevorzugt werden. Sie sind aufgrund ihrer Universalität der Datenstruktur flexibler. Beide Ansätze bieten typischerweise Mechanismen zur Erzeugung von XML-Text aus der internen, baumbasierten Datenstruktur. Die schemabasierte Variante kann durch die Interpretation eines zugehörigen Schemas sogar darüber hinaus sicherstellen, dass gültige XML-Dokumente serialisiert werden. Die Mehrzahl der spezifischen XML-Transformationssprachen basiert auf einem universellen Verarbeitungsmodell.

3.2 Phasen

In den folgenden Abschnitten wird auf die einzelnen Phasen bei der Verarbeitung von XML-Dokumenten eingegangen.

3.2.1 Analyse

Die Analyse eines XML-Dokumentes erfolgt im Wesentlichen in zwei Schritten: die lexikalische und die syntaktische Analyse.

Das Ziel der lexikalischen Analyse ist es, den XML-Text in lexikalische Einheiten (Wörter) zu zerlegen. Diese können aufgehende oder schließende Klammern eines Tags oder Bezeichner von Elementen und Attributen sein. Die lexikalische Analyse ist die Basis für die syntaktische Analyse. Die syntaktische Analyse wird verwendet, um aus den einzelnen lexikalischen Einheiten vollständige Informationseinheiten in den XML-Dokumenten (z. B. Elemente, Attribute, Kommentare) zu erkennen. Neben der Syntaxanalyse eines XML-Textes gegen die XML-Spezifikation kann ebenso eine syntaktische Validierung gegen eine formale Beschreibung (z. B. DTD oder XML Schema) der Quellsprache in diesem Analyseschritt erfolgen. Man spricht in einem solchen Fall von validierenden Parsen.

3.2.1.1 Lexikalische Analyse

Die Hauptaufgabe der lexikalischen Analyse ist es, den Zeichenstrom eines Inputs in eine Folge logisch zusammengehöriger Einheiten, die sogenannten (lexikalischen) Token, zu zerlegen. Token sind (Typ-Wert-)Paare, bestehend aus einem Symbol und einem Lexeme. Im Allgemeinen wird für eine unterschiedliche Folge von Zeichen des Inputs das gleiche Symbol (z. B. `Name` für die Klasse der Bezeichner) als Output erzeugt. Die Menge solcher Zeichenfolge wird durch eine zum Symbol gehörende Regel beschrieben, durch ein sogenanntes Muster. Ein Lexeme ist eine Zeichenfolge im Quelldokument (z. B. der konkreter Name `quelle`), die dem Muster für ein Symbol entspricht. Dieses wird bei der lexikalischen Analyse dem entsprechenden Symbol zugeordnet.

Muster können typischerweise durch reguläre Ausdrücke beschrieben werden, z. B. `Name ::= NameStartChar (NameChar)*`. Aus diesem Grund wird die lexikalische Analyse i. d. R. als endlicher Automat realisiert. Durch die präzise Spezifikation mittels regulärer Ausdrücke oder gleichwertiger Methoden, können die endlichen Automaten durch generierende Werkzeuge (z. B. GNU Flex [Fre95]) automatisiert systematisch konstruiert und effizient implementiert werden. Ein Verfahren zur Überführung eines regulären Ausdrucks in einen endlichen Automaten ist das Berry-Sethi-Verfahren [BS86].

Eine andere Aufgabe der lexikalischen Analyse besteht darin, Fehlermeldungen innerhalb späterer Analyse- und Verarbeitungsschritte entsprechende Positionen im Quelldokument zuzuordnen [ASU86]. Ebenso können u. U. unwichtige Zeichen wie Einrückungen, Zeilenwechsel oder Leerzeichen übersprungen und damit ignoriert werden. Dies kann durch die zugrundeliegende Grammatik oder mit einem separaten Screener umgesetzt werden [Pen94]. Zusätzlich können in diesem Schritt Entity-Referenzen und Namensraumdeklarationen aufgelöst werden. Ignorierte und aufgelöste Zeichen sind für die weiteren Verarbeitungsphasen nicht mehr reproduzierbar. Dadurch vereinfacht sich aber die folgende syntaktische Analyse erheblich.

Das Ergebnis der lexikalischen Analyse bzgl. eines XML-Textes sind Folgen von Symbolen und deren zugeordneten Lexemen. Beispielsweise können innerhalb des XML-Textauschnittes

```
quelle="http://www.it-wissen.info"
```

die Token

```
Name 'quelle'
Equal
Quote
Value 'http://www.it-wissen.info'
Quote
```

identifiziert werden.

XML-Scanner, die die lexikalische Analyse von XML-Dokumenten implementieren, sind in i. d. R. in XML-Parser integriert.

3.2.1.2 Syntaktische Analyse

Die syntaktische Analyse hat das Ziel, aus den Folgen von Token der lexikalischen Analyse die syntaktische Struktur zu erfassen und in geeigneter Form für nachfolgende Phasen darzustellen. Grundlage hierfür ist die XML-Spezifikation, die festlegt, wie XML-Dokumente aufgebaut sein müssen. Die in der Spezifikation festgelegten Konventionen werden von den Parsern, die die Syntaxanalyse durchführen, zum einen geprüft und zum anderen für den Aufbau der Zieldarstellung genutzt. Die Prüfung umfasst sowohl einfache (z. B. ein Tag besitzt grundsätzlich immer eine öffnende und eine schließende eckige Klammer) als auch strukturelle Festlegungen (z. B. korrekte Schachtelung von Tags). Die während der Prüfung sukzessive erzeugte Zieldarstellung kann sich in Abhängigkeit des jeweiligen Parsers stark unterscheiden. Sie reicht von einer datenstrombasierten Repräsentation, in der die syntaktischen Bestandteile in einer kontinuierlichen Abfolge zur Verfügung gestellt werden, bis hin zur Baumdarstellung, in der eine komplette Datenstruktur aufgebaut wird. Unabhängig von der erzeugten Datenstruktur orientieren sich die

bereitgestellten Informationen der syntaktischen Bestandteile i. d. R. am XML Infoset [CT04]. Im Abschnitt 3.1.2 und 3.1.3 werden einige konkrete Repräsentationsformen vorgestellt.

Beschränkt sich die Analysephase lediglich auf diese erste Syntaxanalyse der XML-Spezifikation wird von nichtvalidierenden Parsern gesprochen (z. B. XP [Cla98], Crimson [Apa01]). Validierende Parser (z. B. Xerces [Apa05], Oracle XML Parser [Ora06]) prüfen darüber hinaus, ob ein XML-Dokument dem Schema der Quellsprache (z. B. in DTD oder XML Schema) genügt. Dadurch können weitere zusätzliche hierarchische Abhängigkeiten (z. B. die Benennung und Typisierung von Elementen sowie erlaubten Attributen) gesichert werden. Eine solche Prüfung kann sukzessive während des Aufbaus der Zieldarstellung oder in einem getrennten Analyseschritt nach dem Aufbau der Zieldarstellung erfolgen. Durch die Informationen, die im Schema der Quellsprache enthalten sind, kann die Zieldarstellung bspw. durch Typannotationen angereichert werden. Sie wird daher üblicherweise als PSVI (*Post Schema Validation Infoset*) bezeichnet. Unabhängig davon müssen Parser in sämtlichen Teilschritten der syntaktischen Analyse Fehler erkennen sowie eine Beschreibung erkannter Fehler in Hinblick auf ihre Art und Position erzeugen.

Analyseschritte	**Scanner**	**nichtvalidierende Parser**	**validierende Parser**	
			ohne PSVI	**mit PSVI**
Lexikalische Analyse	ja	ja	ja	ja
Syntaktische Analyse				
- XML-Spezifikation		ja	ja	ja
- Schema			ja	ja

Tabelle 3.1: Gegenüberstellung von Analysekomponenten und unterstützte Analyseschritten.

Tabelle 3.1 stellt die Begrifflichkeiten von Analysekomponenten den Teilschritten der Analysephase, die sie umsetzen, zusammenfassend gegenüber.

3.2.2 Transformation

In der Transformationsphase wird die eigentliche Überführung der internen Repräsentation der XML-Daten in eine andere abstrakte, interne Zielstruktur, die gegebenenfalls wieder ein XML-Text repräsentiert, oder direkt in eine konkrete Zielstruktur in Abhängigkeit der Transformationsdefinition vorgenommen.

Eine **Transformationsdefinition** besteht aus einer Menge von Transformationsregeln, die in ihrer Gesamtheit beschreiben, wie ein XML-Dokument in einer bestimmten Quellsprache in ein Dokument in einer respektiven Zielsprache transformiert wird. Eine **Transformationsregel** selbst stellt eine Beschreibung dar, wie ein oder mehrere Konstrukte der (abstrakten) Quellsprache in ein oder mehrere Konstrukte der (abstrakten) Zielsprache überführt werden (vgl. [MVG06, KKF06]). Eine Transformationsdefinition kann somit als Modell einer konkreten Transformationsaufgabe oder -funktion angesehen werden [FN05], die als Input ebenso mehrere Quelldokumente haben bzw. analog als Output mehrere Zieldokumente erzeugen kann.

Da Transformationsdefinitionen bestimmte Transformationsfunktionen modellieren, können sie grundsätzlich auf ganz unterschiedliche Weise und in unterschiedlichen Sprachen beschrieben werden. Das Spektrum reicht von der Nutzung universeller Programmiersprachen bis hin zu spezifischen XML-Transformationssprachen. Universelle Programmiersprachen bieten APIs, um auf die Daten von XML-Dokumenten zugreifen und diese verarbeiten zu können. Die eigentliche Transformation erfolgt mit den Sprachkonstrukten der universellen Programmiersprache. Spezifische Transformationssprachen stellen hingegen spezielle Sprachkonstrukte für die Verarbeitungsphase bereit, die sich an den Terminologien und Konzepten der XML-Transformationsdomäne orientieren. Zudem kommen XML-spezifische Verarbeitungsmodelle zum Einsatz. Zwischen diesen beiden Extremen des Spektrums (universell vs. spezifisch) existieren eine Reihe von Ansätzen. Beispielsweise gibt es Transformationssprachen, die zwar spezifische Sprachkonstrukte allerdings für eine größere Transformationsdomäne zur Verfügung stellen. Entsprechend werden allgemeinere Verarbeitungsmodelle genutzt, bei denen u. U. Informationsverluste auftreten.

Im Weiteren werden wesentliche Gemeinsamkeiten und Unterschiede von Transformationssystemen, -sprachen und -sprachvorschlägen beschrieben. Die folgende Liste enthält die berücksichtigten Systeme und Sprachen:

- **APIs und Erweiterungen für universelle Programmiersprachen**
  Castor [Cas08], DOM [HHW+04], HaXmL [Wal07], JAXB [VF06], XACT [CKM04, KMS04], XJ [HRS+05], xmerl [xme08], XOBE [KL02], Xtatic [GGP06]
- **universelle Transformationssprachen und -systeme**
  ASF+SDF [ASF07], Scrimshaw [Arn93], SIMON [FW93], Stratego [Vis04, BKVV05], SynTree [TT02], TXL [CCH07, Cor06]
- **spezifische XML-Transformationssprachen und -systeme**
  CDuce [BCF03, CDu08], fxt [BS02a, fxt05], STX [Bec03, CBN+07], TREX [ZWG+03], XDuce [HP03, XDu05], XML Script [Dec02], XSLT [Cla99a, Kay07], XUL [CEP99], XUpdate [LM00]

Grundsätzlich können auch andere Technologieräume genutzt werden, um XML-Dokumente zu transformieren. Beispielsweise ist es möglich, Modelltransformationssprachen bzw. Modelltransformationswerkzeuge (wie z. B. openArchitectureWare [oAW08], ATL [BDJ+03, Gro08]) zu nutzen, indem Brücken zwischen den Technologieräumen gebaut werden. Entsprechend muss das Schema des zu transformierenden XML-Dokumentes bzw. das XML-Dokument selbst in entsprechende Metamodelle und Modelle überführt werden. Das Hauptproblem bei der Verwendung eines anderen Technologieraums besteht darin, dass die eingesetzten Verarbeitungsmodelle unterschiedliche Mächtigkeiten aufweisen. Ungeachtet dessen ist es aus verständlichen Gründen unmöglich alle vorhandenen Systeme und Sprachen in die Domänenanalyse einzubeziehen. Trotzdem wird versucht, möglichst viele unterschiedliche Ansätze zu untersuchen.

Zur Veranschaulichung der Ergebnisse der Domänenanalyse werden in Anlehnung an [CH03, CH06] Merkmaldiagramme verwendet. Ein **Merkmal** bildet dabei eine markante und unterscheidbare Eigenschaft eines Konzeptes ab. Merkmaldiagramme veranschaulichen graphisch die

hierarchischen Beziehungen und Abhängigkeiten von Merkmalen untereinander [CE00]. Abbildung 3.2 zeigt das Merkmaldiagramm auf oberster Ebene. Jeder Subknoten repräsentiert einen Variationspunkt, der jeweils in einem eigenen Unterabschnitt im Einzelnen diskutiert wird.

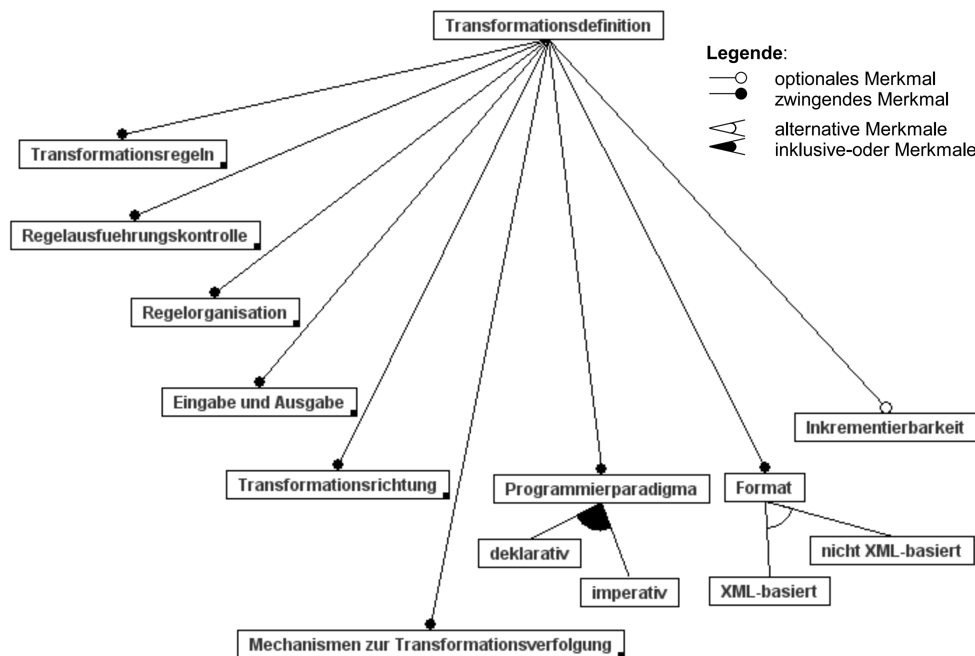


Abbildung 3.2: Merkmaldiagramm auf der obersten Ebene.

3.2.2.1 Transformationsregeln

Eine Transformationsregel im engeren Sinne besteht, wie eine Regel im Allgemeinen, aus einer Prämisse und einer Konklusion. Die Prämisse ist die Bedingung, wann eine Regel angewendet werden darf, und die Konklusion beschreibt die Auswirkung bei ihrer Ausführung. Im weiteren Sinne kann unter dem Begriff Transformationsregel ein Baustein zur Beschreibung eines Teils einer Transformation verstanden werden. Im Kontext dieser Arbeit soll die breitere Definition gebraucht werden, die bspw. Funktionen oder Prozeduren, welche einen Transformationsschritt implementieren, ebenso unter dem Begriff Transformationsregel subsummieren. Abbildung 3.3 gibt die Merkmale von Transformationsregeln in graphischer Form wieder.

Domänen

Transformationsregeln beziehen sich auf Domänen. Regeln haben typischerweise eine Quelldomäne (*left hand side*, kurz LHS) und eine Zieldomäne (*right hand side*, kurz RHS). Es können aber auch mehrere Domänen involviert sein. Dies ist bspw. notwendig, wenn zwei oder mehrere XML-Dokumente zu einem überführt werden. Bei diesem Beispiel wird auf der LHS auf mehrere Domänen zugegriffen. Jede Domäne besitzt dabei i. d. R. ihre eigene, individuelle Sprache, in der die Strukturen beschrieben sind. Diese wird oftmals als domänenspezifische Sprache (*Domain*

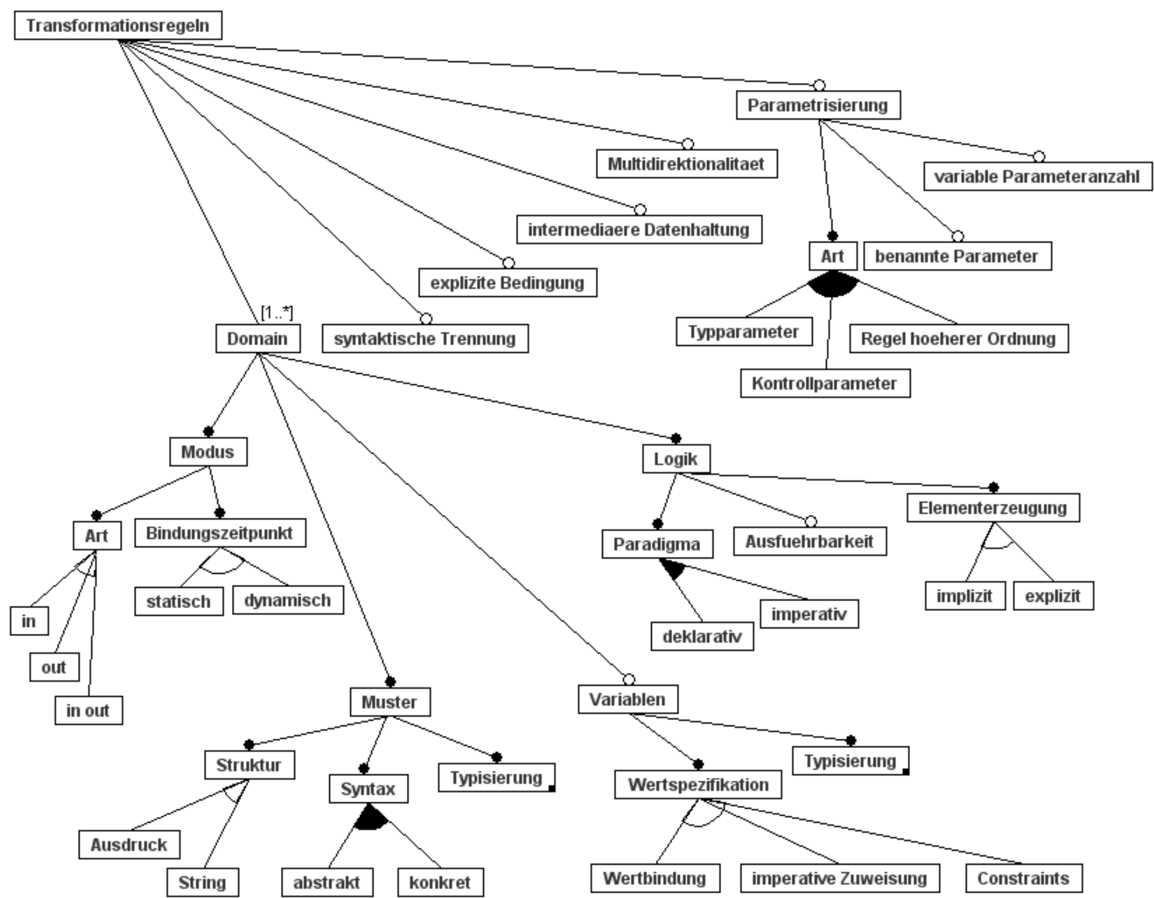


ABBILDUNG 3.3: Merkmale der Transformationsregeln.

Specific Language, kurz DSL) bezeichnet. Transformationen, bei denen sich das Quelldokument und das Zieldokument mit derselben Sprache beschreiben lassen, werden in der Literatur endogene Transformationen genannt. Ist dies nicht möglich, spricht man von exogenen Transformationen [MVG06].

Den Domänen kann darüber hinaus ein Modus zugeordnet werden: in, out, in/out. Dieser legt fest, wie die Domänen innerhalb der Transformationsregeln gebraucht werden: als Quell- oder Zieldomäne oder beides. Die Festlegung kann implizit oder explizit erfolgen. Im Gegensatz zu einer statischen Deklaration des Modus kann er auch erst zur Laufzeit dynamisch bestimmt werden. Die meisten Transformationssysteme unterstützen einen impliziten, statischen Modus.

Zum Zugriff auf die Domänen werden in Transformationsregeln Muster und Variablen verwendet. Muster dienen auf der einen Seite in der Prämisse der Transformationsregel als Suchmuster. Durch das Suchmuster werden zum einen die Bedingung für die Anwendung der Transformationsregel bestimmt und zum anderen mglw. gleichzeitig Informationseinheiten bspw. aus Quelldokumenten bzw. deren internen Verarbeitungsmodellen selektiert. Suchmuster werden typischerweise mit Ausdrücken definiert. Auf der anderen Seite können Muster in der Konklusion der Transformationsregel als Ersetzungsmuster genutzt werden. In einigen Sprachen kann zur Definition von Mustern eine eigene Lokalisierungssprache, wie z. B. XPath, verwendet werden. Dies ist aber nicht zwingend notwendig. Die Navigation im Verarbeitungsmodell in einer imperativen Sprache zu einer gesuchten Informationseinheit kann ebenso ein Suchmuster realisieren. In beiden Fällen handelt es sich um eine abstrakte Syntax. Bei Template-basierten Transformationsmethoden können die Ersetzungsmuster auch in der konkreten Syntax des Zieldokumentes formuliert werden (siehe Abschnitt 3.2.2.9). Diese Stringmuster werden i. d. R. durch Metacode, zum Zugriff auf Informationen und Kontrollstrukturen zur Steuerung der Verarbeitungsreihenfolge, ergänzt.

Variablen sind Container für Informationseinheiten eines Quell- bzw. Zieldokumentes oder einer Zwischenrepräsentation. Um sie von den Variablen innerhalb der zu transformierenden Dokumente zu unterscheiden, werden sie auch Metavariablen genannt [CH03]. Die Werte der Variablen können indirekt über Einschränkungen oder direkt über imperative Wertzuweisung oder deklarativ durch Wertbindung, wie bei der funktionalen Programmierung, spezifiziert werden. Beide, Variablen und Muster, können ungetypt oder getypt auftreten. Bei der Typisierung kann in diverse Kategorien unterschieden werden, in denen sich ein Typsystem einordnen lässt (vgl. Abbildung 3.4). Beispielsweise muss bei einer statischen Typisierung während der Kompilierung der Datentyp einer Variable bekannt sein. Dies kann durch Typinferenz (Ableitung des Datentyps) oder durch explizite Typisierung geschehen. Ein stark getyptes Typsystem einer Transformationssprache kann für eine Transformation statisch sichern, dass diese lediglich korrekte Ergebnisse bzgl. der Zielsprachen erzeugt.

Die zugrundeliegende Logik zur Beschreibung von Berechnungen und Einschränkungen innerhalb der Transformationsregeln kann unterschiedlichen Programmierparadigmen folgen. Beispiele für die deklarative Logik sind XPath-Ausdrücke für die Selektierung von Informationseinheiten sowie die implizite Erzeugung von Zielkonstrukten durch Einschränkungen bei Wertebelegungen. Imperative Logik hingegen verwendet i. d. R. APIs, um die internen Verarbeitungsmodelle direkt

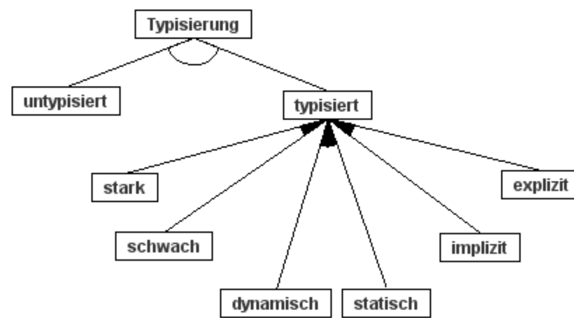


ABBILDUNG 3.4: Merkmale der Typisierung.

zu verarbeiten. Die Verarbeitungsreihenfolge und Erzeugung von Zielkonstrukten erfolgt dabei explizit durch Variablenzuweisungen und Kontrollstrukturen. Neben ausführbaren Transformationsregeln können auch nichtausführbare Regeln definiert werden. Diese definieren allein die Beziehungen zwischen den Quell- und Zieldomänen.

Syntaktische Trennung

Einige Transformationskonzepte unterstützen eine klare syntaktische Trennung innerhalb der Transformationsregeln zwischen Teilen der Regel, die sich auf eine Domäne beziehen, und Teilen der Regel, die sich auf eine andere Domäne beziehen. Ein typisches Beispiel sind klassische Ersetzungsregeln, bei denen die linke Seite (LHS) durch die rechte Seite (RHS) ersetzt wird. Wiederum andere Sprachansätze nehmen nicht eine solche klare Trennung vor, wie z. B. Transformationsregeln, welche mit APIs oder Erweiterungen von universellen Programmiersprachen implementiert werden, oder erlauben eine Mischung aus beiden.

Explizite Bedingung

Transformationsregeln haben i. d. R. eine explizite Bedingung in der Prämisse, die erfüllt sein muss, damit die Konklusion der Regel ausgeführt wird. Eine solche Bedingung kann aber grundsätzlich fehlen. Bedingungslose Transformationsregeln könnten bspw. von anderen Regeln aufgerufen werden oder einfache Default-Regeln sein, die ausgeführt werden, wenn keine andere Regel zutrifft.

Intermediäre Datenhaltung

Manche Sprachkonzepte erlauben Zwischenrepräsentationen von Daten in einer Transformationsregel. Im engeren Sinne ist jede Variablenzuweisung bei imperativen Sprachen eine solche Zwischenrepräsentation. Aber auch einige Sprachen mit anderen Programmierparadigmen können temporäre Strukturen, z. B. in XSLT 2.0 einen temporären Baum, repräsentieren. Neben dieser i. d. R. temporären, intermediären Datenhaltung können Daten auch persistent zwischengespeichert werden, sobald mehrere Outputs möglich sind und diese innerhalb der Transformationsregel wieder eingelesen werden können.

Multidirektionalität

Transformationsregeln können in verschiedenen Richtungen definiert sein. Die einfachste und am häufigsten eingesetzte Form sind unidirektionale Transformationsregeln. Sie geben eine Abbildung von einer Quelldomäne zu einer Zieldomäne an. Es sind aber ebenso multidirektionale Transformationsregeln denkbar, die in verschiedenen Richtungen ausgeführt werden können. Sie können über in/out-Domänen beschrieben werden.

Parametrisierung

Eine Transformationsregel kann zusätzliche Parameter übergeben bekommen, die bei der Verarbeitung der Regel berücksichtigt werden. So können bspw. Kontrollparameter den weiteren Ablauf steuern. Neben der Übergabe oder Referenzierung von Werten eines bestimmten Typs können bei einigen Sprachen (z. B. C++, Java ab J2SE 5.0) Typparameter zur Konfiguration von generischen Datentypen einer Transformationsregel als Argumente mitgegeben werden. Dadurch wird eine höhere Wiederverwendbarkeit einer Transformationsregel erreicht. Eine andere Möglichkeit, um die Wiederverwendbarkeit zu erhöhen, sind Transformationsregeln höherer Ordnung. Diese können als Argumente wiederum Transformationsregeln enthalten oder als Ergebnis eine Transformationsregel liefern. Bei einigen Sprachen können Argumente auch benannt werden. Dadurch wird der Code verständlicher. Ebenso ist es dadurch möglich, die Reihenfolge der Argumente zu verändern. Manche Sprachen, wie XSLT, ermöglichen darüber hinaus Parameter wegzulassen. Dieses Konzept erlaubt es allerdings nicht, mehrere Transformationsregeln mit gleichen Namen und unterschiedlichen Parametern zuzulassen.

3.2.2.2 Regelausführungskontrolle

Drei wesentliche Mechanismen kontrollieren die Regelausführung: die Traversierung durch eine Domäne, Auswahlstrategien der Transformationsregeln, falls mehrere Regeln anwendbar sind, und sonstige Kontrollstrukturen zur expliziten Steuerung der Ausführungsverarbeitung (vgl. Abbildung 3.5).

Traversierung

Die Festlegung in welcher Reihenfolge Transformationsregeln in einer Transformationsdefinition ausgeführt werden, kann implizit oder explizit erfolgen. Implizit bedeutet, dass die Transformationssprache eine Ausführungsreihenfolge standardmäßig vorgibt. Beispielsweise wird in STX das Quelldokument von oben nach unten (*top-down*) in Präorderreihenfolge durchlaufen und es werden die Regeln angewendet, bei denen die kontextsensitiven Bedingungen für den aktuellen Kontext erfüllt sind. Andere vordefinierte Traversierungsstrategien, wie *bottom-up*, *innermost* oder *outermost*, sind ebenso denkbar (z. B. Stratego). Auf der anderen Seite kann bspw. in XSLT die standardmäßige Traversierung in den Transformationsregeln gesteuert bzw. überschrieben werden. Dadurch wird eine andere Ausführungsreihenfolge impliziert. Es ist in XSLT, wie in imperativen und funktionalen Sprachen, ebenso möglich Transformationsregeln

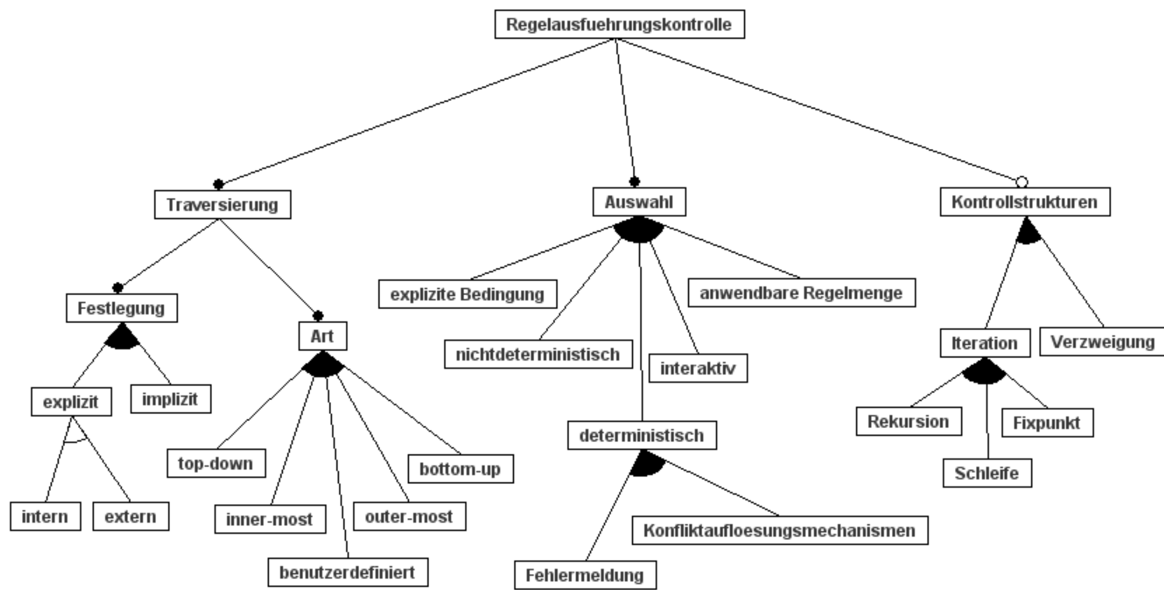


ABBILDUNG 3.5: Merkmale der Regelausführungskontrolle.

explizit aufzurufen. Neben einem solchen internen Mechanismus kann die Ausführungslogik von den Transformationsregeln auch strikt getrennt und extern definiert werden. Einige Transformationssprachen, wie z. B. Stratego, bieten zudem die Möglichkeit benutzerdefinierte Traversierungsstrategien zu definieren. Die Traversierungsstrategien werden dabei in aller Regel bzgl. des Quelldokumentes angewendet. Eine Ausnahme bildet TREX. Bei diesem zielstrukturgetriebenen Transformationssystem richtet sich die Verarbeitungsreihenfolge nach der Traversierung durch das Zieldokument.

Auswahl

Können zu einem Zeitpunkt mehrere Transformationsregeln angewendet werden, weil bei allen die Bedingungen in ihrer Prämisse erfüllt sind, so wird eine Strategie für die Auswahl der Regeln benötigt. Dies kann im Wesentlichen auf drei verschiedenen Arten erfolgen: deterministisch, nichtdeterministisch und interaktiv. Die übliche verwendete Form ist eine deterministische Auswahl. Die wohl einfachste Form ist das Suchen und Ausführen der ersten passenden Transformationsregel (z. B. `case`-Anweisung in einer universellen Programmiersprachen). In einigen Sprachen sind aber auch komplexere Konfliktlösungsmechanismen implementiert. Beispielsweise werden in manchen explizit priorisierte Transformationsregeln bevorzugt (z. B. XSLT). Fehlt eine explizite Priorisierung oder hat diese keine Entscheidung gebracht, so können weitere Kriterien zur Konfliktauflösung herangezogen werden. Zum Beispiel kann eine nicht importierte Transformationsregel oder eine Regel mit genauer spezifizierten Bedingungen den Vorzug erhalten. Weniger häufig sind denkbare nichtdeterministische und interaktive Auswahlverfahren anzutreffen. In einigen Transformationssprachen können auch Modi (z. B. XSLT) oder Bezeichner (z. B. Stratego) für Transformationsregeln definiert werden, mit denen die Regelmenge, die zum aktuellen Zeitpunkt überhaupt angewendet werden darf, eingeschränkt werden kann.

Kontrollstrukturen

Kontrollstrukturen ermöglichen es, die Verarbeitungsreihenfolge von Transformationsregeln zu steuern. Eine solche Ablaufsteuerung kann innerhalb und außerhalb der Regeln definiert sein. Die wesentlichen Kontrollstrukturen sind Verzweigungen und Iterationen. Bei Verzweigungen wird mit Hilfe einer oder mehrerer Bedingungen eine Entscheidung über den weiteren Ablauf getroffen.

Zum Iterieren (lat. *itare* wiederholen) einer Transformationsregel stellen Sprachen unterschiedliche Konzepte zur Verfügung. Eines, das nahezu alle Sprache bereitstellen, ist die Rekursion. Dabei ruft sich die Transformationsregel solange selbst auf, bis (z. B. aufgrund einer Bedingung) der rekursive Aufruf übersprungen wird. Schleifen sind hingegen explizite Sprachkonstrukte zum Wiederholen einer Transformationsregel und werden nicht in allen Sprachen angeboten, z. B. XDuce oder CDuce. Wiederum andere Sprachen, wie XSLT und STX, bieten nur einen eingeschränkten Schleifenkonstrukt an, der sich in der Abbruchbedingung nur auf Kontextbedingungen bzgl. seiner Quelldokumente beziehen kann. Schleifen bestehen aus einem Schleifenrumpf, der den Code enthält, welcher solange ausgeführt wird, bis eine Abbruchbedingung eintritt. Die Abbruchbedingung wird entweder im Schleifenkopf oder im Schleifenfuß definiert. Alternativ zu diesen beiden Konzepten kann der Fixpunktansatz erwähnt werden (z. B. Stratego). Dabei wird eine Transformationsregel solange wiederholt, bis keine Änderungen mehr auftreten. Dieser Ansatz muss i. d. R. in den Sprachen explizit implementiert werden.

3.2.2.3 Organisation von Regeln

Transformationsregeln können bspw. für eine einfachere Entwicklung, aber auch für eine bessere Wartbarkeit und Wiederverwendbarkeit, strukturiert werden. Transformationsregeln bilden selbst die kleinste Einheit innerhalb einer Transformationsdefinition. Sprachen bieten verschiedene Mittel zur Organisation dieser Regeln (vgl. die Merkmale in der Abbildung 3.6). Manche Sprachen erlauben es, Transformationsregeln hinsichtlich bestimmter Kriterien in logische Einheiten zu sogenannten Modulen zusammenzufassen. So können Transformationsdefinitionen in größere funktionale Teilblöcke zerlegt werden. Module können andere Module importieren, um sie miteinander zu verknüpfen und auf deren Inhalt zuzugreifen.

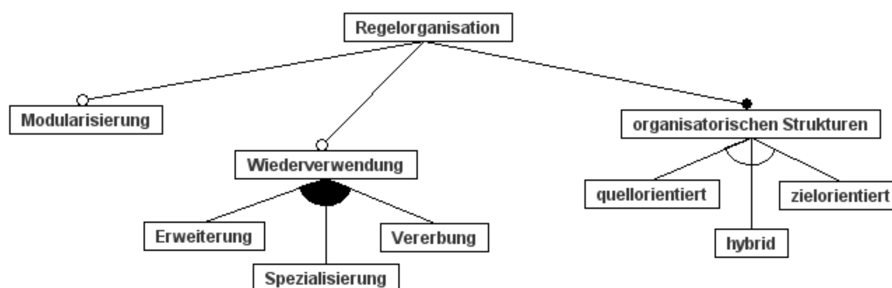


Abbildung 3.6: Merkmale der Regelorganisation.

Neben dieser Komposition von Transformationsregeln sind weitere Wiederverwendungsmechanismen zu finden, wie z. B.:

- Erweiterung von Transformationsregeln und Modulen
- Spezialisierung von Tranformationsregeln und Modulen
- Vererbung von Transformationsregeln und Modulen

Die Organisationsstruktur der Transformationsregeln wird durch die Transformationsmethode, die verfolgt wird, wesentlich beeinflusst. Bei einer quellstrukturgetriebenen Transformationsmethode werden die Regeln in Abhängigkeit der Quelldokumente ausgelöst. Dies kann bspw. durch induzierte Ereignisse oder durch eine vorgegebene oder konfigurierbare Traversierung durch die Quelldomäne erfolgen. Die Reihenfolge, in der das Ziel konstruiert wird, wird maßgeblich durch die Struktur der Quelldokumente bewirkt. Im Gegensatz dazu steht bei einer zielstrukturgetriebenen Transformationsmethode die Struktur des Zieldokumentes bspw. in Form eines Templates weitestgehend fest. Die offenen Stellen des Templates werden durch extrahierte Informationen von mglw. unterschiedlichen Quellen gefüllt. Entsprechend den Ansätzen werden die Regeln quellorientiert oder zielorientiert strukturiert. Verfolgt eine Sprache einen gemischten Ansatz, können die Regeln ebenso unabhängig organisatorisch strukturiert werden.

3.2.2.4 Input und Output

Transformationskonzepte können bzgl. des Inputs und Outputs unterschieden werden, ob sie neue Zieldokumente aus Quelldokumenten bzw. deren Verarbeitungsmodellen erzeugen (CDuce, STX, XMLLamda, XSLT, etc.) oder auf ein und demselben Verarbeitungsmodell arbeiten (XUL, XUpdate, etc.). Sprachen, die nicht zwischen Quell- und Zieldokument differenzieren, werden auch Update- oder Änderungssprachen genannt. Beide Konzepte haben ihre Vor- und Nachteile. Änderungssprachen können nur XML-Daten manipulieren. Reine Transformationssprachen können XML-Daten in XML-Dokumente und in andere Datenformate überführen. Im Kontext von XML-zu-XML-Transformationen sollte deshalb der Transformationsansatz bevorzugt werden, wenn das Zieldokument im Vergleich zum Quelldokument eine komplett unterschiedliche Struktur aufweist. Änderungssprachen sollten hingegen in erster Linie verwendet werden, wenn hauptsächlich der Inhalt des Quelldokumentes modifiziert wird. Universellen Programmiersprachen können beide Ansätze gleichzeitig umsetzen. Abbildung 3.7 zeigt diese Eigenschaften graphisch in einem Merkmaldiagramm.

Um nicht den ganzen Input (bei quellstrukturgetriebenen Transformationsmethoden) oder Output (bei zielstrukturgetriebenen Transformationsmethoden) zu durchlaufen, könnte sich lediglich auf einen Bereich eines Dokumentes beschränkt werden. Nur dieser soll in die Transformation involviert sein. Dies kann u. U. zu signifikanten Performanzvorteilen führen, da nicht für das gesamte Dokument geprüft werden muss, ob in der gerade aktiven Position Transformationsregeln anwendbar sind. Ferner können sich ebenso die Transformationsregeln vereinfachen, da

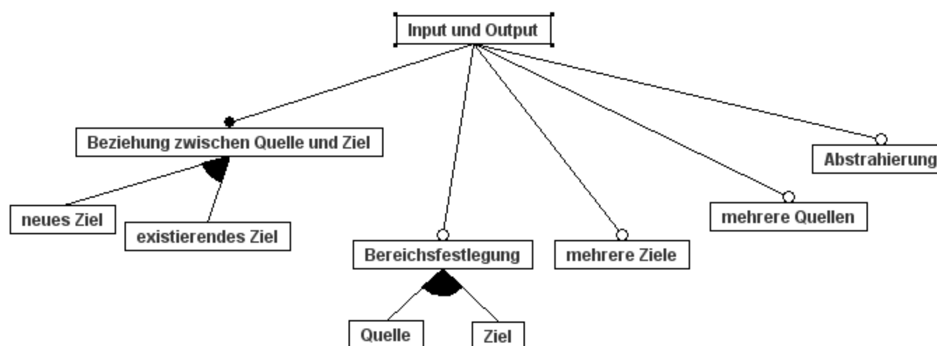


Abbildung 3.7: Merkmale der Inputs und Outputs.

bspw. bestimmte Voraussetzungen für diesen lokalen Bereich erfüllt werden und dadurch in der Transformationsregel selbst nicht beachtet werden müssen.

Das Mischen von zwei oder mehreren XML-Dokumenten erfordert den Umgang mit mehreren Quelldomänen innerhalb einer Transformationsdefinition. Ebenso gibt es Szenarien, in denen Sprachen mehrere Zieldokumente erzeugen müssen. Unterstützt eine Sprache dies nicht, wie z. B. XQuery oder XSLT 1.0, kann dies bereits ein Ausschlusskriterium sein.

Abhängig davon wie der XML-Text während der Analysephase in eine interne Repräsentation überführt wird (vgl. mit Abschnitt 3.2.1), kann sich der Informationsgehalt der Verarbeitungsmodelle unterscheiden. Dementsprechend kann der Abstraktionsgrad der bereitgestellten Informationen variieren (siehe Abschnitt 3.1). Die interne Repräsentation hat somit eine substantielle Auswirkung auf die möglichen Funktionalitäten einer Transformationssprache.

3.2.2.5 Transformationsrichtung

Transformationen können unidirektional und bidirektional sein (vgl. Abbildung 3.8). Bei unidirektionalen Transformationsdefinitionen kann die Transformation nur in eine Richtung – vom Quelldokument zum Zieldokument – durchgeführt werden. Bidirektionale Transformationsdefinitionen erlauben eine Transformation in beide Richtungen – vom Quelldokument zum Zieldokument und umgekehrt. Die Begriffe Quell- und Zieldokument dienen hierbei nur der Verdeutlichung. Ein Dokument kann bspw. beides sein oder es können mehrere Quell- und Zieldokumente an der Transformation beteiligt sein.

Bidirektionale Transformationsdefinitionen können erreicht werden, indem bidirektionale Transformationsregeln oder sich ergänzende unidirektionale Transformationsregeln, die die Transformation zum Zieldokument und die Umkehrrichtung zum Quelldokument beschreiben, definiert werden. Die Invertierbarkeit einer Transformation hängt neben der Invertierbarkeit der Transformationsregeln auch von der Invertierbarkeit der Ausführungslogik der Regeln ab. Die meisten Transformationskonzepte bieten deshalb keine Bidirektionalität.

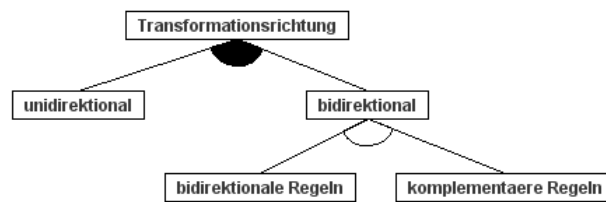


ABBILDUNG 3.8: Merkmale der Transformationsrichtung.

3.2.2.6 Mechanismus zur Transformationsverfolgung

Insbesondere für die Fehlererkennung sind Mechanismen zur Verfolgung der Transformation unverzichtbar. Diese Informationen können bspw. Auskunft geben, welche Einflüsse die Änderungen der Quelldokumente auf die Zieldokumente haben. Einige Sprachen bieten Konstrukte zur manuellen Definition von Fehlermeldungen oder Nachrichten im Allgemeinen an (vgl. Abbildung 3.9). Ein automatisiertes Protokollieren wird derzeit unzureichend unterstützt. Eine Kontrolle, z. B. was protokolliert werden soll, wäre sinnvoll. Vorstellbar wäre eine Konfiguration über die Art der Informationen (z. B. Beziehungen zwischen Quelldomäne und Zieldomäne, Verweis auf die entsprechende Transformationsregel, Zeitpunkt der Erzeugung), der Abstraktion (z. B. Transformationsregeln nur auf höchster Ebene), des Protokollierungsbereiches (z. B. bestimmte Klasse von Regeln oder Bereich in der Quelldomäne) sowie des Ortes, wo die Informationen abgelegt werden (z. B. Quelldokument, Zieldokument, unabhängig von beiden).

ABBILDUNG 3.9: Merkmale der Transformationsverfolgung.

3.2.2.7 Programmierparadigma

Die Sprache, in der die Transformationsdefinitionen beschrieben werden, kann unterschiedlichen Programmierparadigmen folgen. Grundlegende Paradigmen sind die imperative und die deklarative Programmierung. Sowohl universelle Programmiersprachen als auch spezifische Transformationssprachen sind in beiden Kategorien zu finden. Wichtige Ausprägungen sind prozedurale, modulare und objektorientierte bzw. funktionale und logische Programmierung. Die Konzepte einer Sprache werden entscheidend durch ihr Programmierparadigma geprägt.

3.2.2.8 Syntax

Wird die Transformation mit den Mitteln einer universellen Programmiersprachen implementiert, so muss deren Syntax verwendet werden. Kommen hingegen spezifische XML-Transformationssprachen zum Einsatz, so wird häufig eine XML-basierte Notation gewählt (fxt, STX, XML Script, XSLT, XUL, XUpdate, etc.). Dies erlaubt die Nutzung der bestehenden XML-Infrastruktur mit zahlreichen Sprachen und Werkzeugen. Dadurch können bspw. existierende Parser und Serialisierer zur Implementierung eines Interpreters oder Generators wiederverwendet werden.

3.2.2.9 Kategorien von Transformationsmethoden

Transformationssprachen und -systeme unterstützen bestimmte Transformationsmethoden und ihre Konzepte. In diesem Abschnitt werden untersuchte Transformationssprachen und -systeme Kategorien zugeordnet. Jede Kategorie erfüllt dabei dezidierte Merkmale einer Transformation hinsichtlich ihrer Umsetzung. Die gewählte Klassifikation erfolgt dabei nach dem Kriterium des treibenden Elementes: Quellstruktur, Zielstruktur oder beides. Dieses Kriterium beeinflusst vor allem die Organisation der Transformationsregeln maßgeblich. Grundsätzlich sind alternative Kriterien zum Aufbau einer Klassifikation ebenso denkbar. Beispiele für Kriterien sind der Anwendungsbereich (strukturierte vs. XML-spezifische Dokumente), Zweck (XML-zu-XML vs. XML-zu-Text), Korrektheit der Quell- und Zieldokumente [Dvo05] oder Mächtigkeit (universelle Programmiersprachen, universelle Transformationssprachen vs. spezifische XML-Transformationssprachen).

Quellstrukturgetriebene Transformationsmethoden

Bei diesen Transformationsmethoden steuert die Struktur der Quelldokumente wesentlich die Transformation. Vereinfacht gesagt, werden Teile der Zieldomäne erzeugt, sobald bei der Navigation durch die Quelldomäne Bedingungen erfüllt werden. Die Struktur der Zieldokumente hängt dadurch stark von der Quelldokumentstruktur ab. Die dieser Transformationsmethode zugrundeliegende Technik wird demgemäß auch Push-Technik genannt.

- **Direkte Änderungsmethoden**
  Systeme, die diese Transformationsmethode unterstützen, stellen eine API bereit, mit der direkt auf das interne Verarbeitungsmodell lesend und schreibend zugegriffen werden kann. Die Verarbeitungsmodelle können sowohl universelle als auch spezifische Baumrepräsentationen sein (z. B. DOM oder JAXB, vgl. mit Abschnitt 3.1.3). Die API ist i. d. R. in Form eines Frameworks realisiert, welches noch weitere Funktionalitäten anbietet (z. B. Organisation von Transformationsregeln). Die Transformationsregeln selbst, die Regelausführungskontrolle und Mechanismen zur Transformationsverfolgung müssen allerdings typischerweise immer wieder neu und explizit implementiert werden.

Bisher existierende spezifische Änderungssprachen (z. B. XUL oder XUpdate), die ebenso dieser Kategorie zugeordnet werden können, vereinfachen diesen zusätzlichen Entwicklungsaufwand jedoch nur rudimentär an einigen Stellen. Beispielsweise nutzen sie deklarative Lokalisierungssprachen, wie XPath, um durch die Quelldomänen zu navigieren.

- **Suchmustergesteuerte Transformationsmethoden**
  Das zentrale Konzept dieser Kategorie der Transformationsmethoden sind Suchmuster. Ein **Suchmuster** besteht aus eine Menge von Bedingungen bzgl. eines bestimmten Kontextes. Die Bedingungen beschreiben eine gewünschte Struktur, in der bspw. ganz konkrete Typ- oder Wertebelegungen festgelegt sind. Zur Beschreibung dieser Bedingungen können bspw. erweiterte reguläre Ausdrücke verwendet werden (siehe z. B. [HP01] oder [FC04]). Wird bzgl. eines gegebenen Kontextes ein passendes Muster gefunden, welches alle Bedingungen genügt, so werden die in dem Muster explizit oder implizit gebundenen Variablen an das Ersetzungsmuster übergeben. Der gesamte Vorgang der Überprüfung und Übergabe der gebundenen Variablen an das Ersetzungsmuster wird auch *Pattern Matching* genannt. Sind jedoch nicht alle Bedingungen zum aktuellen Kontext erfüllt, wird das nächste angegebene Muster geprüft. Passt keines der Suchmuster, werden in Abhängigkeit des Transformationssystems, Standardroutinen ausgeführt oder eine Fehlermeldung gesendet. Der Kontext, auf welchen das Suchmuster geprüft wird, wird dabei explizit beim Aufruf der Transformationsregel übergeben oder durch das Navigieren in der Quelldomäne implizit bestimmt. Das **Ersetzungsmuster** beschreibt den Teil, der bei der Anwendung der Transformationsregel ausgeführt wird. Er kann bspw. einfache Termersetzungen oder auch Aufrufe weiterer Transformationsregeln beinhalten.

  Vertreter dieser Kategorie sind u. a. die spezifischen XML-Transformationssprachen XDuce und CDuce. Beide funktionale Sprachen zeichnen sich neben dem Pattern Matching-Mechanismus durch ein umfangreiches statisches Typsystem aus. Dadurch ist das Einlesen von fehlerhaften XML-Instanzen sowie das Generieren von ungültigen XML (ungültig im Sinne des Dokumenttyps) nicht möglich. Ebenso können universelle Transformationssprachen, wie ASF+SDF, Scrimshaw, Stratego und TXL, dieser Kategorie zugeordnet werden.

  Bezogen auf die in dieser Arbeit eingeführte Terminologie wird hier ein klare Trennung zwischen LHS und RHS vorgenommen. Die Steuerung der Regelausführung kann dabei intern in der RHS oder extern außerhalb der Transformationsregeln erfolgen, wobei insbesondere bei spezifischen XML-Transformationssprachen i. d. R. das Erstere auftritt.

Zielstrukturgetriebene Transformationensmethoden

Bei diesen Transformationsmethoden steht die Struktur des Zieldokumentes im Vordergrund. Bildlich wird durch die Zieldokumentstruktur navigiert und an entsprechenden Stellen Informationen aus mglw. unterschiedlichen Quellen hinzugefügt. Die Zieldokumentstruktur ist damit von der Quelldokumentstruktur weitgehend unabhängig. Dieser Methode zugrundeliegende Technik wird insofern auch mit Pull-Technik bezeichnet.

- **Template-basierte Transformationsmethoden**
  Bei Template-basierten Transformationen werden die Transformationsregeln mit Hilfe von

Templates beschrieben. **Templates** bestehen aus Konstrukten der Zielsprache und offenen Stellen. Die offenen Stellen enthalten Metacode, der bei der Instantiierung des Templates mit Informationen gefüllt wird oder die weitere Verarbeitungsreihenfolge durch Iterationen oder Aufrufe anderer Templates steuert.

Entsprechend der vorgeschlagenen Terminologie beschreibt die LHS innerhalb der Template-basierten Transformationsmethoden den ausführbaren Code, der auf die Quelldomäne zugreift. Die RHS kombiniert untypisierte Zeichenkettenmuster mit ausführbarer Logik zur Selektion von Informationen und Definition von Kontrollstrukturen. Es findet keine syntaktische Trennung zwischen LHS und RHS statt. Die LHS-Logik zum Zugreifen auf die Informationen der Quelldomäne kann unterschiedliche Ausprägungen haben. In universellen Programmiersprachen kann der Code zum Zugriff auf die API des internen Verarbeitungsmodells die Logik explizit implementieren (z. B. JavaServer Pages™ [Sun03a]). Alternativ können deklarative Pfadausdrücke in eigenen Lokalisierungssprachen eingesetzt werden (siehe z. B. [Har06]). Die Template-basierte Transformationsmethode erlaubt i. d. R. die verschiedensten Formate zu erzeugen. Es gibt allerdings auch Transformationssysteme, wie XACT, die diese Methode lediglich zur Erzeugung von XML-Dokumenten nutzen.

- **Schemabasierte Transformationsmethoden**
  Transformationssprachen dieser Kategorie haben die Besonderheit, dass die Regeln anhand des Zielschemas organisiert werden. Jedem unterstützten Zielelementtyp werden Transformationsregeln zugeordnet. Die Schachtelung der Regeln entspricht den Elementtypen innerhalb des Zielschemas. Das Zielschema wird dadurch mit Transformationsregeln konfiguriert. Ein Transformationssystem, welches diese Transformationsmethode umsetzt, ist TREX. Zur Definition einer Regel, die bspw. auf unterschiedliche Quellen zugreift, wird die Abfragesprache Quilt [CRF01] genutzt.

  Auf der einen Seite kann durch den schemabasierten Ansatz gesichert werden, dass das erzeugte XML-Dokument gültig bzgl. des Zielschemas ist. Auf der anderen Seite ist diese Transformationsmethode weitestgehend unabhängig von der Quelle, so dass sie auch verwendet werden kann, um aus anderen Quellen als aus XML-Dokumenten Informationen zu extrahieren. Die in [BCF+02] vorgestellte Transformationssprache nutzt bspw. SQL als Abfragesprache, um Informationen aus relationalen Datenquellen auszulesen und in das Schema einer XML-Zielsprache an entsprechender Stelle einzufügen.

  Ähnlich wie bei der Template-basierten Transformationsmethode werden hier LHS und RHS nicht klar voneinander getrennt. Im Unterschied zur Template-basierten Transformationsmethode können nur XML-Dokumente erzeugt werden.

Sonstige Transformationsmethoden

Unter dieser Gruppe werden Transformationsmethoden eingeordnet, bei denen die Quell- und Zieldokumentstruktur gleichermaßen die Transformation bestimmt oder Konzepte aus den bisher genannten Kategorien kombiniert werden. Die unterschiedlichen Konzepte können dabei durch separate Komponenten oder feingranularer in der Transformationssprache selbst kombi-

niert werden.

- **Mapping-Ansatz**
  Der Mapping-Ansatz stützt sich auf das mathematische Konzept der Relationen. Es werden Beziehungen zwischen den Quell- und Zielelementtypen durch Relationen spezifiziert. Eine solche Spezifikation ist in ihrer reinen mathematischen Form multidirektional und kann nicht ausgeführt werden. Mapping-Systeme, die eine Transformation zwischen Quell- und Zieldomäne umsetzen, geben den definierten Relationen eine ausführbare Semantik. Diese kann individuell neu festgelegt werden, indem die Relationen durch ausführbare Transformationsregeln detailliert beschrieben werden. Die detaillierten Beschreibungen werden i. d. R. in graphischen Notationsformen (z. B. SynTree, Altova® MapForce [Alt07], Stylus Studio® XML [Dat08]) oder in einer eigenen textuellen Mapping-Sprachen (z. B. SIMON) spezifiziert. Die graphischen und textuellen Transformationsregeln werden entweder mittels universeller Programmiersprachen interpretiert oder es werden entsprechende Transformationsdefinitionen in Java, C++, C# oder XSLT generiert.
- **Suchmustergesteuerter Template-Ansatz**
  Dieser Kategorie können die separaten Transformationssprachen fxt, STX und XSLT zugeordnet werden. Einem Template kann in diesen Sprachen um ein Suchmuster ergänzt werden. Ein solches erweitertes Template ist eine Ausprägung einer konkreten Transformationsregel. Das Suchmuster enthält eine kontextsensitive Bedingung, die gegen den aktuellen Kontext innerhalb des Quelldokumentes geprüft wird. Die Erfüllung des Suchmusters ist eine Voraussetzung zur Ausführung des Templates, welches das Ersetzungsmuster beschreibt.

  Durch die Erweiterung der Templates um Suchmuster können Sprachen dieser Kategorie sowohl Push- als auch Pull-Techniken in einer Transformation umsetzen. Push-Techniken sollten normalerweise bei den Teilen der Transformation angewendet werden, bei denen die Zieldokumentstruktur durch die Quelldokumentstruktur größtenteils geprägt wird. Pull-Techniken hingegen sollten bei den Teilen den Vorzug erhalten, wo die Zieldokumentstruktur unabhängig von der Quelldokumentstruktur ist. Werden beide Techniken kombiniert eingesetzt, so besteht keine klare syntaktische Trennung mehr zwischen LHS und RHS, wie es bspw. bei der suchmustergesteuerten Transformationsmethode der Fall ist.

Bewertung

Obwohl dem suchmustergesteuerten Template-Ansatz in Ausprägung von XSLT in der Praxis die größte Bedeutung zukommt, gibt es nicht *die* Transformationsmethode. Jede der vorgestellten Transformationsmethode hat seine Berechtigung. Beispielsweise für einfache Umbenennung sind Änderungsmethoden oder der Mapping-Ansatz zu bevorzugen. Sollen Daten aus verschiedenen XML-Quellen oder gar aus Datenbanken ausgelesen und in ein XML-Format gespeichert werden, eignet sich der schemabasierte Ansatz gut. Müssen hingegen umfangreiche strukturelle Transformationen vorgenommen werden, sind, je nach Transformationsproblem, suchmustergesteuerte oder Template-basierte Transformationsmethoden geeignet. Ähnliches gilt für die zugeordneten Vertreter der Kategorien. Jede Transformationssprache hat ihre Vorzüge aber auch Schwächen.

Das liegt daran, dass nicht alle Merkmale gleichzeitig unterstützt werden können. In [Bec04] und [Hau06] werden einzelne Transformationssprachen nach eigenen Kriterien evaluiert. [Hau06] schlägt zudem konkrete Verbesserungspotentiale vor.

3.2.3 Serialisierung

Ist das Ergebnis der Transformationsphase eine interne, abstrakte Datenstruktur, so muss diese in eine konkrete Syntax überführt werden. Aufgabe der Serialisierungsphase ist gerade die Abbildung der internen Repräsentationsform in ein externes, physisches Darstellungsformat, um die transienten Datenstrukturen persistent zu machen.

Die Serialisierung aus einer internen XML-Repräsentation, nur diese soll in diesem Abschnitt betrachtet werden, wird unterschiedlich gut unterstützt. Basiert die interne Verarbeitung der XML-Dokumente auf der reinen Textform oder einem datenstrombasierten Verarbeitungsmodell, so steht keine (z. B. SAX) bzw. lediglich rudimentäre (z. B. StAX) Unterstützung bei der Serialisierung bereit. In diesen Fällen muss das Transformationssystem eine eigene Datenstruktur aufbauen und ebenso die Serialisierung implementieren. Verarbeitungsmodelle, die hingegen bereits eine komplette Datenstruktur zur Verfügung stellen, wie z. B. DOM, bieten zusätzliche Routinen zur Erzeugung von wohlgeformten XML-Dokumenten. Da schemaspezifische Verarbeitungsmodelle darüber hinaus nur korrekte Baummanipulationen bzgl. des eingelesenen Schemas erlauben, wird bei dieser internen Repräsentationsform indirekt gesichert, dass nur korrekte XML-Dokumente aus ihr erzeugt werden.

3.3 Implementierungsmethoden

In allen Bereichen der Wissenschaft und der Entwicklung kann man prinzipiell zwischen generischen und spezifischen Ansätzen unterscheiden. Der generische Ansatz bietet eine allgemeine Lösung für viele mögliche Probleme. Bei bestimmten Anwendungsgebieten kann jedoch eine solche Lösung suboptimal sein. Womöglich erlaubt gerade hier ein spezifischer Ansatz, der nur für einen kleinen Problembereich eingesetzt werden kann, eine bessere Lösung. Wie bereits gesehen, können auch im Bereich der XML-Verarbeitung existierende, zumeist universelle Programmiersprachen erweitert oder angepasst (interne DSL) oder eigenständige spezifische Transformationssprachen entwickelt werden (externe DSL). Die wesentlichen Implementierungsmethoden dieser beiden Ansätze werden im Folgenden vorgestellt und anschließend zentrale Vor- und Nachteile diskutiert.

3.3.1 Interne DSL

Universelle Programmiersprachen bieten verschiedene Mechanismen, um dem Entwickler Unterstützung von wohldefinierten Problemen innerhalb eines bestimmten Anwendungsgebietes, wie der Verarbeitung von XML-Dokumenten, zu geben.

Einbettung

Die Verwendung von Bibliotheken ist bspw. die klassische Methode, um wiederverwendbares Domänenwissen zu sammeln. Bei dieser Methode sind die Übergänge zwischen universeller Programmiersprache und DSL fließend. Hier wird eine DSL implementiert, indem eine existierende universelle Sprache durch die Definition von spezifischen Datentypen und Operatoren erweitert wird. Eine DSL ergibt sich somit aus einer Submenge der universellen Sprache, deren Sprachkonstrukte benutzerdefiniert verwendet werden. Beispielsweise können in Sprachen wie C++ Klassenbibliotheken entwickelt werden, die existierende Operatoren mit domänenspezifischer Semantik überladen. Auch in einer Vielzahl anderer Programmiersprachen (Prolog, Haskell, etc.) wird dem Entwickler erlaubt neue Infix-Operatoren einzuführen und sie wie normale Prädikate oder Funktionen zu definieren. Die in der Bibliothek enthaltene zusätzliche Funktionalität kann durch verschiedenste Applikationen aufgerufen werden.

Einen weiteren Mechanismus bieten Frameworks [JF88, RJ96]. Sie geben i. d. R. eine Anwendungsarchitektur vor, die neben den Schnittstellen wie bei klassischen Bibliotheken ebenso den Kontrollfluss definiert. Der Programmierer kann entsprechend den Schnittstellen spezifische Implementierungen erstellen und registrieren, die dann durch das Framework aktiv gesteuert werden. Da insbesondere bei Black-Box-Frameworks die Schnittstellen im Vordergrund stehen, werden beide Mechanismen im Folgenden unter APIs zusammengefasst.

Auch im Bereich der XML-Verarbeitung wurden diverse APIs für universelle Programmiersprachen (z. B. Haskell, Erlang, C++ oder Java) entworfen, um dem Entwickler bestimmte Aufgaben abzunehmen oder zu erleichtern. So entstanden kurz nach der Veröffentlichung der ersten Vorversionen der XML-Spezifikation im November 1996 [BS96] und Dezember 1997 [BPS97], abgesehen von den bereits existierenden APIs zur lexikalischen, textbasierten Verarbeitung, eine Reihe von Parsern, die XML-Dokumente einlesen und analysieren konnten (in Java z. B. Ælfred [Meg97], Lark [Bra98]). Jeder dieser Parser erzeugte allerdings in der Analysephase eine andere abstrakte Zwischenrepräsentation und bot dementsprechend andere APIs zum Zugriff auf die XML-Daten an.

Zeitig wurde die Notwendigkeit erkannt, die ungleichen, proprietären APIs zu standardisieren. Beispielsweise entwickelte sich innerhalb eines halben Jahres aus den oben genannten Parsern eine einheitliche, ereignisbasierte API, deren erste stabile Version im Mai 1998 [Meg98] unter dem Namen SAX 1.0 (*simple event-based API for XML*) veröffentlicht wurde (siehe Abschnitt 3.1.2). Parallel zu diesen zunächst programmiersprachenspezifischen APIs wurden ebenso programmiersprachenunabhängige APIs in Gremien und Standardisierungsorganisationen publiziert. Beispielsweise entstand im Oktober 1998 [CBNW98] die erste offizielle Version von DOM (*document object model*), das eine baumbasierte Datenstruktur für XML-Dokumente und Operationen sowohl zur Extraktion von Informationen als auch zur Modifikation der Datenstruktur umfasste. Im Zuge der Weiterentwicklung von Standards im Umfeld von XML (Spezifikationen über Namensräume [BHLT06], XML-Schema [FW04, TBMM04, BM04], etc.) wurden die APIs auf die dadurch entstandenen Erfordernisse weiter angepasst und ergänzt. Die API vom DOM wurde in der dritten Version 2004 [HHW+04] bspw. um Module mit Operatoren zum Validieren

[CKR04] sowie für das Laden und Speichern [SH04] der Datenstruktur ausgebaut. Die universelle Speicherrepräsentation von XML wurde dabei erhalten (siehe auch Abschnitt 3.1.3).

Trotz der weitreichenden Unterstützung von universellen Programmiersprachen durch die bereitgestellten APIs für das Parsen und für die Datenhaltung von XML-Dokumenten müssen die Transformationsdefinitionen weiterhin mit den Sprachkonstrukten der verwendeten Programmiersprache realisiert werden. Durch die Benutzung einer universellen Programmiersprache können einerseits beliebig komplexe Berechnungen durchgeführt werden. Andererseits erschwert die Abhängigkeit von einer Basissprache eine intuitive Umsetzung von einzelnen Transformationsregeln, da in vielen Fällen die optimale anwendungsspezifische Notation jenseits der Möglichkeiten von den zur Verfügung stehenden universellen Sprachkonstrukten liegt. Dies führt tendenziell zu feingranularen und steifen Beschreibungen.

Sprachen mit Metaprogrammierung

Einige Programmiersprachen besitzen integrierte Spracherweiterungsmechanismen. Reflektive Sprachen, wie Smalltalk oder CLOS, ermöglichen es, ihre Sprachdefinition zu erweitern. Ihre Sprachdefinition ist für einen Entwickler als änderbare und erweiterbare Bibliothek zugreifbar. Dadurch besteht die Möglichkeit zu domänenspezifischen Optimierungen sowie Anpassungen der Notation und Semantik bis zu einem bestimmten Grad zur Laufzeit. Wiederum Techniken der statischen Metaprogrammierung, wie z. B. der C++-Template-Mechanismus, erlauben Optimierungen der Syntax und Semantik zur Compile-Zeit (siehe z. B. [Vel95]). Mit Hilfe dieses Template-Mechanismus wurden mathematische Bibliotheken, wie Blitz++ [Vel05], für C++ entwickelt, die eine bekannte, problemspezifische Notation bereitstellen. Einen Einsatz für die Verarbeitung von XML-Dokumenten ist jedoch nicht bekannt.

Wie bei der Nutzung von APIs gibt es aber ebenso bei dieser Implementierungsmethode keine klare Trennung zwischen Anwendungsprogrammierung, Programmierung für eine bestimmte Applikation, und Metaprogrammierung, Weiterentwicklung von Sprachen, Werkzeugen und Bibliotheken. Dies kann leicht zu einem Verständnis- und Wartungsproblem führen. Zudem wird die Metaprogrammierung i. d. R. nur unzureichend unterstützt. Beispielsweise fehlen Mechanismen zur Fehlerbehebung und Fehlerbeseitigung.

Erweiterbare Interpreter und Compiler

Bei dieser Methode wird der Interpreter oder Compiler bspw. mit domänenspezifischen Codegeneration erweitert. Dadurch ist sowohl eine bessere Typprüfung als auch Optimierung im Vergleich zu einem Präprozessor möglich. Bei einigen Entwicklungsframeworks kann dabei auch die gesamte Sprachumgebung, inkl. Editor, Debugger usw., angepasst werden.

Somit wird klar zwischen Anwendungsprogrammierung und Metaprogrammierung unterschieden. Dadurch kann der Entwickler adäquater unterstützt werden. Die Erweiterung insbesondere von Compilern ist aber i. d. R. sehr schwierig, da sie nicht für diese Aufgabe entwickelt werden. Techniken für eine sichere und modulare Erweiterung von Compilern sind derzeit ein offenes Forschungsgebiet (siehe z. B. der Broadway Compiler [GL05], Open C++ Compiler [Chi95]

oder [SG97, Eng99]). Die Anwendung für die Verarbeitung von XML-Dokumenten ist nicht bekannt.

Präprozessoren

Eine der klassischen Methoden für die Implementierung von DSLs ist mit Hilfe von Präprozessoren. Dabei werden existierende Sprachkonstrukte durch zusätzliche DSL-Konstrukte angereichert. Damit die angereicherten Programme ausgeführt werden können, werden in einem Präprozess die zusätzlichen DSL-Konstrukte (auch Makros genannt) in Sprachkonstrukte der Basissprache überführt. Die Überführung kann auf unterschiedliche Weise realisiert werden:

- **lexikalische Makroverarbeitung**
  Lexikalische Makrosprachen erlauben das Ersetzen von Makros durch beliebige Zeichenketten oder Token. Makros können dabei parametrisiert werden, um die Makroverarbeitung zu steuern oder Platzhalter in den Makrodefinitionen durch aktuelle Parameter aufzufüllen. Parameter sind dabei wiederum Zeichenketten. Beispiele für lexikalische Makrosprachen sind der Präprozessor CPP für die universelle Programmiersprache C [KR78], der in TEX eingebaute Makromechanismus [Knu84] oder der universelle Makroprozessor M4 [KR77]. Bei lexikalischen Makrosprachen kann der Präprozessor völlig unabhängig von der konkreten Ausführungstechnologie der Basisprache vorgeschalten werden.

  In [PS02] wird eine Makrosystem für XML vorgeschlagen. Makros werden selbst in XML notiert. Das Makrosystem sucht diese Makros und ersetzt diese durch größere XML-Fragmente. Es können dabei nur lokale Ersetzungen vorgenommen werden.
- **syntaktische Makroverarbeitung**
  Im Gegensatz dazu arbeiten syntaktische Makrosprachen auf dem abstrakten Syntaxbaum [Lea66]. Diese Methode verlangt zum einen Wissen über die Basissprache und seine Grammatik sowie zum anderen einen Zugriff auf deren Syntaxbaum. Beispiele sind Scheme [KW87] oder <bigwig> [BS02b]. Die Makros bei diesen Sprachen werden auf syntaktische Korrektheit geprüft und vermeiden darüber hinaus Konflikte mit existierenden Symboldefinitionen während der Makroverarbeitung.
- **Transformation**
  Bei der erweiterten Form der syntaktischen Makroverarbeitung wird eine komplette Quellsprache in eine andere Sprache transformiert. Es wird sich somit nicht auf vereinzelte, meist restriktive Makros beschränkt. Der Übergang von der Makroverarbeitung ist allerdings fließend. Ein Beispiel für diese erweiterte Form ist SWUL (*Swing User-interface Language*) [BV04]. SWUL wird direkt im Java-Code verwendet und soll die Beschreibung von Benutzerschnittstellen vereinfachen, indem die Struktur der Swing-Komponenten wiedergegeben werden kann. Im Gegensatz zu völlig eigenständigen DSLs kann in SWUL auf Java-Code referenziert werden bzw. umgekehrt.

Mit dieser Implementierungsmethode können grundsätzlich eine von der Basissprache unabhängige Syntax sowie Programmierparadigma für die domänenspezifischen Konstrukte gewählt werden. Für die XML-Verarbeitung wird diese Implementierungsmethode bspw. genutzt, um native

Datentypen oder XML-spezifische Transformationsoperatoren in die universelle Programmiersprache zu integrieren. XACT [CKM04, KMS04], XJ [HRS+05], XOBE [KL02] sind z. B. Vorschläge Java und Xtatic [GGP06] C# mit domänenspezifischen Sprachkonstrukten zu erweitern. Die Entwicklungskosten sind im Vergleich zur Neuentwicklung relativ gering. Nichtdestotrotz müssen Abstriche bei der Optimierung und Fehlererkennung gemacht werden.

3.3.2 Externe DSL

Die klassische Methode, um eine externe DSL zu implementieren, ist das Erzeugen von Compilern oder Interpretern. Beide Implementierungsmethoden werden kurz vorgestellt.

- **Compiler**
  Compiler sind Applikationen, die ein DSL-Programm in ein Programm einer niedrigeren Sprache übersetzen. Üblicherweise wird der Begriff Compiler verwendet, wenn die Zielsprache eine Assemblersprache, Bytecode oder Maschinensprache ist. Unabhängig davon läuft die Kompilierung, der Vorgang der Übersetzung, in mehreren Phasen ab. Die wesentlichsten Phasen sind die Analysephase, bestehend aus lexikalischer, syntaktischer und semantischer Analyse, und die Synthesephase, bestehend aus Zwischencodeerzeugung, Programmoptimierung und Codegenerierung. Die verschiedenen Phasen können u. U. mit eigenständigen Applikationen realisiert werden (Scanner, Präcompiler, etc.). Mit dieser Methode wurde die XML-Transformationssprache fxt [fxt05] implementiert.
- **Interpreter**
  Interpreter sind Applikationen, die im Gegensatz zu Compilern ein DSL-Programm nicht übersetzen, sondern einlesen, analysieren und ausführen. Da die Analyse zur Laufzeit erfolgt, ist die Ausführungsgeschwindigkeit deutlich langsamer als bei Compilern. Diesem Nachteil steht die leichtere Implementierung sowie spätere Erweiterbarkeit gegenüber. Eine Vielzahl von DSLs auch im Bereich der XML-Verarbeitung sind mit Hilfe eines Interpreter implementiert worden: Saxon [Kay08], XALAN [Apa07] oder XT [Cla99b] für XSLT, Joost [Joo08] für STX, Jaxup [Bol03] für XUpdate, etc.

Die Übergänge zwischen reinen Compilern und reinen Interpretern sind fließend. Eine Kompromisslösung bspw. ist ein JIT-Compiler (*Just-In-Time*-Compiler), bei dem das DSL-Programm erst zur Laufzeit, jedoch direkt in Maschinencode übersetzt wird. Danach wird der übersetzte Code direkt vom Prozessor ausgeführt. Durch Zwischenspeicherung des Maschinencodes müssen mehrfach durchlaufene Programmteile nur einmal übersetzt werden. Auch ermöglicht der JIT-Compiler eine stärkere Optimierung des Binärcodes (siehe z. B. [DS84]).

3.3.3 Bewertung

In diesem Abschnitt werden die Vor- und Nachteile der zwei grundlegenden Ansätze – externe vs. interne DSL – gegenübergestellt. Der unbestritten größte Vorteil der externen DSL ist, dass

keine Einschränkungen bzgl. der Notation, Datentypen oder Sonstigem gemacht werden müssen. Weitere Vorteile der Neuentwicklung sind:

- Die DSL-Syntax kann so eng wie überhaupt möglich an die verwendete Syntax der Domänenexperten angelehnt werden, wodurch Domänenexperten die DSL-Programme vergleichsweise gut verstehen, prüfen oder selbst modifizieren können, da das Domänenwissen direkt in der Notation verkörpert werden kann. Die Grenzen liegen dabei lediglich bei der Fähigkeit, einen Sprachprozessor zu konstruieren, der die Syntax einlesen und weiterverarbeiten kann.
- Analysen, Verifikationen, Fehlererkennungen und -meldungen, Optimierungen und Transformationen können auf Domänenebene erfolgen.

Die wesentlichen Nachteile sind:

- Eine DSL wird komplett neu entworfen und tendiert daher eher zu unvollständigen Entwürfen im Vergleich zu Erweiterungen von universellen Sprachen.
- Der Entwicklungs-, Implementierungs- und Wartungsaufwand ist groß, da ein komplexer Sprachprozessor implementiert werden muss.
- Nachträgliche Erweiterungen dieses Sprachprozessors sind schwierig möglich, da er i. d. R. nicht dafür entworfen wird.
- Es entstehen Sprachbarrieren zwischen universellen Sprachen und externen DSLs. Programmiersprachenumgebungen werden dadurch immer komplexer.

Einige dieser Nachteile können durch Werkzeuge minimiert oder sogar eliminiert werden. Diese Werkzeuge reichen von Generatoren, die Lexer oder Parser erzeugen (z. B. Lex [LS75] und Yacc [Joh75] oder GNUs Flex [Fre95] und Bison [Fre06]), bis hin zu umfangreichen Entwicklungsumgebungen (z. B. Draco [Nei86], ASF+SDF [DHK96, ASF07], Kephera [FNP97], Kodiyak [HB88] oder InfoWiz [NJ97]). Eine umfassende Liste von Werkzeugen kann in [Ger06] gefunden werden. In [MHS05] werden einige von ihnen vorgestellt bzw. wird auf die entsprechenden wissenschaftlichen Beiträge verwiesen.

Bei der Erweiterung einer Basissprache bleiben hingegen alle Funktionalitäten der Basissprache erhalten und müssen dementsprechend nicht erneut implementiert werden. Folgende zentrale Vorteile ergeben sich daraus:

- Der Entwicklungs-, Implementierungs- und Wartungsaufwand ist moderater, da existierende Implementationen wiederverwendet werden.
- In der Regel entsteht eine mächtigere Sprache als bei der Neuentwicklung, da viele Funktionalitäten bereits in der Basissprache enthalten sind.
- Die Infrastruktur der Basissprache kann ebenso wiederverwendet werden, z. B. Entwicklungsumgebungen mit Editoren, Debuggern, etc. Allerdings unterstützen diese Werkzeuge die interne DSL nur zu einem bestimmten Grad.

- Die Trainingskosten für den professionellen Entwickler sind vergleichsweise gering, da viele von ihnen die Basissprache bereits kennen.

Als Nachteile können genannt werden:

- Die Syntax ist oftmals nicht optimal, da die meisten Sprachen keine willkürliche Spracherweiterungen erlauben.
- Das Überladen von existierenden Operatoren kann verwirren, wenn die neue Semantik andere Eigenschaften besitzt als die ursprüngliche.
- Es können nur dürftige Fehlererkennungen angeboten werden, da die Fehlermeldungen in Konzepten der Basissprache ausgegebenen werden anstatt in DSL-Konzepten.
- Domänenspezifische Optimierung und Transformation sind nur mühsam zu implementieren, so dass potentiell die Effizienz beeinflusst wird, insbesondere bei der Einbettung in funktionale Sprachen [Kam98, Slo02].

Die eben aufgelisteten zentralen Vor- und Nachteile sind unterschiedlich stark bei den einzelnen Implementierungsmethoden ausgeprägt. Beispielsweise sind die Entwicklungs-, Implementierungs- und Wartungskosten bei der Einbettung deutlich geringer als bei der Erweiterung von Compilern oder bei Präprozessoren. Soll eine DSL neu entwickelt werden, sollte deshalb vor allem die Einbettung der DSL in eine Basissprache in Betracht gezogen werden. Werden jedoch domänenspezifische Notationen, Fehlermeldungen etc. verlangt, da z. B. die Anwendergruppe sehr groß ist, ist die Einbettung aufgrund ihrer Einschränkungen weniger geeignet. Die Erweiterung von Interpretern oder Compilern sowie der Präprozessoren bieten hier mehr Möglichkeiten. Eine exakte Ableitung, wann eine Implementierungsmethode verwendet werden darf und wann nicht, lässt sich allerdings nicht pauschal herleiten, da in der Praxis viele weitere Faktoren eine Rolle spielen, wie z. B. die spezifischen Eigenschaften einer Domäne oder die Erfahrungen der Entwickler.

3.4 Einordnung der Arbeit in die Verarbeitung von XML-Dokumenten

Diese Arbeit widmet sich der Weiterentwicklung von bestehenden spezifischen XML-Sprachen. Spezifische XML-Sprachen besitzen eine eigene Syntax, die sich an der Taxonomie innerhalb der XML-Domäne orientiert. Sie sind eigenständige Sprachen, die nicht wie APIs von einer Wirtssprache abhängen. Nichtsdestotrotz werden i. d. R. neben dem Zugriff auf der Ebene der Kommandozeile entsprechende Schnittstellen bereitgestellt, die es universellen Programmiersprachen ermöglichen, DSLs zu integrieren bzw. aufzurufen und die Ergebnisse innerhalb dieser weiter zu verarbeiten. So existieren bei verschiedenen spezifischen XML-Sprachen (z. B. XSLT) Schnittstellen, denen bspw. ein DOM übergeben wird und die als Ergebnis den mit den spezifischen Konstrukten der speziellen Sprache transformierten DOM zurückliefern. Eine standardisierte Schnittstelle zur Einbindung von spezifischen XML-Sprachen ist bspw. in Java das *Transformation API for XML* (TrAX).

Wie im Abschnitt 3.3.3 diskutiert, befriedigt eine ideale und sinnvolle, externe DSL die Anforderungen einer bestimmten Domäne gezielter und besser. Durch geeignete Abstraktion können auch Domänenexperten die DSL-Programme selbst verstehen, überprüfen, ändern oder schreiben. Gerade deswegen besteht bei DSLs die Notwendigkeit weniger umfangreich als universelle Programmiersprachen zu sein. Die Eigenentwicklung und Anpassung gestaltet sich somit als Herausforderung, da gerade die Schwierigkeit besteht, den Problemraum exakt abzugrenzen und zu definieren. Er darf nicht zu groß aber auch nicht zu klein gewählt werden. Ähnlich verhält es sich mit den Sprachkonstrukten zur Lösung des definierten Problemraums. Hier muss die richtige Balance zwischen domänenspezifischen und universellen Sprachkonstrukten gefunden werden.

Gerade hier liegt das gegenwärtige Problem von spezifischen XML-Transformationssprachen. Sie stellen keine sprachlichen Mittel bereit, um angepasst oder weiterentwickelt zu werden. Entsprechend sind proprietäre, über den Standard hinausgehende Erweiterungen entstanden. Durch den Eingriff in die Sprachdefinition müssen Sprachprozessoren jedoch modifiziert werden. Für die notwendigen Modifikationen entstehen unverhältnismäßig hohe Kosten. Aus diesem Grund werden diese Spracherweiterungen nur teilweise oder gar nicht von Sprachprozessoren unterstützt werden.

Das im nächsten Teil der Arbeit vorgestellte Framework soll die technische Grundlagen bilden, um spezifische XML-Transformationssprachen mit geringerem Aufwand den gegebenen Bedürfnissen sukzessive und flexibel anpassen zu können.

Teil II

Framework

Der zweiten Teil stellt den Schwerpunkt dieser Arbeit dar. Zunächst werden anhand von zwei typischen Anwendungsszenarien die Probleme zweier konkreter Transformationssprachen exemplarisch gezeigt. Es wird die Lösungsidee skizziert und es werden Anforderungen an die zu entwickelnde Systemarchitektur aufgestellt. Anschließend werden die Konzepte und die Umsetzungen sowohl der Plattform als auch der Generatoren beschrieben. Den Abschluss bilden Vorgehensmodelle, die einen organisatorischen Rahmen geben.

Kapitel 4

Konzept

Eigenständige XML-Transformationssprachen sind auf die Domäne der XML-Transformationen spezialisiert. Sie stellen abhängig von der verwendeten Transformationsmethode bestimmte Operatoren bereit, mit denen XML-Dokumente verarbeitet werden können. Um ein möglichst großes Spektrum der XML-Transformationsdomäne abzudecken, sind die bereitgestellten Operatoren bzgl. dieser (horizontalen) Domäne universell einsetzbar. So werden bspw. Operatoren angeboten, um Elemente oder Attribute von XML-Dokumenten zu erzeugen. Angemessene Operatoren für eine eingegrenzte Transformationsaufgabe, z. B. für eine vorher festgelegte Markup-Sprache oder eine nicht XML-basierte Sprache als Zielsprache, fehlen i. d. R.. Hierdurch entstehen zum einen vergleichsweise umfangreiche Beschreibungen. Zum anderen erlauben die elementaren Operatoren keine geeigneten Fehlererkennungsmechanismen mit adäquaten Fehlermeldungen.

Um die Probleme von bisherigen spezifischen XML-Transformationssprachen anschaulich zu verdeutlichen, werden zwei konkrete Anwendungsszenarien herangezogen. Innerhalb der Beispiele werden erste konzeptionelle Lösungsideen angedeutet. Die Analyse der Ideen führt zur Aufstellung von Anforderungen. Die einzelnen Module des angestrebten Frameworks werden aus den Anforderungen abgeleitet und in eine Systemstruktur integriert.

4.1 Anwendungsszenarien

Die beiden Szenarien werden bewusst gleich auf mehreren Ebenen konträr gewählt. Zum einen gehören die Anwendungsszenarien zu den typischen Aufgabenfeldern der XML-Verarbeitung. Das erste Szenario demonstriert eine Umstrukturierung von XML-Daten mit der Sprache XUpdate. Das zweite Szenario überführt einen XML-Datensatz in ein Darstellungsformat mit Hilfe von XSLT. Für die Umsetzung werden zwei unterschiedliche Transformationsmethoden – Änderung vs. suchmustergesteuerter Template-Ansatz – eingesetzt (siehe Abschnitt 3.2.2.9). Die dafür genutzten Sprachen unterscheiden sich neben der Transformationsmethode ebenso hinsichtlich ihrer Entwicklungshistorie. XUpdate wurde von einer kleinen Entwicklergruppe entworfen und dessen Spezifikation muss als rudimentär bezeichnet werden (vgl. [Ogb05]). Im Gegensatz dazu wurde XSLT von einem großen Konsortium nach einem vergleichsweise langen Standardisierungsprozess veröffentlicht. Als Beispieldaten dienen für beide Szenarien die folgende Liste von Organisationen:

```
1  <organizations>
2    <organization id="CompetenceCenter">
3      <name>Competence Center</name>
4      <description>A first sample supplying organization</description>
5      <status>online</status>
6    </organization>
7    <organization id="CorporateExpress">
8      <name>Corporate Express</name>
9      <description>A second sample supplying organization</description>
10     <status>online</status>
11   </organization>
12   <organization id="PizzaExpress">
13     <name>Pizza Express</name>
14     <description>A third sample supplying organization</description>
15     <status>online</status>
16   </organization>
17 </organizations>
```

CODE 4.1: Liste von Organisationen.

4.1.1 Umstrukturierung

Für viele Applikationen ist es notwendig, XML-Daten nach verschiedenen Kriterien zu sortieren, zu gruppieren oder nach anderen Gesichtspunkten umzustrukturieren, sei es bspw. zum Zwecke des Datenaustausches oder der Migration von Daten. Für diese Aufgabe bieten sich Änderungssprachen an, da sie spezielle Operatoren für eine Umstrukturierung eines XML-Dokumentes bereitstellen.

XUpdate (*XML Update Language*) [LM00] ist ein Vertreter dieser Kategorie (siehe Abschnitt 3.2.2.9). Die aktuelle Spezifikation von XUpdate befindet sich im Entwicklungsstadium [Ogb05]. Die Änderungssprache wird dennoch von einigen nativen XML-Datenbanken unterstützt, wie z. B. Apache XIndice [XIn07], eXist [eXi08] oder X-Hive [X-H07]. XUpdate nutzt eine interne Baumdarstellung als Verarbeitungsmodell (vgl. Abschnitt 3.1.3) und bietet verschiedene Operatoren zur Manipulation dieses Baums an. Als Abfragesprache innerhalb der Operatoren wird XPath 1.0 [CD99] verwendet. Die wichtigsten Operatoren sind:

- `modifications`
  Wurzelelement und Container für alle anderen Operatoren
- `element`, `attribute`, `text`, `processing-instruction`, `comment`
  Knotenarten
- `insert-before`, `insert-after`
  Einfügen eines Knotens vor oder nach einem selektierten Elementknoten
- `append`
  Anhängen eines Knotens an einen selektierten Elementknoten
- `update`
  Änderung des Inhaltes eines selektierten Knotens

- `remove`
  Löschen eines selektierten Knotens und dessen Teilbaum
- `rename`
  Umbenennen eines selektierten Knotens
- `variable`
  Deklaration von Variablen
- `value-of`
  Erzeugung von Teilbäumen
- `if`
  Definition von Entscheidungen

Mit diesen Operatoren können die wesentlichen Änderungsoperationen auf der internen Datenstruktur durchgeführt werden. Da diese Grundoperatoren nahezu atomar sind (der `rename`-Operator könnte z. B. durch eine Kombination aus `insert-before`-, `element`-, `value-of`- und `remove`-Operatoren simuliert werden), ist die Beschreibung einer Umstrukturierung sehr feingranular. Beispielsweise erweist sich das einfache Verschieben oder Austauschen von Teilbäumen als nichttriviale Beschreibung, wie bereits in der Mailingliste[1] festgestellt wurde.

Verschieben von Teilbäumen

Der Beispielcode 4.2 zeigt die Umsetzung eines einfachen Verschiebens mit XUpdate. Es soll die Organisation mit der ID `CompetenceCenter` aus Code 4.1 und deren Kindknoten hinter die Organisation `CorporateExpress` geschoben werden.

```
<modifications version="1.0" xmlns="http://www.xmldb.org/xupdate">          1
  <if test="//organization[@id='CompetenceCenter']">                        2
    <variable name="INTERMEDIATE-MOVE-id01" select="//organization[@id='CompetenceCenter']"/>   3
    <remove select="//organization[@id='CompetenceCenter']"/>               4
    <insert-after select="//organization[@id='CorporateExpress']">          5
      <value-of select="$INTERMEDIATE-MOVE-id01"/>                          6
    </insert-after>                                                         7
  </if>                                                                     8
</modifications>                                                            9
```

CODE 4.2: Verschieben von Teilbäumen mit XUpdate.

Als erster Schritt muss überprüft werden, ob sich überhaupt eine Organisation mit der ID `CompetenceCenter` in der Datenstruktur befindet (Zeile 2). Existiert eine solche Organisation, muss deren gesamter Teilbaum temporär gespeichert werden (Zeile 3). Nun kann sie aus der Datenstruktur gelöscht werden (Zeile 4). Abschließend wird die temporär gespeicherte Organisation an der gewünschten Stelle wieder eingefügt (Zeilen 5–7).

Da das Verschieben von Teilbäumen eine häufig auftretende Operation bei Umstrukturierungen ist, wäre die Einführung eines eigenen Operators sinnvoll. Der neue Operator könnte bspw.

[1] `http://www.mail-archive.com/xupdate-dev@xmldb.org/msg00075.html`

`move-after` heißen und zwei notwendige Attribute besitzen. Eines beschreibt, welche Teilbäume ausgehend von dessen Wurzelelement verschoben werden sollen (`from`), und das andere, hinter welche Elemente die selektierten Teilbäume geschoben werden sollen (`to`). Die Werte der Attribute enthalten entsprechende XPath-Ausdrücke. Anstatt der Zeilen 2–8 des Beispielcodes 4.2 müsste nach der Einführung des neuen Operators lediglich geschrieben werden:

```
<xupe:move-after
   from="//organization[@id='CompetenceCenter']"
   to="//organization[@id='CorporateExpress']"/>
```

Austauschen von Teilbäumen

Auf vergleichbare Weise kann das etwas komplexere Austauschen von zwei Teilbäumen in XUpdate (siehe Beispielcode 4.3) durch einen eigenen kurzen und prägnanten Operator beschrieben werden. Beim Austauschen zweier Teilbäume müssen diese zunächst zwischengespeichert werden (Zeilen 4–7). Danach erfolgt eine Umbenennung des Wurzelknotens dieser Teilbäume und das Einfügen der zwischengespeicherten Teilbäume an die entsprechenden Stellen (Zeilen 8–15). Anschließend werden die Teilbäume mit umbenannten Wurzelknoten gelöscht (Zeilen 16–17). Der Zwischenschritt der Umbenennung ist notwendig, um ursprüngliche und neu eingefügte Teilbäume voneinander unterscheiden zu können und somit mögliche inkorrekte Einfügungen zu vermeiden.

```
1  <modifications version="1.0" xmlns="http://www.xmldb.org/xupdate">
2    <if test="boolean(//organization[@id='CompetenceCenter']) and
3              boolean(//organization[@id='CorporateExpress'])">
4      <variable name="INTERMEDIATE-FIRST-id02"
5                select="//organization[@id='CompetenceCenter']"/>
6      <variable name="INTERMEDIATE-SECOND-id02"
7                select="//organization[@id='CompetenceExpress']"/>
8      <rename select="//organization[@id='CompetenceCenter']">SUBTREES-FIRST</rename>
9      <rename select="//organization[@id='CorporateExpress']">SUBTREES-SECOND</rename>
10     <insert-before select="//SUBTREES-FIRST">
11       <value-of select="$INTERMEDIATE-SECOND-id02"/>
12     </insert-before>
13     <insert-before select="//SUBTREES-SECOND">
14       <value-of select="$INTERMEDIATE-FIRST-id02"/>
15     </insert-before>
16     <remove select="//SUBTREES-FIRST"/>
17     <remove select="//SUBTREES-SECOND"/>
18   </if>
19 </modifications>
```

CODE 4.3: Austauschen von Teilbäumen mit XUpdate.

Der neue Austauschoperator könnte bspw. folgende simple Notation besitzen:

```
<xupe:swap
    first ="//organization[@id='CompetenceCenter']"
    second="//organization[@id='CorporateExpress']"/>
```

Die zwingend erforderlichen Attribute enthalten XPath-Ausdrücke, die die Wurzelelemente der Teilbäume selektieren, die ausgetauscht werden sollen.

4.1.2 Überführung in ein Darstellungsformat

Eine andere typische Verarbeitung für in XML gespeicherte Daten ist die Überführung in ein Darstellungsformat. Dazu werden die XML-Daten mit Darstellungsinformationen angereichert. So können XML-Daten in XHTML oder XSLFO transformiert oder aber auch durch Diagramme in SVG visualisiert werden.

Exemplarisch soll aus den Organisationsdaten (Code 4.1) eine Repräsentation in XHTML erzeugt werden. Da XHTML ebenso ein XML-Format ist, könnte auch hier eine Änderungssprache, wie XUpdate, zur Beschreibung der Überführung eingesetzt werden. Jedoch weist XHTML eine komplett andere Struktur auf. Ein Transformationsansatz, bei dem das Zieldokument vollständig neu aufgebaut wird, ist demnach zu bevorzugen. Eine bedeutende und in der Praxis weit verbreitete Transformationssprache ist XSLT. XSLT soll für dieses Szenario deshalb als Beispielsprache dienen.

XSLT entstand während des Standardisierungsprozesses beim W3C von XSL (*Extensible Stylesheet Language*). XSL ist eine Sprachfamilie, die ursprünglich allein mit dem Ziel entworfen wurde, Sprachen zur Verfügung zu stellen, die aus XML-Dokumenten ein Layout erzeugen können. Die Sprachfamilie besteht aus folgenden drei Sprachen:

- XSLFO (*XSL Formatting Objects*): ein spezielles layoutorientiertes XML-Vokabular zur Formatierung [SBC+01]
- XSLT (*XSL Transformations*): eine spezielle XML-Sprache zur Transformation von XML-Dokumenten [Cla99a]
- XPath (*XSL Path Language*): eine Abfragesprache, die XSLT benutzt, um auf Teile eines XML-Dokumentes zuzugreifen oder zu verweisen [CD99]

Die beiden erstgenannten Sprachen, XSLFO und XSLT, werden selbst in XML ausgedrückt. XPath verwendet hingegen eine kompakte Nicht-XML-Syntax. Jede dieser Spezifikationen kann unabhängig voneinander gebraucht werden. XPath bspw. kommt sowohl in anderen Sprachen des W3Cs (z. B. XML Schema [FW04] oder XQuery [Sco07]) als auch in unabhängigen Sprachen (z. B. XUpdate [LM00]) zum Einsatz.

Bei XSLT handelt es sich um eine Transformationssprache, die jede als XML vorliegende Eingabe in eine beliebig andere Ausgabe, sowohl XML-Syntax als auch Nicht-XML-Syntax, überführen kann. Für die Transformation in XML-Dokumente stellt XSLT entsprechende Operatoren zur Erzeugung von Elementen, Attributen, Kommentaren, usw. bereit. Für eine Nicht-XML-Syntax werden keine speziellen Operatoren angeboten. Wie bei Template-basierten Sprachen üblich (siehe Abschnitt 3.2.2.9), kann dagegen direkt die konkrete Syntax der Zielsprache in einem Template genutzt werden, insofern keine reservierten Schlüsselwörter sowie Trennungszeichen von XML sowie XSLT darin auftreten. Um eine falsche Interpretation zu vermeiden, müssen diese maskiert werden. Die konkrete Syntax kann ebenso für XML-Zieldokumente verwendet werden.

```
1  <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
2
3    <xsl:output method="xml" indent="yes"
4      doctype-public="-//W3C//DTD XHTML 1.0 Transitional//EN"
5      doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"/>
6    <xsl:strip-space elements="*"/>
7
8    <xsl:template match="organizations">
9      <html xmlns="http://www.w3.org/1999/xhtml">
10       <head>
11         <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
12         <link rel="stylesheet" type="text/css" href="style.css"/>
13         <title>Backoffice</title>
14       </head>
15       <body>
16         <table width="100%" border="0" cellspacing="0" cellpadding="0" height="100">
17           <tr align="middle" valign="center">
18             <td class="table_title n e s w" colspan="4">Supplying Organizations</td>
19           </tr>
20           <tr class="table_header">
21             <td class="w s" nowrap="nowrap" width="15">No</td>
22             <td class="w s" nowrap="nowrap">Name</td>
23             <td class="w s" nowrap="nowrap">Description</td>
24             <td class="w e s" nowrap="nowrap">Status</td>
25           </tr>
26           <xsl:for-each select="organization">
27           <tr class="table_detail">
28             <td class="w s center" nowrap="nowrap"><xsl:value-of select="position()"/></td>
29             <td class="w s" nowrap="nowrap"><xsl:value-of select="name"/></td>
30             <td class="w s"><xsl:value-of select="description"/></td>
31             <td class="w e s" nowrap="nowrap"><xsl:value-of select="status"/></td>
32           </tr>
33           </xsl:for-each>
34         </table>
35       </body>
36     </html>
37   </xsl:template>
38
39 </xsl:stylesheet>
```

Code 4.4: Erzeugen von XHTML mit XSLT.

Der Code 4.4 zeigt ein XSLT-Programm (aus historischen Gründen auch Stylesheet genannt), das aus der Organisationsliste (Code 4.1) eine entsprechende XHTML-Datei erzeugt. Es besteht aus lediglich einem Template. Die Anweisungen innerhalb des Templates werden ausgeführt, sobald beim Durchlaufen des Quelldokumentes auf das Element `organizations` gestoßen wird. Ist diese Bedingung erfüllt, werden die XHTML-Literale mit den Stilvorgaben sowie einige Daten, die iteriert aus dem Quelldokument gelesen werden (Zeilen 27–34), in die Zieldatei geschrieben.

Abbildung 4.1 präsentiert das Aussehen der von XSLT erzeugten Datei im Web-Browser Internet Explorer unter der Verwendung des CSSs von Code 4.5. Obwohl die erzeugte Datei angezeigt werden kann, ist sie bzgl. XHTML nicht korrekt. Wird sie gegen eine Grammatik (in DTD oder XML Schema) geprüft, für XHTML kann man den offiziellen Validierungsdienst[2] nutzen, so kann festgestellt werden, dass das Attribut `height` in dem Element `table` und die Attributwerte

[2] http://validator.w3.org/

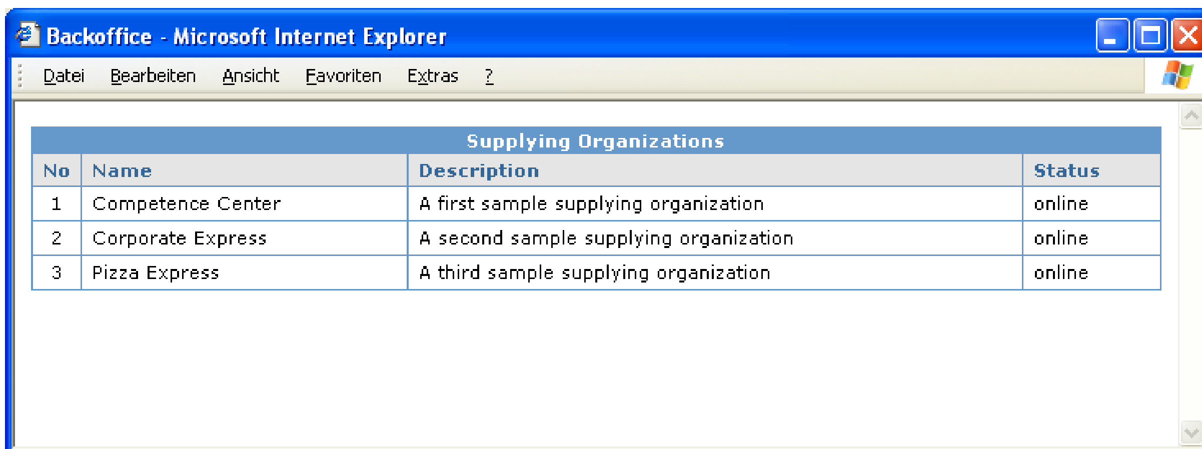


ABBILDUNG 4.1: Erzeugte XHTML-Datei im Browser.

`middle` und `center` für die Attribute `align` und `valign` im Element `tr` nicht erlaubt sind. Das heißt, obwohl XSLT XML-spezifische Sprachkonstrukte bereitstellt, die die XML-Verarbeitung vereinfachen und es erlauben, jegliches XML-Vokabular zu erzeugen, werden keine Sprachkonstrukte für eine bestimmte Klasse von Zieldokumenten angeboten. Sei es XML-Syntax, wie z. B. XHTML, oder auch Nicht-XML-Syntax, wie z. B. LaTeX oder ein anderes Darstellungsformat. Kann bei XML-Dokumenten noch die Wohlgeformtheit des Transformationsergebnisses gesichert werden, so werden die Restriktionen von nicht-XML-basierter Syntax gänzlich ignoriert. Dies macht XSLT zu einer vielseitig anwendbaren aber zugleich fehleranfälligen Sprache.

```
.center {text-align:center}                                                    1
                                                                               2
.n {border-top: 1px solid #6699cb;}                                            3
.e {border-right: 1px solid #6699cb;}                                          4
.s {border-bottom: 1px solid #6699cb;}                                         5
.w {border-left: 1px solid #6699cb;}                                           6
                                                                               7
. table_title {font-size: 11px;font-weight: bold;color: #FFFFFF;background-color: #6699CB;   8
 padding-right: 5px;padding-left: 5px;height: 17px;}                           9
.table_header {font-size: 11px;font-weight: bold;color: #336699;background-color: #e5e5e5;   10
 padding-right: 6px;padding-left: 6px;height: 20px;}                           11
. table_detail {font-size: 11px;color: #000000;background-color: #FFFFFF;      12
 padding-right: 6px;padding-left: 6px;padding-top: 3px;padding-bottom: 3px;height: 18px;}   13
                                                                               14
body {font-family: Verdana, Geneva, Arial, Helvetica, Sans-Serif}              15
```

CODE 4.5: CSS für XHTML.

An dieser Stelle wäre eine Erweiterung von XSLT durch zielsprachenspezifischen Operatoren denkbar. Beispielsweise könnte ein XHTML-spezifischer Operator die selbe Elementdeklaration besitzen, wie sie durch die XHTML-Spezifikation [PAA+02] bzw. durch die zugehörige transitionale DTD oder XML-Schema [Mas02] vorgegeben ist. Ein `tr`-Operator würde dann aus allgemeinen Universalattributen (`id`, `class`, `style` und `title`), Ereignisattributen (`onclick`,

`ondblclick`, `onmousedown`, usw.) sowie Attributen zur Internationalisierung (`dir` und `lang`) bestehen. Dazu kommen spezielle Attribute `align` mit `char` und `charoffset` sowie `valign` zur Textausrichtung in den Zellen einer Tabellenzeile und das Attribut `bgcolor` für die Definitionen der Hintergrundfarbe. Die Attribute `char` und `charoffset` sind dabei nur sinnvoll, wenn das Attribut `align` den Wert `char` besitzt. Durch die Einführung eines solchen zusätzlichen zielsprachenspezifischen `tr`-Operators könnte gesichert werden, dass das erzeugte XHTML-Element `tr` keine falschen Attribute oder Attributwerte enthält.

4.2 Vision

Die beiden Anwendungsszenarien verdeutlichen trotz ihrer Unterschiede den Bedarf, existierende Sprachen zu erweitern. Beim ersten werden die bestehenden Sprachkonstrukte zur Lösung des definierten Problemraums durch neue Sprachkonstrukte ergänzt. Durch die Zusammenfassung von immer wieder auftretenden Operatorkombinationen zu einem prägnanten Operator wird der Code kürzer, lesbarer und besser wartbar. Beim zweiten wird der Problemraum weiter eingegrenzt. Dadurch besteht die Möglichkeit problemangepasste Operatoren zu definieren, die diese Aufgabe gezielter unterstützen, z. B. durch adäquate Fehlermeldungen.

Existierende XML-Transformationssprachen sind monolithische Sprachen. Sie bieten keine Mittel der Metaprogrammierung, um weiterentwickelt zu werden. Das Hauptziel dieser Arbeit ist der Aufbau eines Frameworks, mit dem es ermöglicht wird, höhere Operatoren zu definieren und zu nutzen (vgl. Abschnitt 1.1). Höhere Operatoren können dazu genutzt werden, um bspw. bisher noch nicht unterstützte Sprachmerkmale einer XML-Transformationssprache zu verwirklichen (siehe Abschnitt 3.2.2). Höhere Operatoren sollen zu Sprachkomponenten zusammengefasst werden, die je nach Transformationsproblem so kombiniert werden können, dass eine höhere, problemspezifische Transformationssprache entsteht. Mit einer solchen modular zusammensetzbaren Transformationssprache sollen folgende Vorteile gegenüber monolithische Transformationssprachen erzielt werden können:

- **Verständlichkeit/Kompaktheit**
  Höhere oder problemspezifische Operatoren können repetitive Operatorfolgen zusammenfassen. Dies führt zu kompakteren Transformationsdefinitionen. Die Namensgebung der Operatoren kann so gewählt werden, dass sie der Aufgabe des Operators in dessen Anwendungsdomäne entspricht. Die Kompaktheit und die direkte Repräsentation erhöhen dabei i. d. R. die Lesbarkeit und Verständlichkeit.
- **Prüfbarkeit**
  Die direkte Verkörperung von Transformationsschritten in höhere Operatoren anstatt in vielen elementaren Transformationsoperatoren hat ebenso positive Auswirkungen auf die Prüfbarkeit der Transformationsdefinitionen. Die Überprüfung kann auf einer abstrakteren Ebene vorgenommen werden. Insbesondere bei zielsprachenspezifischen Operatoren wirkt sich die Prüfbarkeit positiv auf die Richtigkeit des Transformationsergebnisses aus.

- **Wiederverwendbarkeit**
  Dadurch, dass Sprachkomponenten modular in verschiedenen Konfigurationen miteinander kombiniert werden können, erhöht sich die Wiederverwendbarkeit einer Sprachkomponente und der darin enthaltenen Operatoren.
- **Adaptivität**
  Kleine Problemstellungen können durch eine kleine Gruppe von notwendigen Sprachkomponenten gelöst werden. Wird das Problem größer, können weitere neue Sprachkomponenten hinzugefügt werden. Modulare Spracherweiterungen vermeiden die Probleme von großen und monolithischen Sprachen (aufwändige Wartbarkeit, etc.; siehe z. B. [DK97]).
- **Agilität**
  Modular zusammensetzbare Transformationssprachen können schneller bzw. in kürzeren Entwicklungszyklen angepasst und erweitert werden als konventionelle, die mit einem abgeschlossenen Compiler implementiert sind. Da in den meisten Fällen ein abgeschlossener Compiler nicht für eine Erweiterung gerüstet bzw. vorbereitet ist, muss eine neue Version einer Sprachspezifikation in einem Schritt vollzogen werden. Die Realisierung in einem Schritt ist ein bedeutender und dadurch langwieriger Schritt, da bspw. Kompatibilitäten zur alten sowie zukünftiger Versionen beachtet werden müssen. Fragwürdige Sprachkonstrukte von festen Sprachen können nicht einfach wieder entfernt werden.

4.3 Anforderungen und Systemeigenschaften

Das grundsätzliche Ziel der Arbeit ist ein technisches Framework zu entwickeln, welches die Definition und Nutzung von höheren Operatoren bzw. Sprachkomponenten erlaubt, um auf Basis von existierenden XML-Transformationssprachen neue Transformationssprachen zu entwerfen. Diese Zielstellung wird durch folgende Teilanforderungen präzisiert und verfeinert.

Anforderung 1: Anwendbarkeit für viele XML-Transformationssprachen

Wie die beiden Anwendungsszenarien in Abschnitt 4.1 demonstriert haben, gibt es in verschiedenen XML-Transformationssprachen Potentiale für höhere Operatoren. Entsprechend soll das Framework eine Vielzahl von XML-Transformationssprachen unterstützen und mit höheren Operatoren erweitern können. Die existierenden Sprachprozessoren sollen dabei weitergenutzt werden können. Dementsprechend darf kein Eingriff in die Sprachdefinition erfolgen. Die Unabhängigkeit von einer Sprache sichert eine breite Anwendbarkeit und erlaubt zugleich eine mögliche Unterstützung von zukünftigen Transformationssprachen.

Anforderung 2: Freiheit bei der Operatordefinition

Es gibt sehr unterschiedliche Motivationen für höhere Operatoren, wie in den vorangegangenen Abschnitten angedeutet wurde. Das Framework soll möglichst wenige Einschränkungen für höhere Operatoren (z. B. bzgl. der Wahl der Namensgebung, des Programmierparadigmas oder

ihrer Aufgaben) machen. Nur so können angemessene Operatoren hinsichtlich bestimmter Kriterien gebildet werden. Eine Freiheit bei der Operatordefinition kann sich u. a. positiv auf die Kompaktheit, Prüfbarkeit und Wiederverwendbarkeit von höheren Operatoren auswirken.

Anforderung 3: Kombinierbarkeit von Sprachkomponenten

Existierende XML-Transformationssprachen haben fest definierte Operatoren, die bzgl. der XML-Transformationsdomäne universell ausgerichtet sind, um für möglichst viele Transformationsprobleme in dieser Domäne anwendbar zu sein. Mit höheren Operatoren können passende problemspezifische Operatoren bspw. für eine bestimmte Zielsprache, wie im zweiten Anwendungsszenario benötigt, hinzugefügt werden. Da jedoch für ein konkretes Anwendungsproblem nicht alle definierten Sprachkomponenten gebraucht werden, soll das Framework problemangepasste Konfigurationen von Sprachkomponenten ermöglichen. Um dies zu erreichen, muss das Framework das modulare Zusammensetzen von Sprachkomponenten erlauben. Das bedeutet u. a., dass sichergestellt sein muss, dass Operatoren trotz gleicher Benennung unterschieden und der jeweiligen Sprachkomponente zugeordnet werden können.

Anforderung 4: Flexible Implementierung

Um die Implementierung von neuen Operatoren weitestgehend zu vereinfachen, soll die Implementierung in zweierlei Hinsicht flexibel sein. Zum einen sollen die neuen Operatoren sowohl auf der Basis von elementaren Operatoren einer Basistransformationssprache als auch auf Basis von bereits neu entwickelten Operatoren implementiert werden können. Zum anderen sollen gerade für diese Überführung unterschiedliche Technologien zur Verfügung stehen, so dass der Implementierer abhängig von der Komplexität der Überführung und seinen Präferenzen eine für ihn optimale Auswahl treffen kann.

Anforderung 5: Generative Unterstützung für die Implementierung

Neben der Flexibilität der Implementierung sollen generative Techniken helfen, den Implementierungsaufwand für die höheren Operatoren weiter zu verringern und die Entwicklungszeit zu verkürzen. Als Grundlage sollen deklarative Strukturbeschreibungen der Operatoren dienen, aus denen schematisch gleicher, technischer Code generiert wird. Das Framework soll entsprechend die Möglichkeit besitzen, solche generative Techniken einzubinden.

Anforderung 6: Anpassbare Fehlermechanismen

Validierungskomponenten, die die Syntax von höheren Operatoren bzgl. ihrer Strukturbeschreibung überprüfen und bei falscher Benutzung Fehlermeldungen ausgeben, sind Voraussetzung für die Nutzbarkeit der Operatoren. Insbesondere, wenn die höheren Operatoren für eine bestimmte Anwenderzielgruppe entwickelt werden, muss die Möglichkeit bestehen, eine dieser Domäne angemessene Fehlermeldung zu spezifizieren. Aber auch der Grad der Fehlererkennung kann je

nach Anwendungsgebiet unterschiedlich sein. Demgemäß soll eine anpassungsfähige Spezifikation der Fehlermeldungen sowie Implementierung von Fehlererkennungsmechanismen unterstützt werden.

Anforderung / **Ziel**	Verständlichkeit/Kompaktheit	Prüfbarkeit	Wiederverwendbarkeit	Adaptivität	Agilität
1. Anwendbarkeit für viele XML-Transformationssprachen	i	i	i	i	i
2. Freiheit bei der Operatordefinition	i	i	i		
3. Kombinierbarkeit von Sprachkomponenten			d	d	i
4. Flexible Implementierung			d		d
5. Generative Techniken für die Implementierung					d
6. Anpassbare Fehlermechanismen	i	d	i		

Legende: d = direkter Zusammenhang, i = indirekter Zusammenhang

Tabelle 4.1: Ziel-/Anforderungsmatrix.

Die aufgestellten Anforderungen an das zu erarbeitende Framework wirken sich unterschiedlich auf die zu erreichenden Ziele (vgl. Abschnitt 4.2) aus. Tabelle 4.1 stellt zusammenfassend dar, welche Anforderungen mit welchen Zielen im direkten oder indirekten Zusammenhang stehen. Das Ziel Verständlichkeit und Kompaktheit von höheren Transformationssprachen kann dabei nicht direkt vom Framework umgesetzt werden. Es kann nur unterstützend die technische Plattform mit möglichst großen Freiheiten bei der Spracherweiterung und mit möglichst großer Flexibilität bei den Fehlermechanismen bereitstellen. Die Verständlichkeit und Kompaktheit selbst kann nur durch den Sprachdesigner erreicht werden, indem er entsprechende höhere Operatoren realisiert. Die Prüfbarkeit kann direkt unterstützt werden, indem Möglichkeiten angeboten werden, den Grad der Fehlererkennungen und Fehlermeldungen kontextspezifisch zu der Sprachkomponente anzupassen. Die Wiederverwendbarkeit kann gleich auf verschiedenen Ebenen gefördert werden. Einerseits auf Sprachentwicklungsebene, indem existierende Sprachkomponenten und deren Operatoren für die Entwicklung anderer Sprachkomponenten wiederverwendet werden können. Andererseits auf Anwendungsebene, indem Sprachkomponenten modular zusammengesetzt werden können. Voraussetzung dafür sind flexible Implementierungstechniken. Durch die Kombinierbarkeit von Sprachkomponenten wird zudem eine wesentlich verbesserte Adaptivität als bei monolithischen Transformationssprachen erzielt. Mit der Offenheit bei der Auswahl von Implementierungstechniken und mit der Einbindung von generativen Techniken wird dass Ziel der kürzeren Entwicklungszyklen (Agilität) erreicht.

4.4 Systemstruktur

Aus den Anforderungen des letzten Abschnittes können notwendige Module des Frameworks abgeleitet werden. Die grundlegende Systemstruktur wird in Abbildung 4.2 gezeigt. Das zentrale Modul ist dabei die Ausführungsumgebung XTC (*XML Transformation Coordinator*), die für die Steuerung des Transformationsprozesses verantwortlich ist. XTC ermöglicht es, verschiedene (Basis-)Transformationssprachen und zusätzliche Bibliotheken anzubinden. Dazu können die entsprechenden Sprachprozessoren über bereitgestellte Schnittstellen in XTC integriert werden. Sämtliche so angebundene Transformationssprachen können einerseits erweitert und andererseits zur Implementierung der Erweiterung eingesetzt werden [FS06]. Auf diese Weise werden die Anforderungen 1 und 4 verwirklicht.

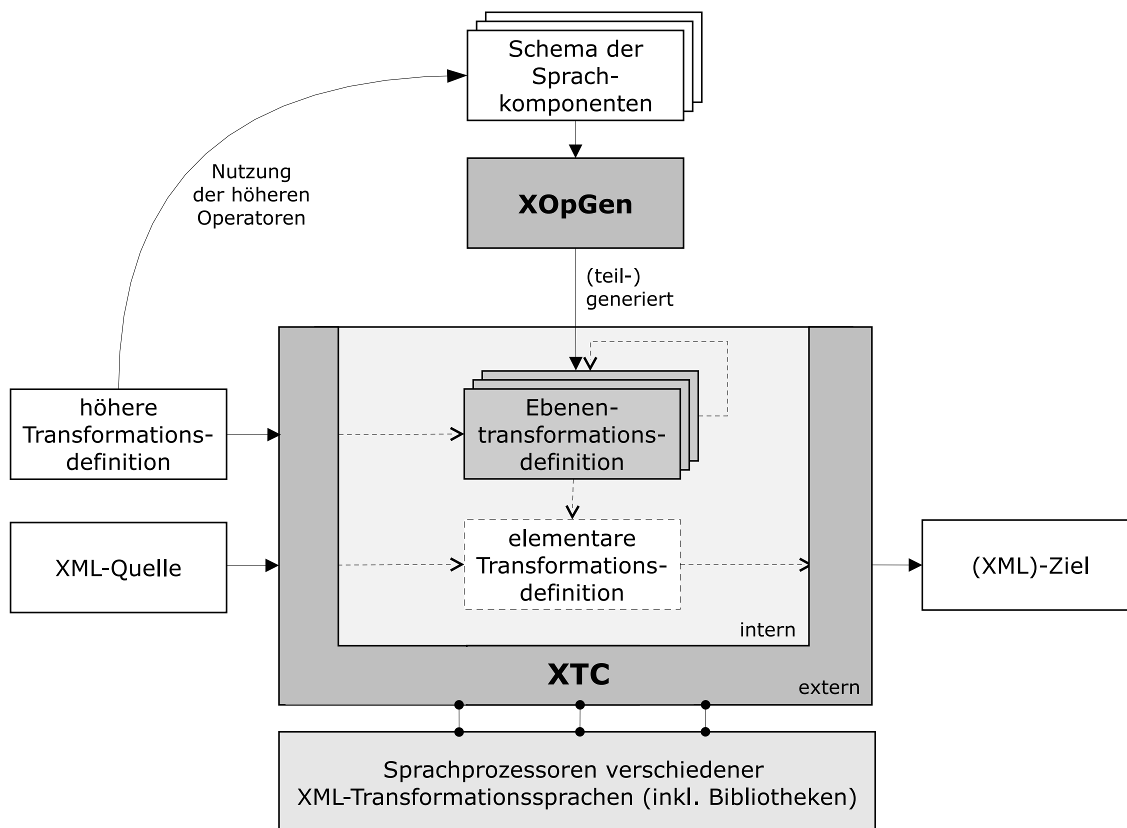


Abbildung 4.2: Struktur des Frameworks.

Die Erweiterung einer Sprache wird mit Hilfe einer oder mehrerer Ebenentransformationen umgesetzt. Die dafür benötigten Ebenentransformationsdefinitionen können in XTC registriert und dadurch eingebunden werden. Eine Ebenentransformationsdefinition kann einfache lokale Ersetzungen von einzelnen Sprachkonstrukten, vergleichbar mit Makros, beschreiben. Durch Ebenentransformationen können aber darüber hinaus auch bspw. verteilte Sprachkonstrukte realisiert werden. Entsprechend reicht das Spektrum von kleinen Erweiterungen einer Basistransformationssprache bis hin zu komplett neuen problemangepassten Transformationssprachen (Anfor-

derung 2). Unabhängig vom Umfang der Ebenentransformationsdefinitionen müssen sie letztendlich elementare Sprachkonstrukte einer Basistransformationssprache und dadurch ausführbare Transformationsdefinitionen erzeugen. Diese Bedingung muss allerdings nicht durch eine Ebenentransformation gesichert sein, sondern kann ebenso durch einen mehrstufigen Ebenentransformationsprozess erfüllt werden.

Durch die Möglichkeit der Hintereinanderausführung von mehreren Ebenentransformationen können zum einen Erweiterungen in Sprachkomponenten modularisiert werden (Anforderung 3). Zum Anderen wird durch die Hintereinanderausführung die Voraussetzung geschaffen, um Sprachkomponenten oder ganze Sprachen auf nichtelementare Sprachkonstrukte, somit auf bereits höhere Sprachkomponenten, aufzubauen und dadurch eine Hierarchie von Sprachkonstrukten zu bilden (Anforderung 4). Das Modul XTC ermöglicht somit den Aufbau einer Operatorhierarchie und deren Anwendung.

Die Implementierung der Ebenentransformationsdefinitionen kann durch das Generatorsystem XOpGen (*XML Operator Generator*) unterstützt werden [Föt07a]. Es generiert auf Basis eines Schemas, welches eine Sprachkomponente oder eine komplette Sprache beschreibt, Validierer (Anforderung 5). Der Validierer überprüft im Voraus, ob die relevanten zusätzlichen Sprachkonstrukte in den höheren Transformationsdefinitionen korrekt bzgl. der Grammatik verwendet wurden. Er ist von der eigentlichen Transformation entkoppelt. Das Generatorsystem wird dabei so offen gestaltet, dass dieses Modul für verschiedene Transformationssprachen einsetzbar ist (Anforderung 1) und kontextspezifische Fehlermeldungen erzeugen kann (Anforderung 6).

Der Aufbau und das Zusammenspiel der einzelnen Module des Frameworks werden in den Kapiteln 5 und 6 vertieft. Kapitel 5 schildert detailliert die notwendigen Aspekte der Operatorhierarchie, um höhere Operatoren aufbauend auf elementaren und nichtelementaren Operatoren zur Verfügung stellen zu können. Kapitel 6 befasst sich mit den generativen Techniken innerhalb des Frameworks. Das Kapitel 7 stellt Vorgehensmodelle vor, welche beschreiben, wie Erweiterungen entwickelt werden können. Im Kapitel 8 wird anschließend exemplarisch die Anwendung des Frameworks unter Einbeziehung der vorgeschlagenen Entwicklungsvorgehen gezeigt.

Kapitel 5

Operatorhierarchie

In diesem Kapitel wird die Ausführungsumgebung XTC vorgestellt. XTC bildet die Infrastruktur, um höhere Transformationsdefinitionen ausführen zu können. Die Grundlage von XTC ist dabei das Konzept der Operatorhierarchie, welches die Definition von höheren auf bereits bestehende Operatoren ermöglicht.

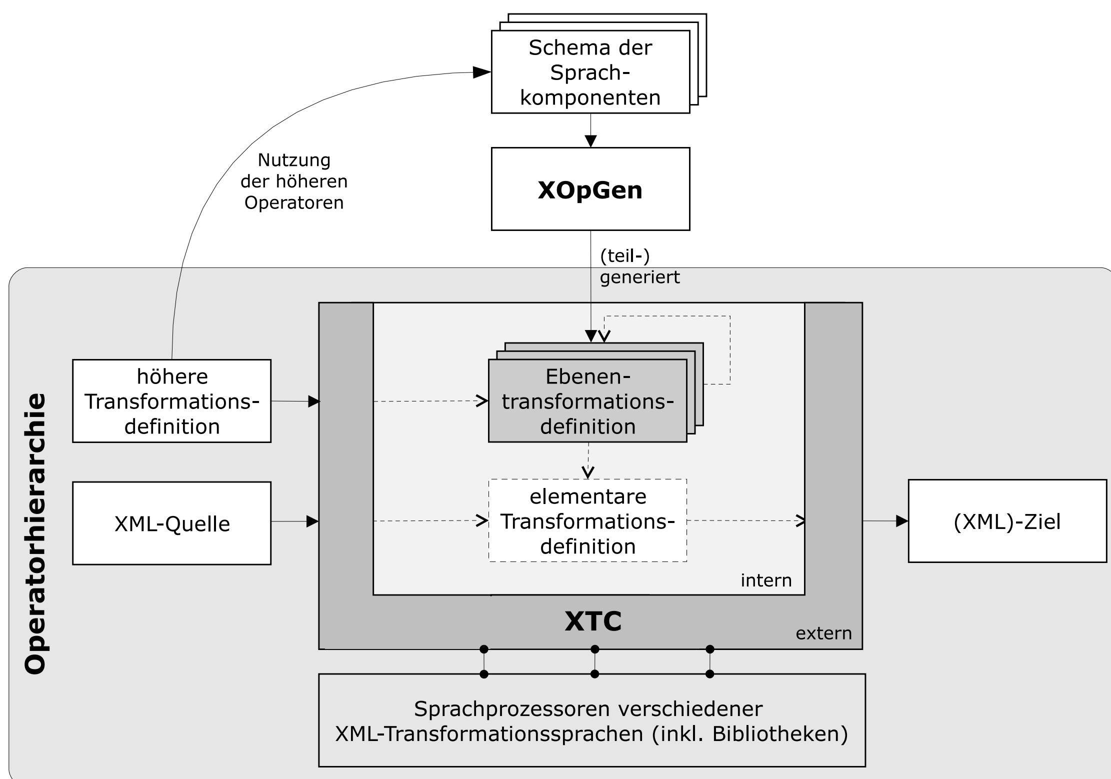


Abbildung 5.1: Einordnung von XTC in die Gesamtstruktur.

Im ersten Abschnitt dieses Kapitels wird in die grundlegenden Ideen der Operatorhierarchie eingeführt. Im nachfolgenden Abschnitt wird auf die für eine Realisierung notwendigen Verarbeitungsschritte detailliert eingegangen. Die Realisierung der Ausführungsumgebung XTC wird im letzten Abschnitt beschrieben.

5.1 Einführung

Bei der Entwicklung von Programmiersprachen im Allgemeinen hat das Prinzip der Hierarchisierung eine besondere Bedeutung. In den Anfängen der Programmierung wurden die Programme in Maschinensprachen, die direkt auf einem Prozessor ausgeführt werden konnten, geschrieben. Diese Form der Programmierung wurde schnell durch Assemblersprachen, die die ausführbaren binären Zahlencodes durch symbolische Zeichen ersetzten, abgelöst. Da Assemblersprachen den Maschinencode eines spezifischen Computers größtenteils reflektieren, werden sie oftmals auch als maschinennahe Sprachen bezeichnet. Die notwendige Übersetzung von Assemblersprache in Maschinensprache übernimmt der sogenannte Assembler.

Höhere Programmiersprachen, die auf der Basis niedrigerer konstruiert werden, sind abstrakter und dadurch weitgehend maschinenunabhängig. Die Programme sind kompakter und verständlicher, da der Entwickler kein detailliertes Wissen über die interne Verarbeitung eines Computers haben muss. Nichtsdestotrotz, muss jedes Programm in einer höheren Programmiersprache indirekt über die Assemblersprache oder direkt in Maschinensprache transformiert werden, um es ausführen zu können. Diese Hierarchie von Programmiersprachen kann sich fortsetzen. So werden einige moderne Programmiersprachen heute erst in weniger hohe Sprachen überführt, aus denen selbst wiederum indirekt Maschinencode gewonnen werden kann. Programme, mit denen die Übersetzungen kompletter Sprachen ausgeführt werden, werden als Compiler bezeichnet. Daneben kann ein Programm in einer höheren Programmiersprache auch interpretiert werden. Dabei wird das Programm nicht vorab in Maschinencode übersetzt, sondern während seiner Laufzeit führt ein Interpreter die Anweisungen aus. Die entstandene Hierarchie von Programmiersprachen folgt nicht dem Metaisierungsprinzip (vgl. Abschnitt 2.5). Eine höhere Programmiersprache ist keine Metasprache einer niedrigeren, sondern vielmehr eine Abstraktion dieser.

Auf einem zu der oben beschriebenen Sprachhierarchie vergleichbaren, aber feingranulareren Konzept basiert die Operatorhierarchie. Jedoch werden keine Ebenen von Sprachen, sondern von Transformationsoperatoren basierend auf einer Basistransformationssprache gebildet. Eine festgelegte Menge von (neuen) Transformationsoperatoren, mglw. auch unterschiedlicher Ebenen, ergibt ein erweitertes oder eine komplett neues Vokabular einer höheren Transformationssprache. Um die zusätzlichen höheren Transformationsoperatoren ausführen zu können, müssen diese in einem Präprozess in Operatoren der Basissprache transformiert werden. Dieser Präprozess ist aufgrund der Hierarchie von Operatoren mehrstufig. Es können, im Gegensatz zu Makrosprachen, unterschiedliche Implementierungstechniken zur Überführung der Operatoren eingesetzt werden.

Ein **Operator** bezeichnet in dieser Arbeit ein Sprachkonstrukt einer Transformationssprache, der in einer bestimmten Weise auf seine Umwelt wirkt und diese verändert (Funktion des Operators). Der Begriff Operator schließt neben der Operatordefinition (die Syntax des Operators) ebenso dessen Implementierung (codierte Semantik des Operators) ein. Das **Pivot** soll in An-

lehnung an [BHW97][1] das zentrale Element eines Operators sein. In aller Regel ist dies das Wurzelelement des Operators, welches für die Umsetzung eines Operators eine wesentliche Rolle spielt.

Neben den funktionalen Eigenschaften eines Operators (Kontrolloperator, Erzeugungsoperator, usw.) können Operatoren mit folgenden, für diese Arbeit wesentlichen nichtfunktionalen Eigenschaften charakterisiert werden (in Anlehnung an [Dör94]):

- **Breite des Wirkungsspektrums**
  Ein Operator weist ein breiteres Wirkungsspektrum als ein anderer Operator auf, wenn er auf mehr Merkmale eines Sachverhaltes in seiner Umwelt verändernd wirkt als der andere Operator.
- **Größe des Anwendungsbereiches**
  Ein Anwendungsbereich eines Operators ist umso größer, je weniger Bedingungen an seine Anwendung geknüpft sind. Wird die Anwendung an eine Vielzahl von Bedingungen geknüpft, so ist der Anwendungsbereich eher klein.
- **Höhe der Ausführungskosten**
  Die Ausführung eines Operators verursacht zeitliche und auch andere Kosten, die unterschiedlich hoch sein können und in manchen Anwendungsfällen berücksichtigt werden müssen.

Sowohl funktionale als auch nichtfunktionale Eigenschaften beeinflussen die Anwendbarkeit bzw. Wiederverwendbarkeit von Operatoren.

Gemäß seines Aufbaus kann ein Operator zudem mit strukturellen Eigenschaften charakterisiert werden. Da der Fokus in dieser Arbeit auf den XML-basierten Transformationssprachen liegt, werden die Operatoren selbst durch XML-Strukturen mit entsprechenden Elementen und Attributen formuliert. Tabelle 5.1 fasst einige der möglichen strukturellen Unterscheidungsmerkmale zusammen. In Tabelle 5.2 erfolgt exemplarisch eine strukturelle Charakterisierung an fünf Beispieloperatoren der Transformationssprache XSLT.

Das Konzept der Operatorhierarchie basiert auf der Idee, eine Schichtenarchitektur aufzubauen, bei der neue höhere Operatoren auf der Grundlage existierender Operatoren hinzugefügt werden können, ohne jedoch die Basistransformationssprache zu verändern [FSH05]. Dies erlaubt eine sukzessive Bottom-up-Komposition und -Adaption von Transformationsoperatoren.

Ausgangspunkt der Hierarchie ist die unterste Ebene von Operatoren. Diese Ebene bildet das Fundament der Hierarchie. Sie enthält elementare Operatoren, die von einer Basistransformationssprache angeboten werden. Dies könnte bspw. die Transformationssprache XSLT sein. **Elementare Operatoren** bezeichnen diejenigen Operatoren, die grundlegende Transformationsoperationen auf einer Datenstruktur ausführen sowie zur Steuerung des Kontrollflusses notwendig sind. Aufbauend auf dieser niedrigsten Ebene können in einer Operatorhierarchie neue abstraktere Ebenen eingeführt werden, die bspw. in den Konzepten einer eingegrenzten

[1] Boyle et al. führten den Begriff Pivot für die Beschreibung von Transformationen im Transformationssystem Tampr ein.

Unterscheidungsmerkmal	Ausprägungen	Beschreibung
Attributierung	attributiert, attributlos, hybrid	Laut Sprachdefinition muss ein Operator (ein oder mehrere) Attribute besitzen oder nicht. Einige Operatoren können sowohl mit als auch gänzlich ohne Attribute auskommen.
Verschachtelung	unverschachtelt, verschachtelt, hybrid	Laut Sprachdefinition besteht ein Operator lediglich aus einem Element (Pivot) oder er setzt sich aus weiteren Kindelementen und mglw. tiefer verschachtelten Elementen zusammen.
Inhalt	leer, nichtleer, hybrid	Laut Sprachdefinition besitzt ein Operator neben den mglw. zum Operator gehörenden Elementen andere Operatoren bzw. textuellen Inhalt oder dies ist nicht möglich. Einige Operatoren sind diesbezüglich nicht eingeschränkt.
referentielle Abhängigkeit	abhängig, unabhängig	Laut Sprachdefinition ist ein Operator immer über eine Referenz mindestens mit einem anderen Operator verbunden, d. h. er ist referentiell abhängig.

Tabelle 5.1: Strukturelle Unterscheidungsmerkmale von XML-basierten Operatoren.

Beispieloperator aus XSLT	Unterscheidungsmerkmal			
	Attributierung	Verschachtelung	Inhalt	Abhängigkeit

`<transform version="1.0">`
`...`
`</transform>`
	attributiert	unverschachtelt	nichtleer	unabhängig

`<import href="infrastructure.xsl" />`	attributiert	unverschachtelt	leer	unabhängig

`<choose>`
`<when test="...">`
`...`
`</when>`
`<otherwise>`
`...`
`</otherwise>`
`</choose>`
	hybrid	verschachtelt	hybrid	unabhängig

`<call-template name="..." />`	attributiert	hybrid	leer	abhängig
`<comment>...</comment>`	attributlos	unverschachtelt	nichtleer	unabhängig

Tabelle 5.2: Beispiele von XML-basierten Operatoren.

Transformationsdomäne (z. B. einer bestimmten Zielsprache) ausgedrückt werden können. Diese höheren Operatoren werden jeweils von Operatoren aus niedrigeren Ebenen konstruiert.[2]

Abbildung 5.2 zeigt schematisch eine mögliche Grobeinteilung der Operatorhierarchie. Darin wird die Menge der neuen Operatoren nach ihrer Spezifität in universelle und domänenspezifische Operatoren unterschieden. Der Begriff Domäne bezieht sich hier nicht auf die gesamte horizontale XML-Transformationsdomäne, sondern vielmehr auf eine bestimmte Klasse von XML-Transformationen innerhalb dieser Domäne. Die zwei neuen Hauptebenen bilden intern wiederum eine Operatorhierarchie (siehe Abschnitt 5.2). Grundsätzlich ist eine gänzlich andere Einteilung unter Nutzung von anderen Kriterien möglich.

Universelle Operatoren sind Operatoren, die häufig verwendete elementare und u. U. andere universelle Operatoren, kapseln. Sie können in der gesamte XML-Transformationsdomäne

[2] In der Algebra spricht man auch von Herleitbarkeit. Beispielsweise können in der relationalen Algebra basierend auf den Armstrong-Axiomen Reflexivität, Verstärkung und Transitivität weitere Beziehungen wie z. B. Komposition, Dekomposition oder Pseudo-Transitivität hergeleitet werden.

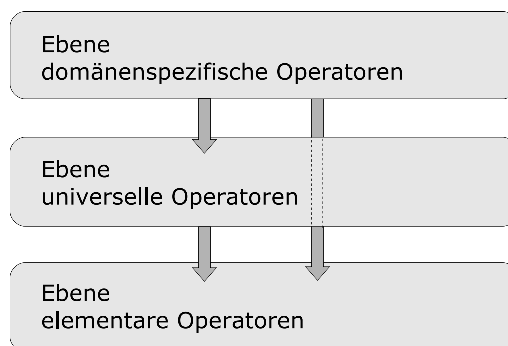


Abbildung 5.2: Grobeinteilung von Operatorhierarchien.

genutzt werden. Durch die Modularisierung des Codes können Redundanzen vermieden und dadurch das Fehlerrisiko potentiell verringert werden. Zusätzlich kann die Breite des Wirkungsspektrums erhöht werden. Unterstützt bspw. eine Änderungssprache keine Operatoren zum Verschieben oder Austauschen von Teilbäumen, wie im ersten Anwendungsszenario des Abschnittes 4.1.1, so können diese aus elementaren Operatoren ohne eine Anpassung der Änderungssprache konstruiert werden.

Domänenspezifische Operatoren werden aus elementaren, universellen und mglw. anderen domänenspezifischen Operatoren zusammengesetzt. Sie können bspw. für eine bestimmte Zielsprache konstruiert werden und sich deshalb an deren Sprachkonstrukten und Mustern orientieren (siehe zweites Anwendungsszenario im Abschnitt 4.1.2). Mit ihnen kann sichergestellt werden, dass Teile des Ergebnisses bzw. das gesamte Ergebnis der Transformation den Beschränkungen der Zielsprache entspricht. Neben der besseren Lesbarkeit der domänenspezifischen Operatoren erlauben sie eine Validierung sowie eine respektive Fehlermeldung auf Domänenebene. Dieser Fakt kann nicht überbewertet werden, da dies direkt mit der Praxistauglichkeit und ferner mit der Produktivität des Entwicklers korrespondiert.

Die fachliche Trennung zwischen universellen und domänenspezifischen Operatoren spiegelt sich neben deren Motivation ebenso im Entwicklungsvorgehen wieder. Da universelle Operatoren typischerweise Low-level-Operatoren zusammenfassen, die immer wieder in strukturell gleichen Kombinationen auftreten, ist ein Bottom-up-Entwicklungsvorgehen zu bevorzugen. Domänenspezifische Operatoren hingegen werden für strikt abgegrenzte Transformationsbereiche definiert. Dabei sind womöglich weitere Beschränkungen, wie bspw. die Präferenzen und Abneigungen der Zielanwendergruppe, zu beachten. Um diese und weitere Einschränkungen bzgl. des festgelegten Anwendungsbereiches zu erfassen, ist eine Domänenanalyse und demgemäß ein Vorgehen nach dem Top-down-Vorgehensmodell zu empfehlen. Eine tiefergreifende Diskussion der unterschiedlichen Vorgehensmodelle gibt Kapitel 7.

5.2 Detaillierte Beschreibung

Die Operatorhierarchie ist sehr flexibel. Nach Bedarf können neue Operatoren, basierend auf existierenden Operatoren, hinzugefügt werden. Technisch gesehen spielt es hierbei eine untergeordnete Rolle, ob es sich um universelle oder domänenspezifische Operatoren handelt. Beide Kategorien von Operatoren werden durch Ebenentransformationen implementiert, die die neu definierten Operatoren in existierende Operatoren überführen. Existierende Operatoren können entweder direkt durch eine Basistransformationssprache ausgeführt werden oder wurden ihrerseits durch Ebenentransformationen umgesetzt. Die Implementierung jedes höheren Operators in der Operatorhierarchie erfolgt somit durch Ebenentransformationsdefinitionen, die in ihrer Gesamtheit den Ebenentransformationsprozess in niedrigere bzw. letztendlich immer elementare Operatoren beschreiben. Die Semantik eines einzelnen Operators wird somit durch alle Ebenentransformationsdefinitionen, die nötig sind, um ihn in elementare Operatoren einer Basistransformationssprache zu überführen, und durch die überführten elementaren Operatoren selbst in ausführbarer Form codiert.

In besonderen Fällen (z. B. bei XML-basierten Operatoren) kann die Basistransformationssprache für die Ebenentransformationen wiederverwendet werden, da die höheren Operatoren ebenfalls in XML spezifiziert werden. Das Konzept der Operatorhierarchie ist dessen ungeachtet ein technologieunabhängiges Konzept. Es kann sowohl für den Technologieraum XML als auch für andere Technologieräume (z. B. Programmtransformationen oder Modelltransformationen) eingesetzt werden.

Für das Anwendungsszenario aus Abschnitt 4.1.1 kann bspw. XSLT als Sprache zur Beschreibung der Ebenentransformation genutzt werden. Der Code 5.1 zeigt einen Ausschnitt eines möglichen XSLT-Programms zur Überführung der neuen, universellen `move-after`- und `swap`-Operatoren in elementare XUpdate-Operatoren (vgl. Code 4.2 und Code 4.3). Die Beispielimplementierung der Ebenentransformation enthält dafür entsprechende Templates, die die höheren Operatoren ersetzen (Zeilen 7–16 und Zeilen 18–34). Sämtliche andere Operatoren werden einfach kopiert (Zeilen 37-42).

Die Ebenentransformationsdefinition im Code 5.1 beschreibt die einfachste und häufig auftretende Form einer Ebenentransformation. Es werden die universellen Operatoren `move-after` und `swap` lokal an Ort und Stelle durch niedrigere, hier elementare, Operatoren ersetzt. Dabei sind alle Informationen für die Ersetzung im höheren Operator sowie der Ebenentransformationsdefinition enthalten. Mit anderen Worten, es müssen keine weiteren erforderlichen Informationen global bzw. in der Umgebung des Operators gesammelt werden. Zudem ist die Erzeugung unbedingt, d. h. nur vom Pivot des Operators abhängig. Tabelle 5.3 fasst wichtige Unterscheidungsmerkmale von Ebenentransformationen zusammen.

```
<xslt:transform version="1.0" exclude−result−prefixes="xupe"                1
  xmlns:xslt="http://www.w3.org/1999/XSL/Transform"                         2
  xmlns="http://www.xmldb.org/xupdate"                                      3
  xmlns:xupe="http://www.informatik.uni−kiel.de/XUpdateExtend">             4
                                                                            5
<!−− Transformation der entsprechenden Operatoren −−>                       6
<xslt:template match="xupe:move−after" mode="perform−transformation">       7
  <xslt: variable name="id" select="generate−id()"/>                        8
  <if test="{@from}">                                                       9
    <variable name="INTERMEDIATE−MOVE−{$id}" select="{@from}"/>             10
    <remove select="{@from}"/>                                              11
    <insert−after select="{@to}">                                           12
      <value−of select="$INTERMEDIATE−MOVE−{$id}"/>                         13
    </insert−after>                                                         14
  </if>                                                                     15
</xslt:template>                                                            16
                                                                            17
<xslt:template match="xupe:swap" mode="perform−transformation">             18
  <xslt: variable name="id" select="generate−id()"/>                        19
  <if test="boolean({@first}) and boolean({@second})">                      20
    <variable name="INTERMEDIATE−FIRST−{$id}" select="{@first}"/>           21
    <variable name="INTERMEDIATE−SECOND−{$id}" select="{@second}"/>         22
    <rename select="{@first}">SUBTREES−FIRST</rename>                       23
    <rename select="{@second}">SUBTREES−SECOND</rename>                     24
    <insert−before select="//SUBTREES−FIRST">                               25
      <value−of select="$INTERMEDIATE−SECOND−{$id}"/>                       26
    </insert−before>                                                        27
    <insert−before select="//SUBTREES−SECOND">                              28
      <value−of select="$INTERMEDIATE−FIRST−{$id}"/>                        29
    </insert−before>                                                        30
    <remove select="//SUBTREES−FIRST"/>                                     31
    <remove select="//SUBTREES−SECOND"/>                                    32
  </if>                                                                     33
</xslt:template>                                                            34
                                                                            35
<!−− Kopieren der restlichen Operatoren −−>                                 36
<xslt:template match="node()|@∗" mode="perform−transformation">             37
  <xslt:copy>                                                               38
    <xslt:apply−templates select="node()|@∗" mode="perform−transformation"/> 39
  </xslt:copy>                                                              40
</xslt:template>                                                            41
<xslt:templafte match="text()" mode="perform−transformation"/>              42
                                                                            43
</xslt:transform>                                                           44
```

Code 5.1: Ebenentransformation für das Verschieben und Austauschen von Teilbäumen mit XUpdate.

Grundsätzlich sind alle Ausprägungskombinationen von Operatorebenentransformationen, die in der Tabelle 5.3 aufgeführt sind, mit dem Konzept der Operatorhierarchie realisierbar. Zwei Ausprägungen und damit auch ihre Ausprägungskombinationen sind allerdings problematisch für eine automatische Auswahl der Ebenentransformationsdefinitionen beim Ebenentransformationsprozess. Ein Problemfall liegt vor, wenn die Quellinformationen nicht nur lokal im Operator, sondern auch außerhalb des Operators global im Quelldokument verteilt sind. Dann steht dieser Operator zwangsläufig mit anderen Operatoren in Beziehung. Infolgedessen kann das Ergebnis des Ebenentransformationsprozesses abhängig von der Ausführungsreihenfolge der Ebenentransformationen nichtdeterministisch sein. Dies tritt ein, wenn bspw. die anderen, in

Unterscheidungsmerkmal	Ausprägungen	Beschreibung
Ort der Quellinformationen	lokal, global	Um eine Ebenentransformation zu konfigurieren, reichen die lokalen Informationen, die im zu transformierenden Operator abgelegt sind. Alternativ müssen u. U. Informationen anderer Operatoren, z. B. aus der Umgebung des zu transformierenden Operators, selektiert werden.
Ort der Zieloperatoren	lokal, verteilt	Bei der Ausführung der Ebenentransformation werden Zieloperatoren lokal an der Stelle erzeugt, an der sich der ursprüngliche Operator befand. Alternativ können sich die Zieloperatoren auch auf verschiedene Stellen im Zieldokument verteilen.
Bedingtheit der Operatorerzeugung	unbedingt, existenzbedingt, typbedingt, wertbedingt, hybrid	Die Zieloperatoren können unbedingt (immer wieder gleich) aus dem zu transformierenden Operator erzeugt werden. Variationen in den Zieloperatoren können abhängig von Existenzen (Elemente, Attribute), Typen (Typ des Textinhaltes eines Elementes, Typ des Attributwertes), konkreten Werten (Textinhalt eines Elementes, Attributwert) oder einer Mischung aus diesen im zu transformierenden Operator resultieren.

Tabelle 5.3: Unterscheidungsmerkmale einer Operatorebenentransformation.

Beziehung stehenden Operatoren von einer anderen Ebenentransformation erzeugt werden. Ein solche Abhängigkeit besteht bspw. beim Funktionsmechanismus zwischen Definition der Funktion und Aufruf der Funktion (siehe auch Abschnitt 8.1.1). Ähnliches gilt bei der Erzeugung verteilter Zieloperatoren. Um diese an den richtigen Stelle einfügen zu können, sind notwendigerweise Informationen über entsprechende Operatoren des Zieldokumentes erforderlich. Es ergibt sich hier ein vergleichbares Problem, wie bei der Verwendung voneinander abhängiger Aspekte innerhalb der aspektorientierten Programmierung (siehe z. B. [NBA05] oder [MW08]). Beides kann zu unerwünschtem Verhalten führen, da die Operatorhierarchie flexibel durch neue Operatoren angereichert werden kann, die damit die Quell- und Zieldokumentstruktur verändern. Um den ersten Fall für eine automatische Ausführung nicht generell verbieten zu müssen, muss sichergestellt werden, dass solche Operatoren zeitgleich transformiert werden und somit alle notwendigen Informationen während der Ebenentransformation zur Verfügung stehen. Entsprechend werden solche Operatoren in sogenannten Sprachkomponenten zusammengefasst.

Sprachkomponenten

Eine **Sprachkomponente** definiert eine nichtleere Menge von Operatoren und ihre Kontextabhängigkeiten. Eine Sprachkomponente ist somit eine Sammlung von in direkter Beziehung stehenden Operatoren sowie eigenständigen Operatoren, die aber funktional oder auf andere Weise kohärent sind. Sprachkomponenten können entsprechend der in ihnen definierten Operatoren in universelle oder domänenspezifische Sprachkomponenten unterschieden werden. Ungeachtet dessen, werden die in einer Sprachkomponente gebündelten Operatoren durch eine entsprechende Ebenentransformationsdefinition implementiert, die die Überführung für jeden einzelnen Operator enthält. Die Konzentration in einer Ebenentransformationsdefinition verringert das Risiko, das in direkter Beziehung stehende Operatoren in unterschiedlichen Ebenentransformationen abgearbeitet werden. Gemeinsam mit dem Ranking einer Sprachkomponente kann eine gleichzeitige Abarbeitung garantiert werden. Jeder Sprachkomponente wird ein Namensraum zugeordnet. Da jeder Operator nur einmal in genau einer Sprachkomponente definiert sein darf,

können alle Operatoren über den Namensraum der Sprachkomponente eindeutig identifiziert werden. Darüber hinaus kann mit Hilfe des Namensraums die Ebenentransformationsdefinition, welche die in der Sprachkomponente definierten Operatoren implementiert, automatisiert zugeordnet werden (siehe Abschnitt 5.3).

Mit der Einführung von Sprachkomponenten kann nun in Operatoren unterschieden werden, die aus existierenden Operatoren anderer Sprachkomponenten (inkl. den elementaren Operatoren einer Basistransformationssprache), der gleichen Sprachkomponente oder einer Mischung aus beiden konstruiert werden. Basieren neue Operatoren u. a. auf Operatoren der eigenen Sprachkomponente, muss eine mehrmalige Anwendung der zugehörigen Ebenentransformationsdefinition im Ebenentransformationsprozess erfolgen. In Sonderfällen kann es auch sinnvoll sein, dass ein Operator durch sich selbst rekursiv definiert wird. Grundsätzlich sind solche direkten rekursiven aber auch indirekten, wechselseitigen rekursiven Definitionen zwischen mehreren Operatoren auch verschiedener Sprachkomponenten aufgrund der Maxime, dass neue Operatoren immer nur aus existierenden Operatoren aufgebaut sind, ausgeschlossen. Wenn jedoch bestimmte Restriktionen (z. B. an einer determinierten Stelle muss sich die rekursive Definition auflösen sowie die Lokalität der Quellinformationen und der erzeugten Zieloperatoren muss gegeben sein) eingehalten werden, stellen solche direkten und indirekten zyklische Abhängigkeiten zwischen Operatoren kein Problem dar. Ist dies nicht gewährleistet, können bspw. unerwünschte Endlosschleifen im Ebenentransformationsprozess auftreten.

Abhängig von den in einer Sprachkomponente definierten Operatoren können Sprachkomponenten einer Ordnung (Rank) unterzogen werden. Alle Operatoren in einer Sprachkomponente und die der Sprachkomponente zugeordnete Ebenentransformationsdefinition erhalten das gleiche Ranking wie ihre Sprachkomponente. Mit Hilfe des Rankings kann gesichert werden, dass alle Operatoren einer Sprachkomponente zur gleichen Zeit transformiert werden[3]. Zudem kann durch das Ranking von Sprachkomponenten die Anzahl der notwendigen Ebenentransformationen, die in einem vollständigen Ebenentransformationsprozess ausgeführt werden müssen, minimiert werden (vgl. die verschiedenen Algorithmen für den Ebenentransformationsprozess). Ausschlaggebend für die Gewichtung einer Sprachkomponente sind die Gewichtungen der Operatoren, aus denen die in dieser Sprachkomponente definierten Operatoren konstruiert werden. Dies sind genau die Operatoren, die aus der zugeordneten Ebenentransformationsdefinition erzeugt werden können. Die elementaren Operatoren der Basistransformationssprache erhalten per Definition den niedrigsten Wert 0. Können nun nur elementare Operatoren erzeugt werden, so erhält die Sprachkomponente den Wert 1 (Inkrementierung um eins). Sind die Operatoren dieser Sprachkomponente das Ergebnis einer Ebenentransformation einer anderen Sprachkomponente, so bekommt sie mindestens einen Wert 2 oder höher, falls zusätzlich andere, höher gewichtete Operatoren erzeugt werden. Zyklische Abhängigkeiten führen zu keiner Inkrementierung.

Abbildung 5.3 demonstriert eine stark abstrahierte Beispielhierarchie. Sie soll den grundlegenden Aufbau veranschaulichen und verzichtet deshalb bewusst auf Verschachtelung, Attributierung

[3]Eine Ausnahme stellen (wechselseitige) rekursive Operatordefinitionen dar, die dadurch direkte oder indirekte zyklischen Abhängigkeiten mit anderen Operatoren besitzen. Für sie müssen jedoch gesonderte Restriktionen eingehalten werden.

und Inhalte von Operatoren sowie referentielle Abhängigkeiten zwischen Operatoren. Jeder Operator ist dadurch attributlos, unverschachtelt, leer und referentiell unabhängig. Um die Operatorhierarchie möglichst einfach zu gestalten, werden die Operatorebenentransformationen ebenso weitestmöglich eingeschränkt. Es sind nur lokale Ersetzungen ohne Bedingungen möglich. Die Operatoren werden durch (weiße) Rechtecke und die Kompositionsbeziehungen zwischen den Operatoren durch Pfeile (gerichtete Kanten) abgebildet. Eine gerichtete Kante ist ein geordnetes Paar von Operatoren, bei der der erste Operator (Startknoten) durch den zweiten Operator (Endknoten) ersetzt wird. In der Abbildung 5.3 kann ein Operator dabei mehrere ausgehende und eingehende Kanten besitzen, da er aus verschiedenen Operatoren komponiert sein kann bzw. selbst komponiert wird. Um dabei noch Mehrfachkanten zum selben Operator zu vermeiden, weil der Zieloperator mehrfach bei der Operatorebenentransformation erzeugt wird, steht der zweite Operator für viele Operatoren der selben Art. Von der Reihenfolge, in der die Operatoren erzeugt werden, wird in Abbildung 5.3 aus Übersichtsgründen ebenso abstrahiert.

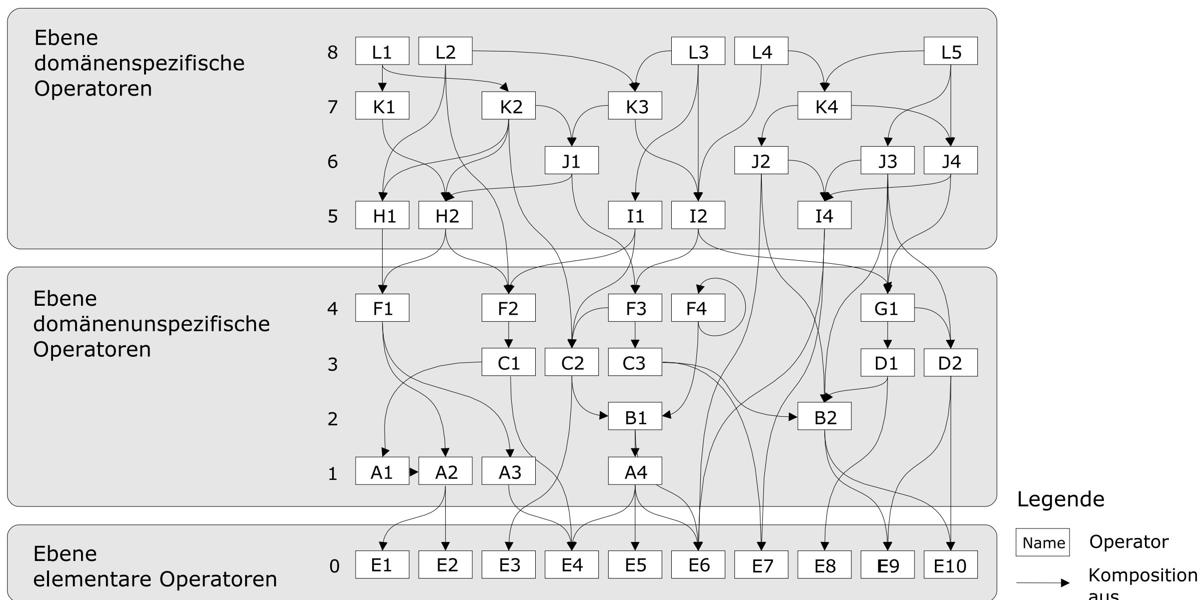


ABBILDUNG 5.3: Schematische Beispieloperatorhierarchie mit Sprachkomponenten.

Wie in Abbildung 5.3 den Namen der Operatoren zu entnehmen ist, werden die neuen Operatoren in den Sprachkomponenten $\mathrm{SK_A}$, $\mathrm{SK_B}$, $\mathrm{SK_C}$, $\mathrm{SK_D}$, $\mathrm{SK_F}$, $\mathrm{SK_G}$, $\mathrm{SK_H}$, $\mathrm{SK_I}$, $\mathrm{SK_J}$, $\mathrm{SK_K}$ und $\mathrm{SK_L}$ definiert. Die Sprachkomponente $\mathrm{SK_A}$ stellt bspw. die Operatoren A1, A2, A3 und A4 bereit ($\mathrm{SK_A} = \{A1, A2, A3, A4\}$). Zudem gilt $\mathrm{SK_B} = \{B1, B2\}$, $\mathrm{SK_C} = \{C1, C2, C3\}$, usw.

Als Beispiel sollen die Ebenentransformationsdefinitionen $\mathrm{ETD_A}$, $\mathrm{ETD_B}$ und $\mathrm{ETD_C}$ für die respektiven Sprachkomponenten genauer angegeben werden. Sie sind durch folgende Menge von simplen lokalen Transformationsregeln gegeben:

$$\text{ETD}_\text{A} = \{\text{A1} \Rightarrow \text{A2 A2},\ \text{A2} \Rightarrow \text{E1 E2 E1},\ \text{A3} \Rightarrow \text{E4},\ \text{A4} \Rightarrow \text{E4 E5 E4 E6}\}$$
$$\text{ETD}_\text{B} = \{\text{B1} \Rightarrow \text{A4 E6 E6},\ \text{B2} \Rightarrow \text{E10 E9 E9}\}$$
$$\text{ETD}_\text{C} = \{\text{C1} \Rightarrow \text{A1 E4},\ \text{C2} \Rightarrow \text{E3 E3 B1},\ \text{C3} \Rightarrow \text{E8 B2 B2}\}$$

Obige Regeln bestehen aus einer linken und einer rechten Seite, die durch einen Pfeil ($\Rightarrow$) voneinander getrennt sind. Eine Transformationsregel kann angewendet werden, wenn ein Operator mit der linken Seite der Regel übereinstimmt. In diesem Fall wird er durch die Operatoren der rechten Seite der Regel ersetzt.

Für das stark vereinfachtes Beispiel ohne Verschachtelung von Operatoren reicht folgende Vorschrift für eine Ebenentransformation (LEVELTRANSFORM). Die Eingabe (eine Transformationsdefinition TD, welche in einer bestimmten Reihenfolge Operatoren enthält) wird auf eine determinierte Weise (von links nach rechts) durchlaufen. Sobald auf einen Operator eine Transformationsregel innerhalb der Ebenentransformationsdefinition anwendbar ist, wird diese ausgeführt. Anschließend wird die Ebenentransformation beim nächsten Operator bzgl. der Traversierungsreihenfolge fortgesetzt. Kann keine Transformationsregel auf einen Operator angewendet werden, wird dieser übersprungen. Sind mehrere Transformationsregeln auf einen Operator anwendbar, wird eine Fehlermeldung erzeugt und die Ebenentransformation mit entsprechendem Fehlerstatus beendet.

Anhand der Beispielebenentransformationsdefinitionen ETD_A, ETD_B und ETD_C können die Gewichtungen der Sprachkomponenten SK_A, SK_B und SK_C durch den auf Seite 91 beschriebenen RANK-Algorithmus berechnet werden. Demnach erhält SK_A den Wert 1 (Zieloperatoren in ETD_A sind nur elementare und eigene Operatoren von SK_A, was zu keiner Inkrementierung führt), SK_B den Wert 2 (Zieloperatoren sind sowohl elementare aber auch Operatoren von SK_A) und SK_C den Wert 3 (Zieloperatoren sind u. a. auch Operatoren von SK_B). Entsprechend den Gewichtungen sind die Operatoren in der Beispieloperatorhierarchie von Abbildung 5.3 in Ebenen angeordnet. Auf einer Ebene können durchaus mehrere Sprachkomponenten vorkommen. Dies ist aber nur möglich, wenn die Operatoren in diesen Sprachkomponenten nicht aufeinander aufbauen.

Sprachen

Sprachkomponenten sind die Basis für modular zusammensetzbare Sprachen [FP08]. Ein Sprache wird durch eine nichtleere Menge von Sprachkomponenten definiert. Das Vokabular einer neuen Sprache resultiert aus den definierten Operatoren in den Sprachkomponenten. Abhängig von dem genutzten Sprachkomponenten können sich folglich grundverschiedene Sprachvokabulare ergeben. Die Sprache L_A in Abbildung 5.4 setzt sich bspw. aus den elementaren Operatoren der Basistransformationssprache $\text{L}_\text{E} = \{w : w \in \{\text{E1}, \text{E2}, \text{E3}, \text{E4}, \text{E5}, \text{E6}, \text{E7}, \text{E8}, \text{E9}, \text{E10}\}^*\}$ und den Operatoren der Sprachkomponente SK_A, der Sprachkomponente SK_B und der Sprachkomponente SK_C zusammen ($\text{L}_\text{A} = \{w : w \in (\text{L}_\text{E} \cup \text{SK}_\text{A} \cup \text{SK}_\text{B} \cup \text{SK}_\text{C})^*\}$). Die Basistransformationssprache wird somit um universelle Operatoren angereichert. Im Kontrast dazu wird die Sprache L_B

durch die Sprachkomponenten SK_K und SK_L definiert ($L_B = \{w : w \in (SK_K \cup SK_L)^*\}$). Es ergibt sich somit ein komplett neues Vokabular mit nur domänenspezifischen Operatoren. Die Sprache L_C ist eine Mischung aus beidem, sie beinhaltet sowohl Operatoren der Basistransformationssprache, universelle Operatoren als auch domänenspezifische Operatoren. Eine Kombination aus nur universellen Sprachkomponenten ist ebenso möglich. Beispielsweise könnte so eine neue universelle XML-spezifische Transformationssprache entstehen, die „sauberer“ formuliert werden kann, weil bestimmte elementare Operatoren in ihr nicht verfügbar sind. Ein bekanntes Anwendungsbeispiel ist der Sprungbefehl [Dij02], der in höheren Programmiersprachen (z. B. Java[4]) bewusst weggelassen wird.

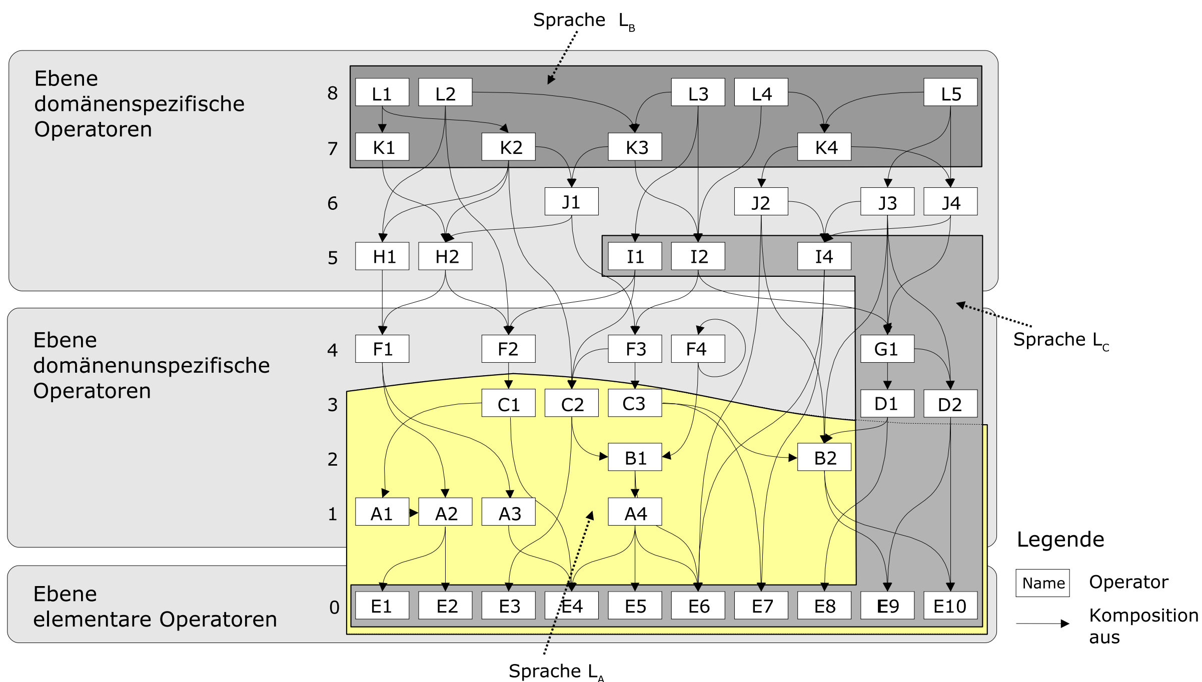


ABBILDUNG 5.4: Mögliche Sprachen einer schematischen Beispieloperatorhierarchie.

Der äußere Unterschied einer modular, aus mehreren Sprachkomponenten zusammengesetzten Sprache zu einer monolithischen Transformationssprache besteht lediglich darin, dass Operatoren verschiedene Namensräume aufweisen können. Dies sind genau die Namensräume der Sprachkonstrukte, in denen sie definiert werden. Der Anwender muss die Namensräume der Sprachkomponenten kennen, aus denen die höhere Transformationssprache zusammengesetzt wird. Der interne Unterschied liegt im Wesentlichen darin, dass die gesamte Sprache nicht in einem Schritt übersetzt oder in einer anderen Form interpretiert wird, wie bei der Sprachhierarchie von universellen Programmiersprachen, sondern in einem mehrstufigen Ebenentransformationsprozess in letztendlich elementare, ausführbare Operatoren transformiert wird. Die Ausführungsreihenfolge der Ebenentransformationen im Ebenentransformationsprozess spielt dabei für die Anzahl der notwendigen Ebenentransformationen eine entscheidende Rolle, wie die im fol-

[4]In Java wird nur das entsprechende Schlüsselwort `goto` reserviert, aber nicht implementiert.

genden Abschnitt vorgeschlagenen Algorithmen zeigen. Für eine vollautomatische Ausführung des Ebenentransformationsprozesses müssen einige Vorbedingungen erfüllt sein.

Algorithmen für den Ebenentransformationsprozess

Für den Ebenentransformationsprozess zur Überführung einer höheren Transformationsdefinition TD mit Hilfe einer endlichen, nichtleeren Menge von Ebenentransformationsdefinitionen ETD, wobei die Ebenentransformationsdefinition ETD_i jeweils der entsprechenden Sprachkomponenten SK_i zugeordnet ist, können verschiedene Algorithmen verwendet werden.

Algorithmus 1 stellt eine mögliche Umsetzung vor. Aufgrund der Eigenschaft des Algorithmus, dass die Ebenentransformationsdefinitionen in Abhängigkeit von der Eingabe ausgeführt werden, unterstützt er nur bedingte und unbedingte lokale Ersetzungen von Operatoren an Ort und Stelle. Beispielsweise bei Operatorebenentransformationen mit verteilten Zieloperatoren ist der Determinismus des Algorithmenergebnisses nicht mehr garantiert (siehe Tabelle 5.4).

Algorithmus 1 Ebenentransformationsprozessalgorithmus: `first`

Input: $\mathrm{TD} = [\mathrm{op}_1, \mathrm{op}_2, ..., \mathrm{op}_m]$, $\mathrm{SK} = \{\mathrm{SK}_i : 1 \leq i \leq n \text{ und } i, n \in \mathbb{N}\}$,
$\mathrm{ETD} = \{\mathrm{ETD}_i : 1 \leq i \leq n \text{ und } i, n \in \mathbb{N}\}$

Output: ersetzte $\mathrm{TD} = [\mathrm{op}_1, \mathrm{op}_2, ..., \mathrm{op}_o]$, wobei $\mathrm{op}_j \notin \mathrm{SK}_i$ für $1 \leq j \leq o$ und $j, o \in \mathbb{N}$

```
1: function First(TD, SK, ETD)
2:     foreach op ∈ TD do
3:         if op ∈ SK_i then
4:             TD ← LevelTransform(TD, ETD_i)
5:             return First(TD, SK, ETD)
6:         end if
7:     end foreach
8:     return TD
9: end function
```

Der `first`-Algorithmus sucht das erste Vorkommen eines höheren Operators op in der Transformationsdefinition TD. Wird kein höherer Operator gefunden, wird die Transformationsdefinition TD als Ergebnis ausgegeben. Ansonsten wird für den gefundenen Operator ($\mathrm{op} \in \mathrm{SK}_i$) die zugeordnete Ebenentransformationsdefinition ETD_i mit der gesamten Transformationsdefinition TD als Eingabe ausgeführt. Die Ebenentransformation transformiert dabei entsprechend der Ebenentransformationsdefinition ETD_i diesen Operator und alle zu dieser Sprachkomponente SK_i gehörenden Operatoren in elementare und/oder nichtelementare Operatoren. Mit der durch das Ergebnis der Ebenentransformation ersetzten Transformationsdefinition TD wird die First-Funktion erneut aufgerufen, um weitere höhere Operatoren zu transformieren.

Die Abbildung 5.5 zeigt die Anwendung des `first`-Algorithmus auf die Eingabe `E7 A1 E7 B1 E4 C1 C2 E4 E7 A4`. An den Ebenentransformationen (durch graue Pfeile gekennzeichnet) des Ebenentransformationsprozesses sind jeweils die eingesetzten Ebenentransformationsdefinitionen sowie die daraus angewendeten Regeln notiert. Die zu ersetzenden Operatoren werden unterstrichen und die ersetzten Operatoren fett abgebildet.

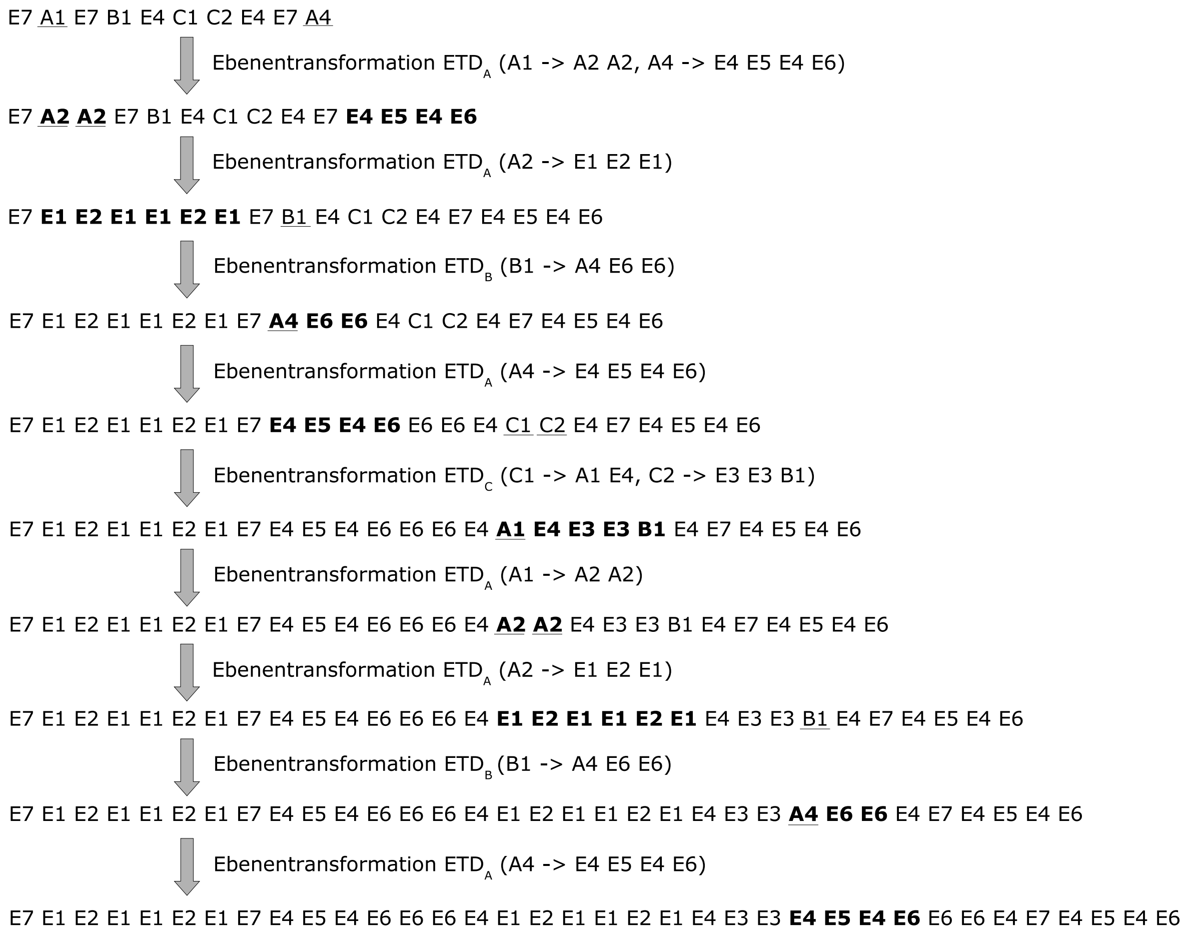
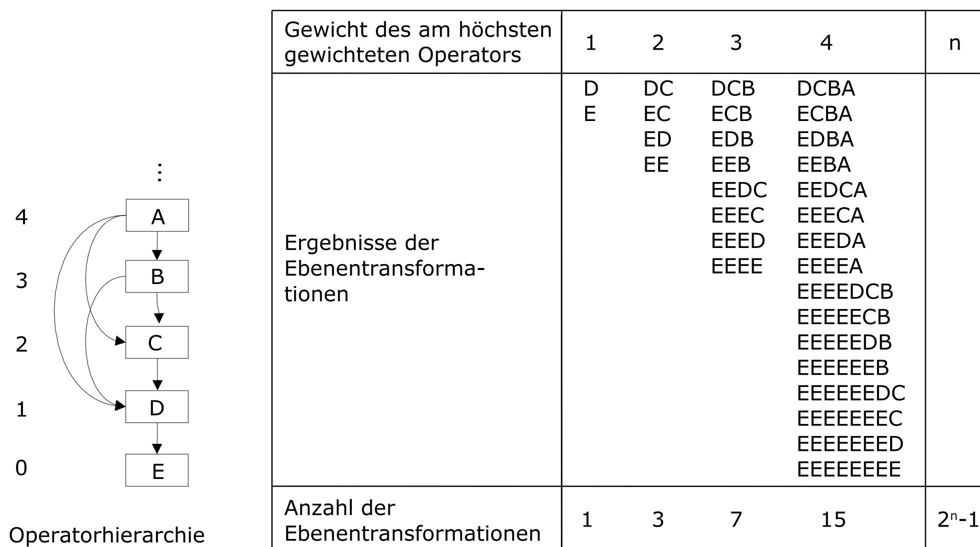


ABBILDUNG 5.5: Beispielanwendung des `first`-Ebenentransformationsprozessalgorithmus.

Der Ebenentransformationsprozess in Abbildung 5.5 veranschaulicht, dass selbst bei diesem simplen Beispiel viele Ebenentransformationen notwendig sein können. Betrachtet man die Anzahl der Ebenentransformationen genauer, so können leicht Beispiele konstruiert werden, bei denen der `first`-Algorithmus sogar $2^n - 1$ Ebenentransformationen erfordert. n bezeichnet dabei die Gewichtung des am höchsten gewichteten Operators innerhalb einer Transformationsdefinition TD. Die Anzahl von $2^n - 1$ Ebenentransformationen wird dabei schon erreicht, wenn keine zyklischen Abhängigkeiten existieren. Bestehen zyklische Abhängigkeiten erhöht sich diese Anzahl u. U. zusätzlich.

In Abbildung 5.6 wird ein Ausschnitt aus einer solchen Beispieloperatorhierarchie dargestellt. Es wird dabei angenommen, dass die Operatoren A, B, C und D höhere Operatoren aus jeweils eigenen Sprachkomponenten sind. Die Pfeile in der Operatorhierarchie geben die Kompositionsbeziehungen an. So wird bspw. der Operator A durch die Operatoren DCB lokal ersetzt, der Operator B durch DC lokal ersetzt, usw.



Gewicht des am höchsten gewichteten Operators	1	2	3	4	n
Ergebnisse der Ebenentransformationen	D E	DC EC ED EE	DCB ECB EDB EEB EEDC EEEC EEED EEEE	DCBA ECBA EDBA EEBA EEDCA EEECA EEEDA EEEEA EEEEDCB EEEEECB EEEEEDB EEEEEEB EEEEEEDC EEEEEEEC EEEEEEED EEEEEEEE	
Anzahl der Ebenentransformationen	1	3	7	15	2^n-1

Abbildung 5.6: Untere Schranke des `first`-Algorithmus.

Zur Minimierung der Ebenentransformationsanzahl müssen die Einflussgrößen betrachtet werden. Die Anzahl der Ebenentransformationen ist im Wesentlichen von folgenden Faktoren abhängig:

- der Kompliziertheit (Anzahl der Operatoren sowie Sprachkomponenten) und Komplexität der Operatorhierarchie (Abhängigkeiten zwischen den Operatoren),
- der Art und Anzahl der Transformationsregeln in den Ebenentransformationsdefinitionen,
- der Art und Anzahl der verwendeten Operatoren in der Eingabe und
- dem Algorithmus des Ebenentransformationsprozesses.

Nimmt man die ersten drei Kenngrößen – Kompliziertheit und Komplexität der Operatorhierarchie, Art und Anzahl der Transformationsregeln und Art und Anzahl der verwendeten Operatoren in der Eingabe – als gegeben an, können durch eine effiziente Ausführung des Ebenentransformationsprozesses die Anzahl der Ebenentransformationen und die dadurch auftretenden Kosten für das zusätzliche Lesen und Durchlaufen der Eingabe, für das Abarbeiten der Ebenentransformation sowie für das Schreiben der Ausgabe reduziert werden.

Anstatt die Ebenentransformationen zufällig nach dem Auftreten der höheren Operatoren in den Transformationsdefinitionen auszuführen, können die Ebenentransformationsdefinitionen nach einer bestimmten Priorität angewendet werden. Ein Prioritätsmerkmal, was einen erheblichen Einfluss auf die Anzahl der Ebenentransformationen nimmt, ist das Ranking der Operatoren. Im Folgenden soll ein Algorithmus vorgeschlagen werden, der immer die am höchsten gewichteten Operatoren bevorzugt, wobei die Gewichte der Operatoren den Gewichten g der Sprachkomponenten ($\text{SK} = \{\text{SK}_{i,g} : 1 \leq i \leq n \text{ und } \ i, n, g \in \mathbb{N}\}$) entsprechen, in denen sie definiert werden. Aufgrund dessen ist der Algorithmus 2 nicht auf die Lokalität der Quellinformationen eines Operators beschränkt (siehe Tabelle 5.4).

Algorithmus 2 Ebenentransformationsprozessalgorithmus: `maxrank`

Input: $\text{TD} = [\text{op}_1, \text{op}_2, ..., \text{op}_m]$, $\text{SK} = \{\text{SK}_{i,g} : 1 \leq i \leq n \text{ und } i, n, g \in \mathbb{N}\}$,
$\text{ETD} = \{\text{ETD}_i : 1 \leq i \leq n \wedge i, n \in \mathbb{N}\}$

Output: ersetzte $\text{TD} = [\text{op}_1, \text{op}_2, ..., \text{op}_o]$, wobei $\text{op}_j \notin \text{SK}_i$ für $1 \leq j \leq o$ und $j, o \in \mathbb{N}$

```
 1: function MAXRANK(TD, SK, ETD)
 2:     max ← 0
 3:     foreach op ∈ TD do
 4:         if op ∈ SK_{i,g} ∧ RANK(SK_{i,g}) > max then
 5:             max ← g
 6:             ETD_max ← ETD_i
 7:         end if
 8:     end foreach
 9:     if max ≠ 0 then
10:         TD ← LEVELTRANSFORM(TD, ETD_max)
11:         return MAXRANK(TD, SK, ETD)
12:     else
13:         return TD
14:     end if
15: end function
```

Beim `maxrank`-Algorithmus wird die gesamte Transformationsdefinition TD durchlaufen und immer wenn ein höherer Operator op ($\text{op} \in \text{SK}_{i,g}$) gefunden wird, wird die Gewichtung g der Sprachkomponente $\text{SK}_{i,g}$ mit der bisher am höchsten gewichteten (Maximum) verglichen. Sollte die aktuelle Gewichtung g höher als das bisherige Maximum sein oder es existiert bisher kein Maximum, dann wird die Gewichtung g als neues Maximum gesetzt und die zugehörige Ebenentransformationsdefinition ETD_i zwischengespeichert. Wird in diesem Präprozess kein Maximum gesetzt, da keine höheren Operatoren in TD existieren, wird der Ebenentransformationsprozess beendet und die Transformationsdefinition TD als Ergebnis ausgegeben. Gibt es jedoch ein

Maximum, wird die zwischengespeicherte Ebenentransformationsdefinition ETD_{max} mit der gesamten Transformationsdefinition TD als Eingabe gestartet. Die Ebenentransformation transformiert entsprechend der Ebenentransformationsdefinition ETD_{max} alle zu dieser Sprachkomponente gehörenden Operatoren in elementare und/oder nichtelementare Operatoren. Mit der durch das Ergebnis der Ebenentransformation ersetzten Transformationsdefinition TD wird die MaxRank-Funktion erneut aufgerufen, um weitere höhere Operatoren zu transformieren.

Die zentrale Idee des `maxrank`-Algorithmus ist, dass durch die Bevorzugung von höher gewichteten Operatoren im Durchschnitt bei einer Ebenentransformation mehr Operatoren transformiert werden können und dadurch weniger Ebenentransformationen benötigt werden. Die Anzahl der angewendeten Transformationsregeln selbst verkleinert sich nicht. Durch die Reduzierung der Anzahl der Ebenentransformationen verringert sich jedoch der entsprechende Mehraufwand für das Einlesen und Verarbeiten der Ebenentransformationsdefinitionen. Es müssen allerdings die Kosten für die einmalige Gewichtung der Sprachkomponenten und ihrer Operatoren und die wiederkehrenden Kosten für das vollständige Durchlaufen der Transformationsdefinitionen zum Finden der am höchsten gewichteten Operatoren dagegen gerechnet werden.

Bei einer Eingabe von `E7 A1 E7 B1 E4 C1 C2 E4 E7 A4` würde unter Anwendung des Optimierungsansatzes der in Abbildung 5.7 gezeigte Ebenentransformationsprozess entstehen. Im Gegensatz zu den ursprünglichen neun Ebenentransformationen, die jeweils mit dem Lesen der Eingabe und der Ebenentransformationsdefinition selbst sowie dem Schreiben der Ausgabe verbunden war (vgl. Abbildung 5.5), müssen in diesem konkreten Anwendungsbeispiel nun nur noch vier Ebenentransformationen durchgeführt werden. Schon bei diesem minimalen Beispiel reduzieren sich die Mehrkosten für das Einlesen und Schreiben deutlich.

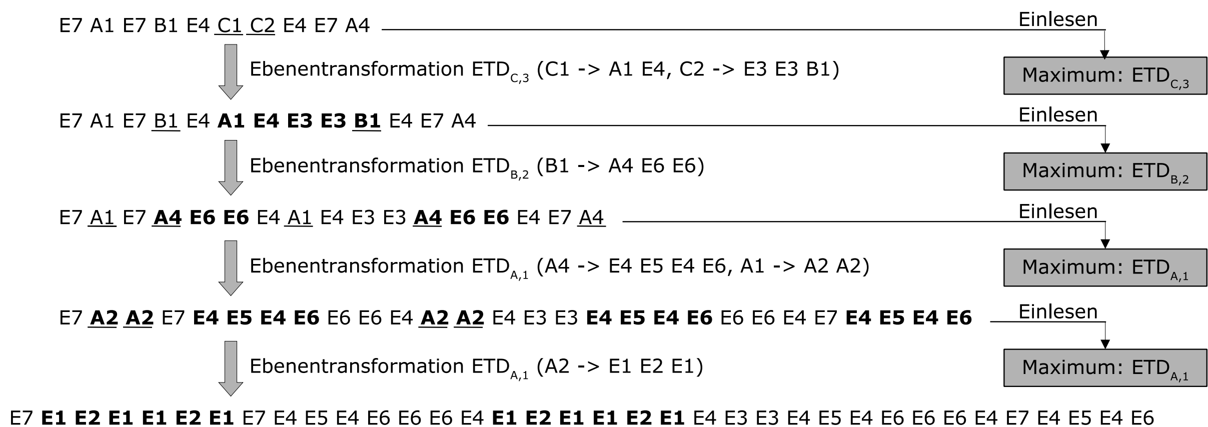


Abbildung 5.7: Beispielanwendung des `maxrank`-Ebenentransformationsalgorithmus.

Betrachtet man wiederum die konstruierte Beispielhierarchie aus Abbildung 5.6, so kann gezeigt werden, dass mit dem `maxrank`-Algorithmus nur n Ebenentransformationen notwendig sind. Es kann ebenso gezeigt werden, dass der optimierte Algorithmus höchstens genauso viele Ebenentransformationen haben kann, wie der einfache `first`-Algorithmus. Dies ist genau dann der Fall, wenn die höheren Operatoren bereits bzgl. der Gewichtung in absteigend sortierter Form

als Eingabe vorliegen.

Hinsichtlich der Anzahl der Ebenentransformationen ist der `maxrank`-Algorithmus optimal. Allerdings verlangt er jeweils ein vollständiges Einlesen der Transformationsdefinitionen innerhalb des Präprozesses nach jeder Ebenentransformation. Eine Alternative stellt der Algorithmus 3 dar.

Algorithmus 3 Ebenentransformationsprozessalgorithmus: `sort`

Input: $\text{TD} = [\text{op}_1, \text{op}_2, ..., \text{op}_m]$, $\text{SK} = \{\text{SK}_{i,g} : 1 \leq i \leq n \text{ und } i, n, g \in \mathbb{N}\}$,
$\text{ETD} = \{\text{ETD}_i : 1 \leq i \leq n \text{ und } i, n \in \mathbb{N}\}$

Output: ersetzte $\text{TD} = [\text{op}_1, \text{op}_2, ..., \text{op}_o]$, wobei $\text{op}_j \notin \text{SK}_i$ für $1 \leq j \leq o$ und $j, o \in \mathbb{N}$

1: **function** SORT(TD, SK, ETD)
2: $\quad$ $tree_{\text{ETD}} \leftarrow$ nach dem Ranking absteigend sortierter Baum
3: $\quad$ **foreach** op $\in$ TD **do**
4: $\quad\quad$ **if** $\text{op} \in \text{SK}_{i,g} \wedge \text{ETD}_i \notin tree_{\text{ETD}}$ **then**
5: $\quad\quad\quad$ $tree_{\text{ETD}} \leftarrow \text{ADD}(tree_{\text{ETD}}, \text{ETD}_i, \text{RANK}(\text{SK}_{i,g}))$
6: $\quad\quad$ **end if**
7: $\quad$ **end foreach**
8: $\quad$ **if** $tree_{\text{ETD}} \neq \emptyset$ **then**
9: $\quad\quad$ **while** $tree_{\text{ETD}} \neq \emptyset$ **do**
10: $\quad\quad\quad$ $\text{ETD}_{max} \leftarrow \text{FIRST}(tree_{\text{ETD}})$
11: $\quad\quad\quad$ $tree_{\text{ETD}} \leftarrow \text{REMOVE}(tree_{\text{ETD}}, \text{ETD}_{max})$
12: $\quad\quad\quad$ $\text{TD} \leftarrow \text{LEVELTRANSFORM}(\text{TD}, \text{ETD}_{max})$
13: $\quad\quad$ **end while**
14: $\quad\quad$ **return** SORT(TD, SK, ETD)
15: $\quad$ **else**
16: $\quad\quad$ **return** TD
17: $\quad$ **end if**
18: **end function**

Im `sort`-Algorithmus wird im Präprozess einmal die gesamte Eingabe nach höheren Operatoren durchsucht und bei jedem höheren Operator ($\text{op} \in \text{SK}_{i,g}$) eine Referenz auf die entsprechende Ebenentransformationsdefinition (ETD_i) in eine Datenstruktur (z. B. binärer Baum) abgelegt, falls diese noch nicht darin vorhanden ist. Um ein schnelles Finden zu gewährleisten, wird die Datenstruktur bereits nach den Gewichtungen g absteigend sortiert. Ist die Datenstruktur nach dem Durchsuchen leer, weil keine höheren Operatoren in der Transformationsdefinition TD existieren, so wird der Ebenentransformationsprozess beendet und die Transformationsdefinition TD als Ergebnis ausgegeben. Ist die sortierte Datenstruktur nicht leer, dann werden die Ebenentransformationen beginnend mit der am höchsten gewichteten Ebenentransformationsdefinition mit der gesamten Transformationsdefinition TD als Eingabe ausgeführt. Die Ebenentransformation transformiert entsprechend der Ebenentransformationsdefinition ETD_i den höheren Operator und alle zu dieser Sprachkomponente $\text{SK}_{i,g}$ gehörenden Operatoren in elementare und/oder nichtelementare Operatoren. Nun wird die ausgeführte Ebenentransformationsdefinition ETD_i aus der Datenstruktur gelöscht und die nun am höchsten gewichtete Ebenentransformationsdefinition mit der ersetzten Transformationsdefinition TD ausgeführt. Dies wird solange wiederholt, bis die Datenstruktur leer ist. Erst dann wird die daraus resultierende Transformationsdefinition

TD auf mglw. während einer Ebenentransformation entstandene, höhere Operatoren untersucht, indem die SORT-Funktion erneut aufgerufen wird.

Durch das temporäre Zwischenspeichern entsprechender Informationen in einer Datenstruktur müssen i. d. R. nicht so viele Transformationsdefinitionen eingelesen werden und es wird der damit verbundene Mehraufwand verringert (sie z. B. Abbildung 5.8). In der Tat kann bei diesem `sort`-Algorithmus aber nicht mehr garantiert werden, dass die minimale Anzahl von Ebenentransformationen benötigt wird, da u. U. Ebenentransformationen wiederholt ausgeführt werden müssen. Im Abschnitt 5.3.5 wird ein Beispiel präsentiert, in dem dies genau der Fall ist. Dadurch kann der `sort`-Algorithmus nur für Operatoren uneingeschränkt angewendet werden, die bei der Ebenentransformation lokal ersetzt werden (siehe Tabelle 5.4). Zudem entstehen zusätzliche Kosten für die Verwaltung und vor allem Sortierung der notwendigen Datenstruktur.

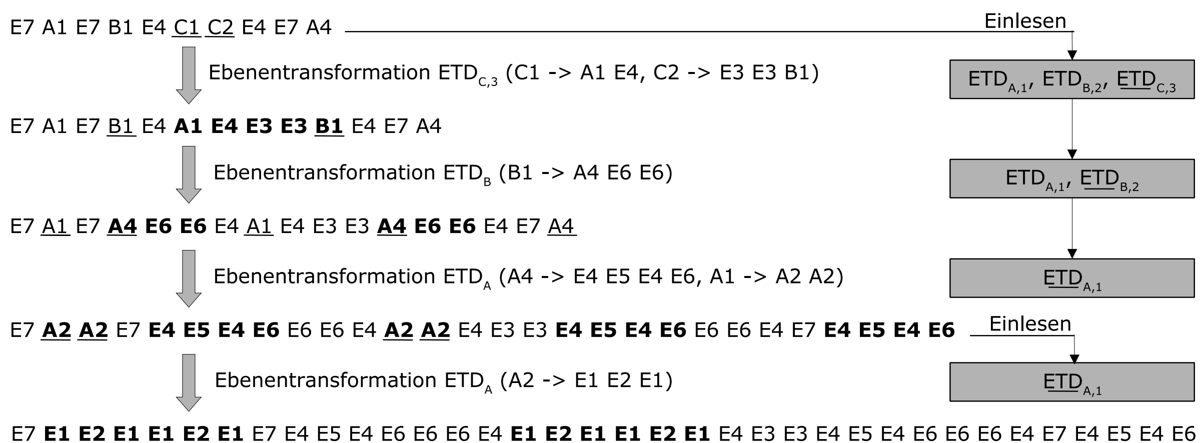


ABBILDUNG 5.8: Beispielanwendung des `sort`-Ebenentransformationsalgorithmus.

Die vorgestellten Algorithmen nehmen an, dass die Ebenentransformationsdefinition selbst direkt von einem Sprachprozessor ausgeführt werden kann. Da das Konzept der Operatorhierarchie ebenso für die Ebenentransformationsdefinitionen zum Einsatz kommen soll, müssen auch für diese Transformationsdefinitionen die Algorithmen rekursiv angewendet werden. Der entsprechende Ablauf wird im Zusammenhang mit der Realisierung der übrigen Komponenten im kommenden Abschnitt erläutert.

Für ein deterministisches Ergebnis des Ebenentransformationsprozesses sind für die vorgestellten Algorithmen nur bestimmte Merkmalskombinationen erlaubt. Tabelle 5.4 fasst sie jeweils für die vorgestellten Algorithmen zusammen. Ob ein Algorithmus uneingeschränkt genutzt werden kann oder nicht, ist dabei unabhängig davon, ob die Zieloperatoren bedingt oder unbedingt bei der Ebenentransformation erzeugt werden (Symbolisierung durch * in der Tabelle 5.4). Vielmehr hängt es von der Lokalität der Quellinformationen sowie der Zieloperatoren ab. Die Algorithmen `first` und `sort` können uneingeschränkt gebraucht werden, wenn die höheren Operatoren lokal an Ort und Stelle durch niedrigere ersetzt werden. Für die anderen Merkmalskombinationen müssen bestimmte zusätzliche Restriktionen erfüllt sein. Sind bspw. für eine Ebenentransformation eines Operators Informationen anderer Operatoren notwendig, so dürfen keine Operatoren

in den höheren Transformationsdefinitionen benutzt werden, die auf solchen Operatoren aufbauen. Beim `maxrank`-Algorithmus müssen solche Restriktionen nicht beachtet werden, da er auf einer Gewichtung der bzgl. der erzeugten Operatoren basiert. Da eben gerade die höher gewichteten Operatoren zuerst transformiert werden, stehen beim `maxrank`-Algorithmus immer alle notwendigen Informationen für eine Ebenentransformation auf Quellseite zur Verfügung. Es brauchen somit keine weiteren Metainformationen in Bezug auf die Ausführungsreihenfolge der Ebenentransformationsdefinitionen gegeben werden.

Merkmalskombination			Algorithmus		
Quellinformationen	**Zieloperatoren**	**Bedingtheit**	**first**	**maxrank**	**sort**
lokal	lokal	*	ja	ja	ja
global	lokal	*	nein	ja	nein
lokal	verteilt	*	nein	nein	nein
global	verteilt	*	nein	nein	nein

Tabelle 5.4: Uneingeschränkte Anwendbarkeit der Algorithmen `first`, `maxrank` und `sort` bzgl. der Unterscheidungsmerkmale von Operatorebenentransformationen.

Wie die Tabelle 5.4 zudem zeigt, kann keiner der vorgestellten Algorithmen uneingeschränkt verteilte Zieloperatoren erzeugen. Verteilte Zieloperatoren haben das Problem, dass sie bzgl. bestimmter Kontextoperatoren eingefügt werden. Diese Kontextoperatoren können aber beim Konzept der Operatorhierarchie bspw. durch andere Operatoren verkörpert oder so verändert werden, dass die Ausführungsreihenfolge der Ebenentransformationen wiederum signifikant ist und nicht über die Gewichtung der erzeugten Operatoren entschieden werden kann. Dieses Problem ist allerdings auch nicht vollautomatisch lösbar, wie analoge Probleme in der Aspektorientierung zeigen. In der Aspektorientierung werden Aspekte genutzt, um an sogenannten *Joinpoints* Codeschnipsel hinzuzufügen. Da der hinzuzufügende Code selbst potentielle *Joinpoints* enthalten kann, können Wechselwirkungen zwischen einzelnen Aspekten entstehen. Abhängig von der Reihenfolge, wie Aspekte eingewebt werden, können sich infolgedessen unterschiedliche Ergebnisse ergeben (siehe z. B. [LHBL06].) Es werden deshalb unterschiedliche Ansätze entwickelt, wie in die Ausführungsreihenfolge manuell eingegriffen werden kann, um diese zu steuern.

Die Ansätze können grob in interne und externe Priorisierung unterschieden werden. Bei der internen Priorisierung werden die Steuerungsinformationen im Aspekt angegeben. Dies unterstützen die meisten aspektorientierten Sprachen (z. B. AspectJ [Asp08]). Die Steuerungsinformationen können aber auch extern, außerhalb des Aspekts spezifiziert werden. Das Spektrum reicht dabei von expliziten, imperative Reihenfolgeangaben [Asp08] bis hin zu impliziten, deklarativen Beziehungsbeschreibungen, aus denen die Ausführungsreihenfolge errechnet wird (z. B. [NBA04, MW08]). Dabei müssen Algorithmen die Widerspruchsfreiheit der Beziehungsbeschreibungen überprüfen und sichern [RSB04, DFS04].

In XTC wird bisher eine externe Priorisierung mit Gewichtungen der Sprachkomponenten umgesetzt. Wie die Nutzung der prototypischen Implementierung zeigt, ist dieser Ansatz nicht optimal. Eine deklarative Beziehungsbeschreibung zwischen den Sprachkomponenten ist vermutlich vorzuziehen.

5.3 Realisierung

Zur Umsetzung des Operatorhierarchiekonzeptes wird die Ausführungsumgebung XTC (*XML Transformation Coordinator*) aufgebaut [FS06]. Sie bildet die notwendige Infrastruktur für die Automatisierung des mehrstufigen Ebenentransformationsprozesses. XTC besteht im Wesentlichen aus drei Hauptbestandteilen:

- das *Core System*, welches den gesamten Transformationsprozess steuert,
- die *Level Library*, welche eine Sammlung von Ebenentransformationsdefinitionen bereithält und
- Adapter, welche die Sprachprozessoren der verschiedenen Transformationssprachen in das *Core System* integrieren.

Im Folgenden werden diese Bestandteile einzeln vorgestellt, bevor auf die Interaktionen zwischen ihnen eingegangen wird.

5.3.1 Core System

Das *Core System* ist für den reibungslosen Ablauf des mehrstufigen Transformationsprozesses verantwortlich. Dieser beinhaltet u. U. die korrekten Aufrufe der Sprachprozessoren entsprechend der spezifischen Transformationsdefinitionen sowohl für die elementaren Transformationen als auch für die erforderlichen Ebenentransformationen. Mehrere Komponenten, dargestellt in Abbildung 5.9, sind bei der Durchführung dieser Aufgabe beteiligt. Die Wichtigsten sind:

- **Transformation Manager**
  Diese zentrale Komponente koordiniert und organisiert die Kommunikation zwischen allen anderen Komponenten und nach außen. Er kann von der Kommandozeile (CLI, *Command Line Interface*) oder von einer Java-Applikation über JAXP aufgerufen werden. JAXP ist eine Standardschnittstelle. Wird über diese API ein bestimmter Sprachprozessor in die Java-Applikation eingebunden, so kann dieser einfach durch XTC ausgetauscht werden, ohne Änderungen am Code vornehmen zu müssen (siehe Abschnitt 5.3.3 für weitere Informationen).
- **Input Handler**
  Der *Input Handler* lädt die über Parameter referenzierten Quelldokumente und Transformationsdefinitionen sowie, wenn nötig, die Ebenentransformationsdefinitionen von der *Level Library* und überführt diese in interne, verarbeitbare Datenströme.
- **Preprocessor**
  Der *Preprocessor* verarbeitet die vom *Input Handler* überführten Transformationsdefinitionen. Er sammelt Informationen über die verwendeten Namensräume. Diese Informationen werden benötigt, um auf Ebenentransformationsdefinitionen verweisen zu können.

- **Engine Loader**
  Der *Engine Loader* ist verantwortlich für das Aufrufen der Sprachprozessoren der einzelnen Transformationssprachen bzw. der entsprechenden Adapter. Die Inputdaten, wie die Quelldokumente und Transformationsdefinitionen, müssen dabei als Parameter übergeben werden.
- **Library Manager**
  Diese Komponente kann auf Metadaten über die Ebenentransformationsdefinitionen, die in der *Level Library* abgelegt sind, zugreifen. Mit diesen Metadaten können die richtigen Ebenentransformationsdefinitionen gefunden und deren Ausführungsreihenfolge bestimmt werden. Für bestimmte Merkmalskombinationen der Ebenentransformationsdefinition müssen die Metadaten manuelle Priorisierungen für eine korrekte Ausführungsreihenfolge enthalten (siehe Tabelle 5.4).

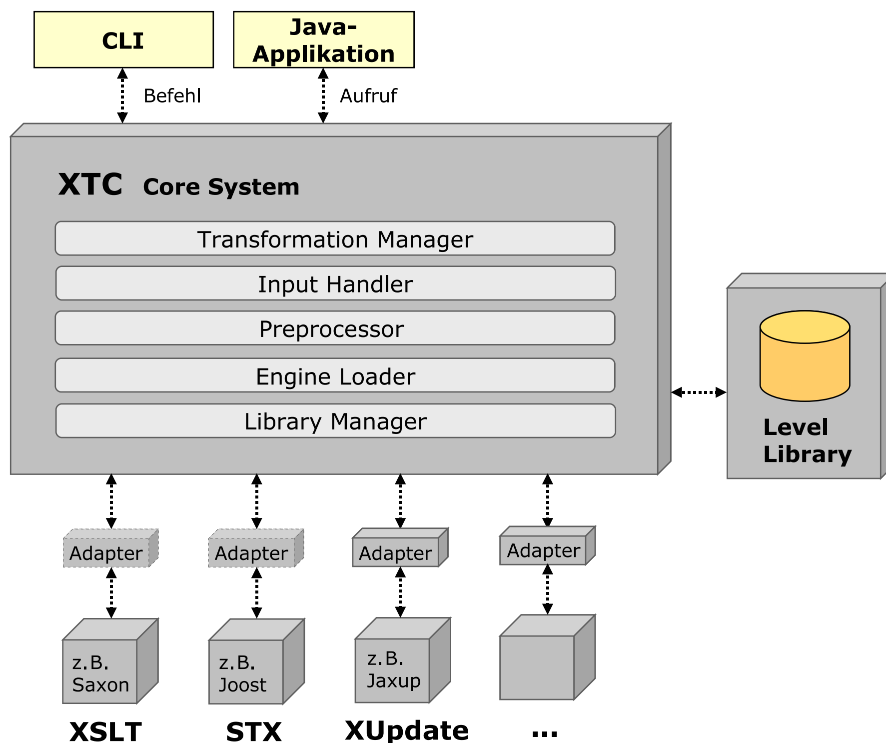


ABBILDUNG 5.9: Struktur von XTC.

5.3.2 Level Library

Die *Level Library* stellt eine Sammlung von wiederverwendbarem Code für wohldefinierte Probleme zur Verfügung. Sie bietet somit, wie andere Standardbibliotheken auch, eine thematisch gebündelte Menge an zusätzlichen Funktionalitäten, um gemeinsame Aufgaben zu realisieren. Im großen Unterschied zu klassischen Bibliotheken enthält die *Level Library* jedoch Ebenentransformationsdefinitionen, die die Überführung von höheren Transformationsoperatoren in niedrigere

spezifizieren, und somit Operatoren einer Sprachkomponente implementieren. Da verschiedene Transformationssprachen bzw. deren Sprachprozessoren in XTC integriert sind, können diese für die Beschreibung der Ebenentransformationen genutzt werden. Es kann somit für eine konkrete Ebenentransformation die bevorzugte Transformationssprache ausgewählt werden.

Ein weiterer wesentlicher Unterschied zu klassischen Bibliotheken besteht bei der Einbindung der Verweise auf die Ebenentransformationsdefinitionen in den höheren Transformationsdefinitionen. Diese werden nicht, wie üblich, durch eigene Anweisungen eingebunden (z. B. `include` in C++ oder `import` in Java). Mehrere Argumente sprechen dagegen:

- Es treten Konflikte mit herkömmlichen Einbindungsmechanismen auf (z. B. `import` in XSLT), wenn sie (zufällig) die gleiche Signatur verwenden (gleicher Name des Operators und gleiche Anzahl, Namen sowie Typen der Operatorparameter).
- Es treten ebenso Konflikte bei höheren Operatoren auf, falls diese (zufällig) die gleiche Signatur besitzen, bspw. wenn mehrere Ebenentransformationsdefinitionen eingebunden werden.
- Um die Basistransformationssprachen nicht erweitern zu müssen, muss der zusätzlich benötigte Einbindungsoperator in den höheren Transformationsdefinitionen bei deren Ebenentransformation wieder gelöscht werden. Dies erfordert einen weiteren Vorverarbeitungsprozess.

In XTC erfolgen deshalb die Verweise auf die Sprachkomponenten bzw. ihrer Ebenentransformationsdefinitionen mit Hilfe des Namensraummechanismus. Zunächst wird jeder Ebenentransformationsdefinition eine URI zugeordnet. Um höhere Operatoren in einer Transformationsdefinition zu nutzen, müssen sie nun an gerade den mit dieser URI deklarierten Namensraum gebunden werden. Die Zuweisung des Namensraums kann bspw. mit einem entsprechenden Präfix erfolgen (siehe Abschnitt 2.2.4). Die höheren Operatoren werden somit indirekt über die URIs an ihre Ebenentransformationsdefinitionen gekoppelt. Einerseits wird dadurch jeglichen Namenskonflikten aus dem Weg gegangen und andererseits werden keine zusätzlichen Operatoren benötigt.

Insgesamt bildet die *Level Library* die Basis, um Transformationen auf einer abstrakteren Ebene mit abstrakterer Syntax beschreiben zu können. Klassische Bibliotheken, die ebenso bei spezifischen XML-Transformationssprachen zu finden sind, definieren stattdessen wiederverwendbaren Code mit den Mitteln der Basissprache. Es werden keine neuen Operatoren als solches bereitgestellt.

5.3.3 Integration der Sprachprozessoren

Zur Integration wird eine standardisierte Schnittstelle (API) benötigt, die unabhängig von den verschiedenen Transformationssprachen und deren konkreten Implementierungen funktioniert. Auf diese Weise können leicht weitere Sprachen in XTC eingebunden werden. Zudem können die Sprachprozessoren, also die Implementation einer Transformationssprache, durch andere

ersetzt werden. Hierdurch werden Abhängigkeiten von einem bestimmten Produkt oder einem bestimmten Hersteller vermieden.

Für Java existiert ein solches API. Zu Beginn wurde es unter dem Namen TrAX (*Transformation API for XML*) entworfen. Seit der Version 1.1 ist es ein Bestandteil von JAXP (*Java API for XML Processing*). JAXP [Sun07] bietet neben TrAX eine Sammlung von weiteren Schnittstellen zum Validieren und Parsen von XML-Dokumenten. JAXP wurde erstmals mit der J2SE 1.4 [Sun03b] in die Standardbibliothek von Java aufgenommen.

TrAX wurde ursprünglich nur für die Sprache XSLT entwickelt. Eines der Ziele war es, eine Abstraktionsebene einzuführen, die es erlaubt, eine XSLT-Transformationsdefinition in eine Java-Applikation einzubinden, ohne sich um die konkreten XSLT-Prozessoren kümmern zu müssen. Man sollte, falls notwendig, die Sprachprozessoren einfach austauschen können. Durch ein modulares Framework sollte darüber hinaus eine größtmögliche Unabhängigkeit von der Art der XML-Quelle und der Art des XML-Ergebnisses erzielt werden. Dafür werden die Schnittstellen `Source` und `Result` im Paket `javax.xml.transform` definiert. In JAXP werden bspw. konkrete Klassen für Byte-Ströme (`StreamSource`, `StreamResult`), SAX-Datenströme (`SAXSource`, `SAXResult`), StAX-Datenströme (`StAXSource`, `StAXResult`) und DOM-Bäume (`DOMSource`, `DOMResult`) bereitgestellt. Andere APIs (z. B. dom4j [Str05] oder JDOM [HM07]) enthalten eigene Implementierungen.

```
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 transformationDefinition = args[1]; // die Transformationsdefinition
10     try
11     {
12       // Erzeugung einer Factory-Instanz
13       TransformerFactory tFactory = TransformerFactory.newInstance();
14
15       // Erzeugung eines Transformer-Objektes basierend auf einer Transformationsdefinition
16       Transformer transformer = tFactory.newTransformer(new StreamSource(transformationDefinition));
17
18       // Transformation einer Eingabe in eine Ausgabe (auf die Konsole)
19       transformer.transform(new StreamSource(sourceId),new StreamResult(System.out));
20     }
21     catch (TransformerException e)
22     {
23       // Fehlerbehandlung
24       System.err.println(e.getMessage());
25     }
26   }
27 }
```

CODE 5.2: Aufruf einer Transformation in TrAX.

Der Code 5.2 zeigt einen Beispielaufruf einer Transformation unter Nutzung von TrAX. Zunächst

werden die über die Kommandozeile mitgegebenen Namen der XML-Quelldatei und der Datei, die die Transformationsdefinition beinhaltet, zugewiesen. Anschließend werden innerhalb des `try`-Blockes die drei zentralen Schritte bei der Verwendung von TrAX ausgeführt: es wird eine `Factory` für `Transformer`-Objekte erzeugt (Zeile 13), diese legt anschließend unter Angabe der zu verwendenden Transformationsdefinitionen ein geeignetes `Transformer`-Objekt an (Zeile 16), welches schließlich die Transformation ausführt (Zeile 19).

Da, wie der Code 5.2 beispielhaft veranschaulicht, TrAX grundsätzlich jegliche Interna der Transformation verbirgt und infolgedessen keine XSLT-Spezifika aufweist, kann auch jede andere Transformationssprache gekapselt werden. So nutzt der Sprachprozessor Joost [Joo08] bspw. TrAX für die Transformationssprache STX [CBN+07].

Aufgrund dieser Eigenschaften wird TrAX auf beiden Seiten von XTC eingesetzt: auf der einen Seite zum Aufrufen von XTC aus einer Java-Applikation und auf der anderen Seite zur Integration der Prozessoren der unterschiedlichen Sprachen. Das Erstere ermöglicht, dass existierende Applikationen, die über die TrAX-Schnittstellen Transformationen ausführen, mit minimalem Aufwand auf XTC umgestellt werden können. Dafür müssen keine Änderungen im Quelltext vorgenommen werden, da in TrAX das *Factory Pattern* (siehe [GHJV95]) angewendet wird.

Um das *Factory Pattern* zu realisieren, wird in TrAX die Klasse `TransformerFactory` als abstrakt definiert. Sie kann somit nicht direkt instantiiert werden. Der Aufruf in Zeile 13 des Codes 5.2 liefert stattdessen ein Objekt vom Typ `TransformerFactory`. Von welcher konkreten abgeleiteten Klasse dieses Objekt eine Instanz ist, kann in Java über spezielle System-Properties konfiguriert werden. Diese können beim Starten einer Java-Applikation

```
java -D<name>=<value> ...
```

oder im Code der Java-Applikation

```
System.setProperty("<name>", "<value>");
```

gesetzt werden. Der Name des speziellen System-Propertys `<name>` für TrAX ist `javax.xml.transform.TransformerFactory` und der Wert `<value>` für XTC `net.sf.xtc.trax.TransformerFactoryImpl`. Ohne diese zusätzliche Konfiguration stellt Java 6 [Sun06] eine Referenzimplementierung bereit, bei der das `Factory`-Objekt einen XSLT-Prozessor enthält (siehe Abbildung 5.10). Durch das Festlegen dieser System-Property kann auf einfache Weise XTC anstatt der Referenzimplementierung in einer Java-Applikation verwendet werden.

In gleicher Weise können auf der anderen Seite mit Hilfe von TrAX unterschiedliche Sprachprozessoren durch XTC aufgerufen und damit integriert werden. Die Auswahl der Sprachprozessoren erfolgt dabei dynamisch während der Programmausführung. Anhand der deklarierten Namensräume in den Transformationsdefinitionen entscheidet der *Engine Loader* zur Laufzeit, welcher Sprachprozessor für diese Transformation genutzt wird. Die erforderlichen Metadaten über die Sprachprozessoren (unterstützte Namensräume und Ort der Transformationskomponente) werden dem *Engine Loader* zuvor übergeben. Der Codeausschnitt 5.3 zeigt schemenhaft die benötigten Teilschritte.

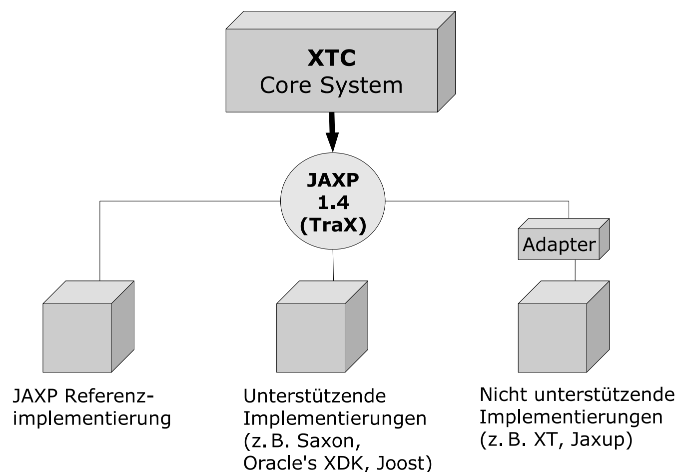


ABBILDUNG 5.10: Integration von Sprachprozessoren.

```
1  Hashtable registeredProcessors = getRegisteredProcessors(PROCCESSOR_FILE_ID);
2
3  public void transform(Source transformationDefinition, Source source, Result result)
4  {
5    // Laden der deklarierten Namensräume der Transformationsdefinition
6    Hashtable usedNamespacesInTD = getUsedNamespacesInTD(transformationDefinition);
7    String namespace = null;
8
9    // Suchen nach einem unterstützten Namensraum
10   for (Enumeration e = usedNamespacesInTD.elements(); e.hasMoreElements(); )
11   {
12     Object namespaceTD = e.nextElement();
13     if (registeredProcessors.containsKey(namespaceTD))
14     {
15       namespace = (String)namespaceTD;
16       break;
17     }
18   }
19
20   // Setzen des entsprechenden Property-Wertes
21   System.setProperty("javax.xml.transform.TransformerFactory",
22                      (String) registeredProcessors.get(namespace));
23   try
24   {
25     TransformerFactory tFactory = TransformerFactory.newInstance();
26     Transformer transformer = tFactory.newTransformer(transformationDefinition);
27     transformer.transform(source, result);
28   }
29   //...
30 }
```

CODE 5.3: Dynamische Auswahl der integrierten Sprachprozessoren.

Einige Sprachprozessoren unterstützen TrAX nicht (vgl. Abbildung 5.10). Um diese in XTC integrieren zu können, werden Adapter benötigt, welche die inkompatiblen Schnittstellen überbrücken. Ruft der *Engine Loader* Operationen einer solchen Adapterinstanz auf, delegiert der Adapter diese Anfrage weiter und ruft seinerseits die entsprechenden Operationen der Sprachprozessoren auf. Der Adapter adaptiert dadurch die Schnittstelle des Sprachprozessors, so dass

XTC diesen nutzen kann. Das *Adapter Pattern* (siehe [GHJV95]) ermöglicht somit trotz inkompatibler Schnittstellen eine Kommunikation zwischen den beiden Komponenten.

5.3.4 Funktionsweise

Jeder Transformationsprozess in XTC beginnt mit der Bereitstellung von mindestens einer XML-Quelle sowie einer Transformationsdefinition, unabhängig davon, ob XTC über die Kommandozeile (CLI) oder von einer Java-Applikation über JAXP initiiert wird (siehe Abbildung 5.11). Der *Input Handler* lädt die Inputs und überführt sie in eine für den weiteren Transformationsprozess verarbeitbare Form. Im nächsten Transformationsschritt protokolliert der *Preprocessor* sämtliche Namensräume, die in der Transformationsdefinition deklariert werden. Anschließend werden diese Namensräume mit den URIs der registrierten Ebenentransformationsdefinitionen verglichen, die vom *Library Manager* bereitgestellt werden und sich physisch in der *Level Library* befinden.

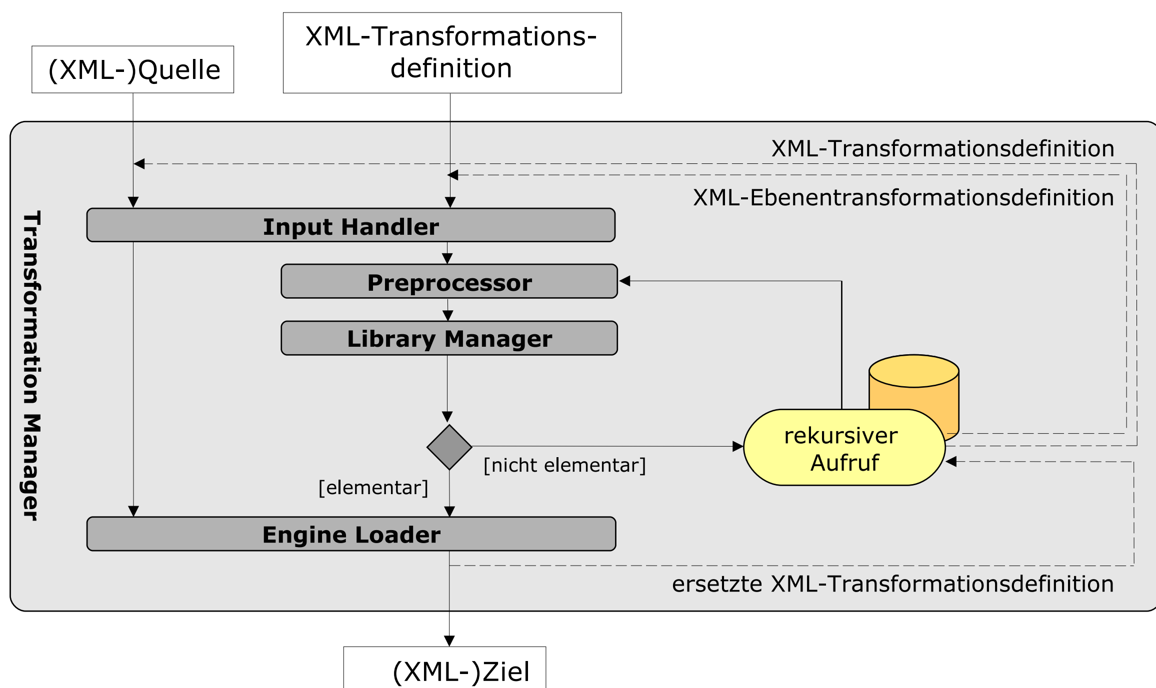


Abbildung 5.11: Zusammenspiel der Komponenten von XTC.

Existieren keine übereinstimmenden URIs, handelt es sich um eine elementare Transformation. Mit den Metadaten über die registrierten Sprachprozessoren entscheidet nun der *Engine Loader*, welcher Sprachprozessor diese elementare Transformation ausführt. Das dem *Engine Loader* zurückgelieferte Ergebnis wird durch den *Transformation Manager* nach außen gegeben. Wird kein passender Sprachprozessor über den Abgleich der URIs gefunden, wird eine Fehlermeldung generiert.

Existieren jedoch übereinstimmende URIs, sind per Definition in der Transformationsdefinition höhere Transformationsoperatoren verwendet worden. In diesem Fall werden die höheren Transformationsoperatoren mit Hilfe der referenzierten Ebenentransformationsdefinitionen in niedrigere, mglw. elementare Operatoren überführt. Der *Transformation Manager* veranlasst einen demgemäß rekursive Aufrufe des Gesamtprozesses. Bei einem rekursiven Aufruf ist die Transformationsdefinition die Quelle und die erste referenzierte Ebenentransformationsdefinition die neue Transformationsdefinition. Als Ergebnis liefert diese (rekursive) Ebenentransformation eine ersetzte Transformationsdefinition, die wiederum Ausgangspunkt für die nächste identifizierte Ebenentransformation ist.

Sind alle bisher identifizierten Ebenentransformationsdefinitionen abgearbeitet, wird nun die ursprüngliche Transformation mit dieser ersetzten Transformationsdefinition fortgesetzt. Dazu wird die ersetzte Transformationsdefinition erneut auf Vorhandensein von höheren Operatoren überprüft. Dieser Ablauf wird solange wiederholt bis die Transformationsdefinition keine höheren Transformationsoperatoren mehr besitzt und eine elementare Transformation ausgeführt werden kann.

Da in den Ebenentransformationsdefinitionen ebenso höhere Transformationsoperatoren genutzt werden können, können auch hier Ebenentransformationen notwendig sein. Abbildung 5.12 zeigt beispielhaft einen schematischen Transformationsprozess in XTC.

ABBILDUNG 5.12: Beispielablauf eines Transformationsprozesses in XTC.

Im ersten Schritt wird XTC mit einer Quelle und einer Transformationsdefinition gestartet. Da die Transformationsdefinition höhere Operatoren enthält, deren URI in der *Level Library* von XTC registriert werden, werden in den rekursiven Aufrufen (Schritt 2 bis 4) entsprechende

Ebenentransformationen durchgeführt. Die Ebenentransformationsdefinitionen bestehen dabei nur aus elementaren Operatoren und können infolgedessen direkt mit einem Sprachprozessor ausgeführt werden. Für die Ebenentransformation können verschiedene Sprachen bzw. deren registrierte Sprachprozessoren in XTC eingesetzt werden. Durch die Ebenentransformationen werden aus der ursprünglichen Transformationsdefinition aus Schritt 1 die ersetzten Transformationsdefinitionen. Zur Unterscheidung werden diese jeweils mit einem Stern gekennzeichnet: 1*, 1** und 1***. Da immer noch nicht alle höheren Operatoren der ursprünglichen Transformationsdefinition (Schritt 1***) in elementare Operatoren transformiert wurden, muss eine weitere Ebenentransformation ausgelöst werden (Schritt 5). Die erforderliche Ebenentransformationsdefinition wurde dabei mit höheren Operatoren beschrieben. Um sie mit einem Sprachprozessor verarbeiten zu können, wird daher ein erneuter Aufruf (Schritt 6) veranlasst, der für diese Transformationsdefinition eine Ebenentransformation initiiert. Diese eben beschriebene Kette von Rekursionen in Richtung der Ebenentransformationsdefinitionen kann sich fortsetzen. Dadurch entstehen in zwei Dimensionen rekursive Aufrufe. Die in Schritt 6 benötigte Ebenentransformationsdefinition muss ihrerseits durch Ebenentransformationen (Schritt 7 und 8) in eine elementare Transformationsdefinition übersetzt werden. Erst jetzt können die Ebenentransformationen von Schritt 6** und Schritt 5* sowie die eigentliche Transformation des ersten Schrittes durch entsprechende Sprachprozessoren ausgeführt und das Ergebnis von XTC ausgegeben werden.

5.3.5 Experimentelle Untersuchungen

Im *Transformation Manager* wurden die drei im Abschnitt 5.2 vorgeschlagenen Algorithmen `first`, `maxrank` und `sort` zur Steuerung des Ebenentransformationsprozesses implementiert. Die bevorzugte Ausführungsstrategie kann mittels Parameter beim Aufruf von XTC gesetzt werden. Eine nachträgliche Konfiguration ist ebenso möglich.

Im Folgenden werden die drei umgesetzten Algorithmen verglichen. Da die Effizienz eines Algorithmus von einer Vielzahl von Einflussgrößen abhängt (siehe Liste auf Seite 97), werden unterschiedliche Experimente auf einer isolierten Messumgebung durchgeführt. Als Messumgebung dient ein Rechner mit Intel® Core™ Duo, 2GHz und 2GB Arbeitsspeicher. Als Betriebssystem wird Windows XP mit installierter JRE 6.0 eingesetzt. Die Transformationsdefinitionen wurden mit XSLT implementiert und mit Hilfe des Sprachprozessors Saxon 9.0 [Kay08] ausgeführt, der AElfred 7.0 [Meg02] als XML-Parser einsetzt.

Untere Schranke des first-Algorithmus

Im ersten Experiment soll die Operatorhierarchie aus Abbildung 5.6 als Beispielhierarchie dienen. Mit diesem Experiment soll veranschaulicht werden, dass der einfache `first`-Algorithmus auch praktisch signifikant langsamer als die anderen beiden Algorithmen sein kann. Um dies eindrucksvoll zu verdeutlichen, wird die Operatorhierarchie von Abbildung 5.6 auf 20 Ebenen erweitert. Die höheren Operatoren werden dabei nach dem gleichen Schema auf Operatoren niedrigerer Ebenen überführt, wie in Abbildung 5.6 gezeigt.

$$
\begin{aligned}
\mathrm{ETD_D} &= \{\mathrm{D} \Rightarrow \mathrm{E}\}\\
\mathrm{ETD_C} &= \{\mathrm{C} \Rightarrow \mathrm{D}\}\\
\mathrm{ETD_B} &= \{\mathrm{B} \Rightarrow \mathrm{D\ C}\}\\
\mathrm{ETD_A} &= \{\mathrm{A} \Rightarrow \mathrm{D\ C\ B}\}\\
\mathrm{ETD_Z} &= \{\mathrm{Z} \Rightarrow \mathrm{D\ C\ B\ A}\}\\
\mathrm{ETD_Y} &= \{\mathrm{Y} \Rightarrow \mathrm{D\ C\ B\ A\ Z}\},\ usw.
\end{aligned}
$$

Insgesamt werden somit 20 Ebenentransformationsdefinitionen mit jeweils gleicher Dateigröße (40,36 KByte) entwickelt bzw. generiert. Die Dateigröße wird exakt gleich lang gewählt, um Verzerrungen durch unterschiedliche Geschwindigkeiten beim Parsen der Ebenentransformationsdefinitionen zu minimieren.

Die folgende Tabelle 5.5 zeigt die gemessenen Eingaben. Die Operatoren in den Eingaben sind so gewählt, dass sie bzgl. ihrer Gewichtung aufsteigend sortiert sind. Da in diesen Fällen der `first`-Algorithmus wiederholt Operatoren niedrigerer Ebenen ausführt, benötigt er $2^n - 1$ Ebenentransformationen und arbeitet dementsprechend gegenüber dem `maxrank`- und dem `sort`-Algorithmus sehr ineffizient.

Nummer n	Eingabe
1	D
3	D C B
5	D C B A Z
7	D C B A Z Y X
9	D C B A Z Y X W V
12	D C B A Z Y X W V U T S
15	D C B A Z Y X W V U T S R Q P
18	D C B A Z Y X W V U T S R Q P O N M
20	D C B A Z Y X W V U T S R Q P O N M L K

TABELLE 5.5: Eingaben für die Beispielmessungen (1. Experiment).

Tabelle 5.6 gibt beispielhaft die Messungen für $n = 3$ an. Schon bei diesen wenigen Operatorenebenen sind die Unterschiede in den Laufzeiten merkbar. Aufsummiert (netto) fallen sie jedoch geringer aus als erwartet. Dies liegt daran, dass der Sprachprozessor Saxon für die erste Ebenentransformation beträchtlich mehr Zeit u. a. für deren Initialisierung benötigt.

Betrachtet man neben der reinen Laufzeit für die Ebenentransformationen (Summe netto) ebenso die benötigte Laufzeit für die Steuerung der Ebenentransformationsaufrufe innerhalb von XTC (Summe brutto), so reduziert sich der prozentuale Laufzeitverlust des `first`-Algorithmus gegenüber den beiden anderen Algorithmen geringfügig. Diese Beobachtung lässt sich aus dem zusätzlichen Aufwand für das Finden des Maximums (`maxrank`) oder für das Sortieren (`sort`) begründen.

Da die Anzahl der benötigten Ebenentransformationen beim `first`-Algorithmus jedoch bei den Eingaben von Tabelle 5.5 exponentiell wächst, können die zusätzlichen Kosten bei den anderen

	Laufzeiten in ms (Ebenentransformationsdefinition)		
	first	**maxrank**	**sort**
1. Ebenentransformation	496,53 (ETD_D)	492,62 (ETD_B)	489,64 (ETD_B)
2. Ebenentransformation	100,93 (ETD_C)	99,18 (ETD_C)	99,59 (ETD_C)
3. Ebenentransformation	75,50 (ETD_D)	76,84 (ETD_D)	77,92 (ETD_D)
4. Ebenentransformation	68,62 (ETD_B)		
5. Ebenentransformation	95,09 (ETD_D)		
6. Ebenentransformation	63,24 (ETD_C)		
7. Ebenentransformation	58,52 (ETD_D)		
Durchschnitt	136,92	222,84	222,38
Median	75,50	99,18	99,59
Summe (netto)	958,43	668,53	667,15
Summe (brutto)	1.086,96	780,07	776,58

Tabelle 5.6: Übersicht der Einzelwertmessungen der Ebenentransformationen für n=3 der Algorithmen `first`, `maxrank` und `sort` (1. Experiment).

beiden Algorithmen vernachlässigt werden. Zumal durch diesen Aufwand ein lineare Abhängigkeit erreicht werden kann. Die Abbildung 5.13, in der die Bruttogesamtlaufzeiten der Algorithmen für den Ebenentransformationsprozess erfasst werden, zeigt dies unübersehbar. Lediglich bei einer Hierarchieebene ($n = 1$) wäre der `first`-Algorithmus zu bevorzugen.

Trotz linearem Anwachsen der Ebenentransformationen (z. B. bei der Eingabe n=20 sind 20 Ebenentransformationen erforderlich) entwickelt sich die Laufzeit der Algorithmen `maxrank` und `sort` ebenso exponentiell (siehe Abbildung 5.13). Der Grund für dieses schnelle Wachstum ist bei den Ebenentransformationsdefinitionen von Seite 112 selbst zu finden. Operatoren auf n-ter Ebene erzeugen indirekt 2^{n-1} elementare Operatoren. Das Einlesen und Ausgeben der Zwischenergebnisse während des Ebenentransformationsprozesses verbraucht somit die notwendige Zeit.

Untere Schranke des maxrank-Algorithmus

Mit dem zweiten Experiment soll der schlechteste Fall des `maxrank`-Algorithmus betrachtet werden. Dieser Fall zeichnet sich durch eine absteigende Gewichtung der verwendeten Operatoren in der übergebenen, höheren Transformationsdefinition aus, die ebenfalls in allen Zwischenresultaten (ersetzten Transformationsdefinitionen) gegeben sein muss. Damit diese Einschränkung erfüllt ist, muss zum einen die Eingabe bzgl. der Gewichtungen der höheren Operatoren absteigend sortiert sein. Zum anderen müssen die Ebenentransformationsdefinitionen selbst Operatoren mit absteigender Gewichtung erzeugen.

Im zweiten Experiment werden die folgenden Transformationsregeln in den Ebenentransformationsdefinitionen implementiert, die den obigen Bedingungen genügen. Der Operator E ist wiederum ein elementarer Operator und alle anderen Operatoren sind nichtelementare Operatoren.

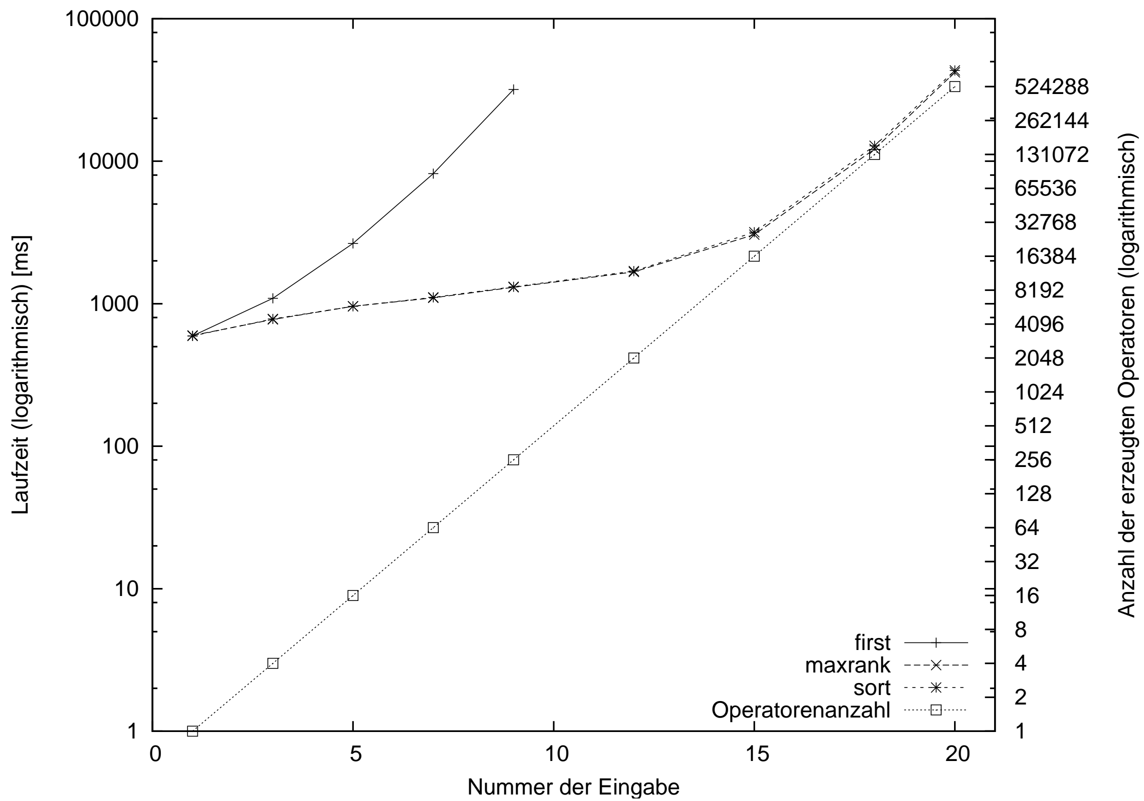


ABBILDUNG 5.13: Laufzeiten der Ebenentransformationsprozessalgorithmen `first`, `maxrank` und `sort` im Vergleich (1. Experiment).

$$\begin{aligned}
\mathrm{ETD_D} &= \{\mathrm{D} \Rightarrow \mathrm{E}\}\\
\mathrm{ETD_C} &= \{\mathrm{C} \Rightarrow \mathrm{D}\}\\
\mathrm{ETD_B} &= \{\mathrm{B} \Rightarrow \mathrm{C}\}\\
\mathrm{ETD_A} &= \{\mathrm{A} \Rightarrow \mathrm{B}\}\\
\mathrm{ETD_Z} &= \{\mathrm{Z} \Rightarrow \mathrm{A}\}\\
\mathrm{ETD_Y} &= \{\mathrm{Y} \Rightarrow \mathrm{Z}\},\ usw.
\end{aligned}$$

Dieses zweite Experiment soll zeigen, dass der `maxrank`-Algorithmus in dem für ihn ungünstigsten Fall nicht erheblich langsamer ist, als der `first`-Algorithmus. Dabei muss erwähnt werden, dass diese spezielle Konstellation sogleich der optimale Fall des `first`-Algorithmus ist. Der `maxrank`-Algorithmus durchläuft hier unnötigerweise jeweils die gesamten höheren Transformationsdefinitionen im Präprozess, um das Maximum zu finden. Letztlich wird jedoch immer die Ebenentransformationsdefinition vom ersten höheren Operator ausgeführt, weil diese der maximalen Gewichtung entspricht.

Für die Durchführung des Experimentes werden erneut insgesamt 20 Ebenentransformationsdefinitionen mit wiederum gleicher Dateigröße (40,36 KByte) entwickelt bzw. generiert. Die Zeitmessung wird an folgenden Eingaben vorgenommen (siehe Tabelle 5.7).

Nummer n	**Eingabe**
1	D
3	B C D
5	Z A B C D
7	X Y Z A B C D
9	V W X Y Z A B C D
12	S T U V W X Y Z A B C D
15	P Q R S T U V W X Y Z A B C D
18	M N O P Q R S T U V W X Y Z A B C D
20	K L M N O P Q R S T U V W X Y Z A B C D

Tabelle 5.7: Eingaben für die Beispielmessungen (2. Experiment).

Tabelle 5.8 gibt exemplarisch die Messungen für $n = 5$ an. Da die Eingaben für den `first`-Algorithmus optimal sind, benötigt er genauso viele Ebenentransformationen, wie die anderen beiden Algorithmen. Aus diesem Grund ist er für alle Eingaben aus Tabelle 5.7 schneller als der `maxrank`-Algorithmus. Gegenüber dem `sort`-Algorithmus ist der `first`-Algorithmus in diesem Experiment allerdings bereits ab der Eingabe $n = 3$ langsamer (vgl. Abbildung 5.14). Der Grund liegt in der Eingabeoptimierung des `sort`-Algorithmus.

Unabhängig davon, zeigt die Abbildung 5.14, dass im zweiten Experiment (im Gegensatz zum ersten Experiment) bei allen Algorithmen die Laufzeit linear zu n ansteigt. Der Grund liegt darin, dass in diesem Experiment nicht die Anzahl der erzeugten Operatoren mit größer werdendem n exponentiell zunimmt, sondern vielmehr linear zur Eingabe wächst.

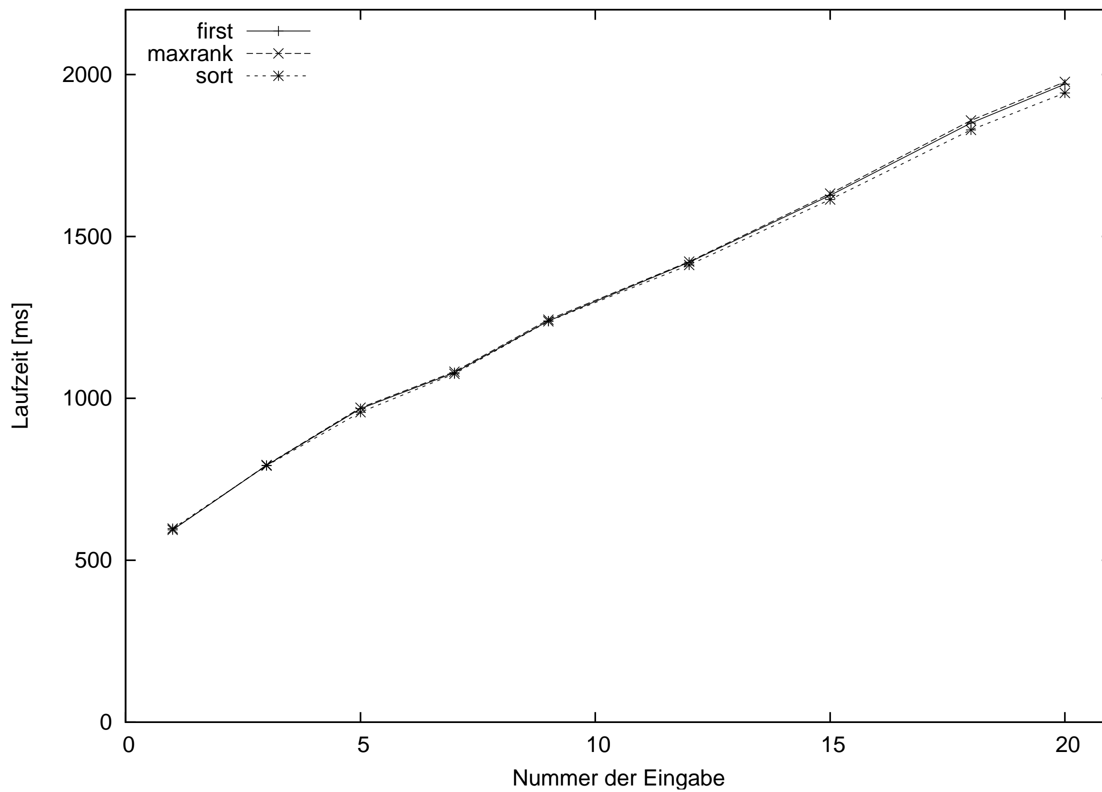


ABBILDUNG 5.14: Laufzeiten der Ebenentransformationsprozessalgorithmen `first`, `maxrank` und `sort` im Vergleich (2. Experiment).

	Laufzeiten in ms (Ebenentransformationsdefinition)		
	first	**maxrank**	**sort**
1. Ebenentransformation	496,65 (ETD_Z)	497,44 (ETD_Z)	496,04 (ETD_Z)
2. Ebenentransformation	101,65 (ETD_A)	102,22 (ETD_A)	101,39 (ETD_A)
3. Ebenentransformation	78,65 (ETD_B)	77,60 (ETD_B)	80,60 (ETD_B)
4. Ebenentransformation	69,35 (ETD_C)	70,92 (ETD_C)	70,47 (ETD_C)
5. Ebenentransformation	97,32 (ETD_D)	96,35 (ETD_D)	94,81 (ETD_D)
Durchschnitt	168,72	168,90	168,66
Median	97,32	96,35	94,81
Summe (netto)	843,62	844,52	843,30
Summe (brutto)	970,58	974,18	963,89

Tabelle 5.8: Übersicht der Einzelwertmessungen der Ebenentransformationen für n=5 der Algorithmen `first`, `maxrank` und `sort` (2. Experiment).

Vergleicht man den `maxrank`-Algorithmus mit dem `sort`-Algorithmus in den bisherigen Experimenten genauer, so kann festgestellt werden, dass bisher der `sort`-Algorithmus mit Ausnahme jeweils der Eingabe $n = 1$ schneller ist als `maxrank`. Je größer n und somit je mehr Ebenentransformationen erforderlich sind, umso größer der Zeitgewinn des `sort`-Algorithmus. Dies ist folgerichtig, da dieser Algorithmus nicht jedes Mal die Zwischenresultate (ersetzten Transformationsdefinitionen) nach dem am höchsten gewichteten Operator bzw. dessen zugeordneten Ebenentransformationsdefinition durchsucht.

Untere Schranke des sort-Algorithmus

In diesem dritten Experiment soll gezeigt werden, dass es auch Fälle gibt, in denen sowohl der `maxrank`-Algorithmus als auch der `first`-Algorithmus beträchtlich weniger Laufzeit benötigen als der `sort`-Algorithmus. Um dies zu veranschaulichen, können die Ebenentransformationsdefinitionen vom zweiten Experiment auf Seite 115 erneut genutzt werden. Es müssen lediglich die Eingaben so modifiziert werden, dass sie die Schwachpunkte des `sort`-Algorithmus aufdecken. Der Algorithmus nimmt an, dass in der Vielzahl der Fälle sich schon in der übergebenen Transformationsdefinition annähernd alle höheren Operatoren befinden und nicht erst während des Ebenentransformationsprozesses erzeugt werden. Der `sort`-Algorithmus verzichtet ebendeswegen darauf jedes Zwischenresultat (ersetzte Transformationsdefinition) einzulesen und auf höhere Operatoren zu prüfen (wie z. B. der `maxrank`-Algorithmus). Werden bei den Eingaben bewusst Lücken gelassen, indem bspw. Operatoren mit nur geraden oder nur ungeraden Gewichtungen gewählt werden (siehe Tabelle 5.9), entgehen dem `sort`-Algorithmen viele hoch gewichtete Operatoren. Bei deren Transformation in niedrigere Operatoren werden dann zwangsläufig niedrig gewichtete Ebenentransformationen wiederholt ausgeführt.

Für die Eingaben aus Tabelle 5.9 kann errechnet werden, dass der `sort`-Algorithmus $\lceil n/2 \rceil (n - \lceil n/2 \rceil + 1)$ Ebenentransformationen verlangt, wobei n die Nummer der Eingabe bezeichnet. Der `first`-Algorithmus und der `maxrank`-Algorithmus brauchen hingegen bloß n Ebenentransformationen. Dieser Unterschied spiegelt sich in den Laufzeiten wider. Tabelle 5.10 gibt eine Übersicht über die Messdaten und deren Durchschnitte, Mediane und Summen für $n = 5$.

Nummer n	Eingabe
1	D
3	B D
5	Z B D
7	X Z B D
9	V X Z B D
12	S U W Y A C
15	P R T V X Z B D
18	M O Q S U W Y A C
20	K M O Q S U W Y A C

TABELLE 5.9: Eingaben für die Beispielmessungen (3. Experiment).

	Laufzeiten in ms (Ebenentransformationsdefinition)		
	first	maxrank	sort
1. Ebenentransformation	489,70 (ETD_Z)	489,48 (ETD_Z)	487,08 (ETD_Z)
2. Ebenentransformation	98,97 (ETD_A)	98,81 (ETD_A)	98,60 (ETD_B)
3. Ebenentransformation	77,58 (ETD_B)	78,89 (ETD_B)	78,15 (ETD_D)
4. Ebenentransformation	67,64 (ETD_C)	68,10 (ETD_C)	68,50 (ETD_A)
5. Ebenentransformation	94,19 (ETD_D)	93,32 (ETD_D)	92,03 (ETD_C)
6. Ebenentransformation			62,53 (ETD_B)
7. Ebenentransformation			59,29 (ETD_D)
8. Ebenentransformation			59,10 (ETD_C)
9. Ebenentransformation			84,37 (ETD_D)
Durchschnitt	165,62	165,72	164,87
Median	98,97	93,32	92,03
Summe (netto)	828,09	828,60	946,18
Summe (brutto)	949,91	954,05	1.220,28

TABELLE 5.10: Übersicht der Einzelwertmessungen für n=5 der Algorithmen `first`, `maxrank` und `sort` (3. Experiment).

Zum Vergleich der Laufzeiten für die andere Eingaben aus Tabelle 5.9 wird auf die Abbildung 5.15 verwiesen.

Fazit

Die experimentellen Untersuchungen bestätigen, dass der `maxrank`-Algorithmus immer die minimale Anzahl von Ebenentransformationen benötigt. Diese Optimalität erkauft er sich mit vergleichsweisen geringen zusätzlichen Laufzeitkosten gegenüber dem `first`-Algorithmus bzw. dem `sort`-Algorithmus. Aus diesem Grund wird der `maxrank`-Algorithmus als Standardstrategie in XTC gewählt. Der `sort`-Algorithmus sollte bevorzugt werden, wenn viele höhere Operatoren aus verschiedenen Sprachkomponenten zum Einsatz kommen. Je mehr höhere Operatoren bereits die übergebene Transformationsdefinition enthält und je weniger höhere Operatoren erst während des Ebenentransformationsprozesses erzeugt werden, umso vorteilhafter ist der `sort`-Algorithmus. Dies gilt insbesondere, wenn es sich um eine große Transformationsdefinition handelt. Der `first`-Algorithmus sollte den Vorrang erhalten, wenn von vornherein feststeht, dass nur höhere Operatoren einer Sprachkomponente oder nicht korrelierende Operatoren Anwendung finden, bei denen die Reihenfolge der Ebenentransformationen nicht entscheidend für die Anzahl der Ebenentransformationen ist.

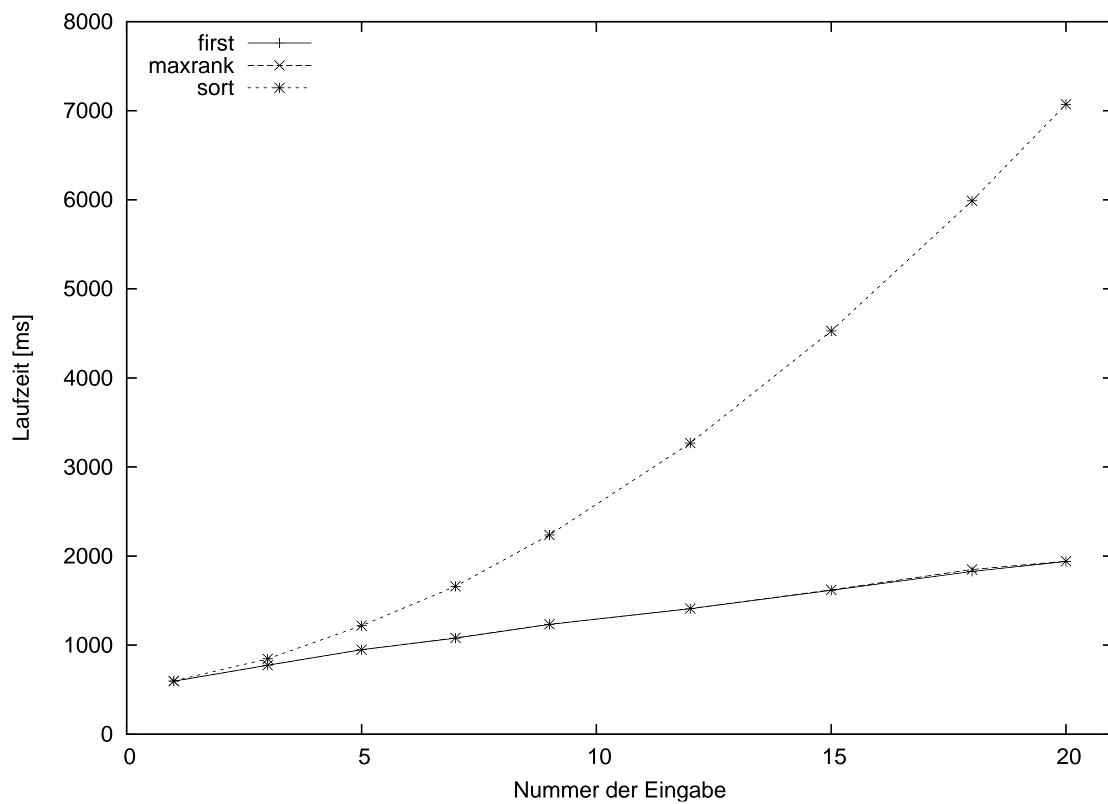


ABBILDUNG 5.15: Laufzeiten der Ebenentransformationsprozessalgorithmen `first`, `maxrank` und `sort` im Vergleich (3. Experiment).

Kapitel 6

Generative Techniken

In diesem Kapitel werden die konzeptionellen Grundlagen und die Realisierung des Generatorsystems XOpGen vorgestellt. Aufgabe von XOpGen ist die Generation von Validierungskomponenten u. a. für die syntaktische Überprüfung von höheren Operatoren. Die generierten Validierungskomponenten werden in die Infrastruktur der Ebenentransformationsdefinitionen integriert (siehe Abbildung 6.1). Ziel ist es, den Sprachentwickler bei der Implementierung von höheren Operatoren zu entlasten.

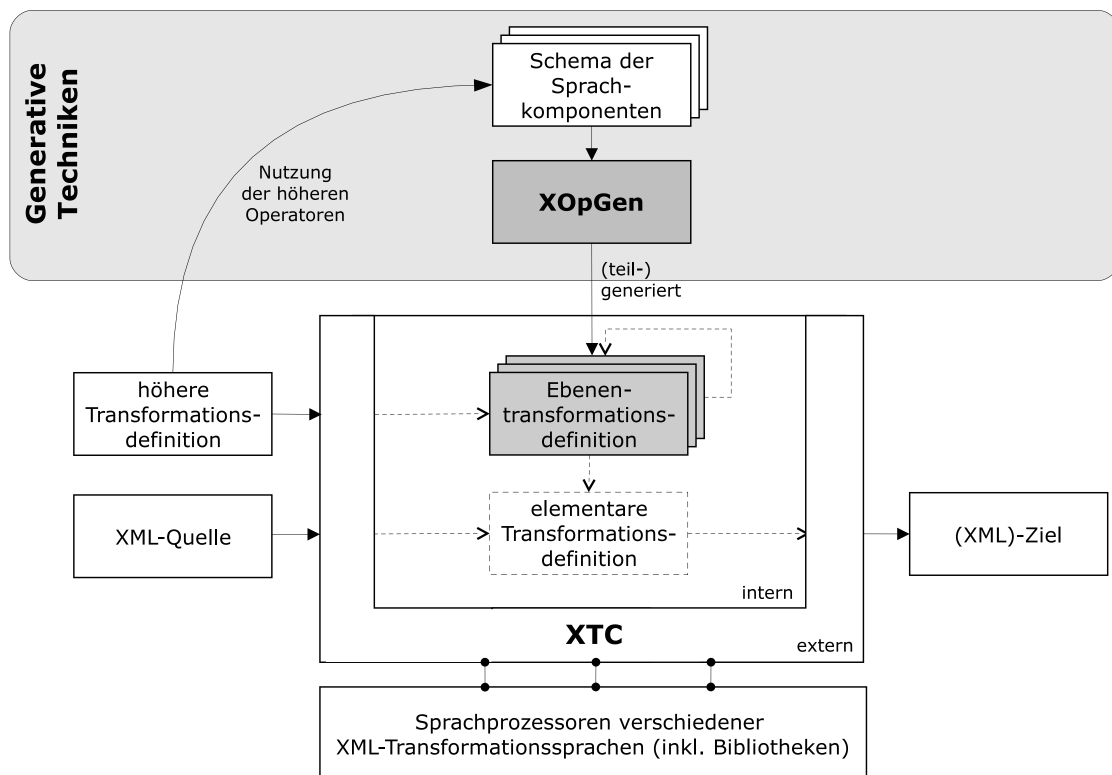


Abbildung 6.1: Einordnung des Generatorsystems XOpGen in die Gesamtstruktur.

Im ersten Abschnitt dieses Kapitels wird eine konkrete Umsetzung einer Operatorvalidierung beschrieben. Anschließend wird ein Konzept vorgeschlagen, welches den technischen, schematisch gleichen Code aus einer abstrakteren Strukturbeschreibung automatisch erzeugt. Die dafür

benötigten Generationsschritte werden im darauffolgenden Abschnitt näher erläutert. Der letzte Abschnitt stellt die dafür notwendigen Generatoren vor.

6.1 Einführung

Die für das Anwendungsszenario der Umstrukturierung vorgeschlagene Ebenentransformationsdefinition (Code 5.1) beschreibt, welche neu hinzugefügten, nicht elementaren Operatoren wie in elementare Operatoren überführt werden. Die restlichen, nicht erkannten Operatoren werden während der Ebenentransformation einfach kopiert. Ein wichtiger Aspekt wurde dabei bisher vernachlässigt: die Validierung der neuen Operatoren. Unter einer **Validierung** soll in dieser Arbeit der Vorgang der Überprüfung eines Dokumentes gegen eine bestimmte vorgeschriebene Spezifikation, welche Strukturen und Datentypen festlegt, verstanden werden. Ohne eine Validierung werden fehlerbehaftete Operatoren nicht oder falsch erkannt und interpretiert. Fehlerquellen können bspw.

- im Operatornamen (z. B. durch Schreib- oder Tippfehler),
- bei Attributen (z. B. zwingend erforderliche Attribute vergessen oder nicht erlaubte Attribute verwendet),
- bei statischen Attributwerten bzgl. eines bestimmten Attributtyps (z. B. bei Aufzählungstypen) oder
- durch Missachtung von Abhängigkeiten zwischen Attributen (z. B. mindestens ein Attribut von mehreren muss eingesetzt werden oder bei gegenseitig ausschließenden Attributen)

auftreten. Neben diesen lokalen Fehlerquellen innerhalb eines Operators können ebenso globale Abhängigkeiten und Integritäten zwischen Operatoren (z. B. Reihenfolge, Anzahl oder gegenseitiger Ausschluss) sowie zwischen deren Attributen (z. B. bestimmte Attributwertabhängigkeiten oder gegenseitiger Ausschluss) für eine korrekte Ausführung der Ebenentransformation vorausgesetzt sein.

Ein **Validierer**[1] ist eine Software, welche ein Dokument analysiert und Meldungen zur Konformität dieses Dokumentes bzgl. dessen Spezifikation liefert. Er setzt somit eine Validierung um. Die Beschreibung eines Validierers kann, wie die Ebenentransformationsdefinition selbst, mit Hilfe von XSLT realisiert werden. Der Code 6.1 zeigt exemplarisch eine Überprüfung der Syntax des `move-after`-Operators. In den Zeilen 8–13 und 14–19 wird demonstriert, wie getestet werden kann, ob die erforderlichen Attribute `from` und `to` existieren. Der Test in den Zeilen 20–27 überprüft, ob unerlaubte Attribute verwendet wurden. Die Zeilen 28–31 sichern, dass der `move-after`-Operator ein leeres Element ist. Hierdurch werden Kindelemente generell ausgeschlossen.

[1] Der Begriff Validierungskomponente wird verwendet, wenn betont werden soll, dass der Validierer Teil einer größeren Infrastruktur ist.

Wie der kleine Beispielcode 6.1 bereits verdeutlicht, ergeben sich bei dieser Implementierungsmethode mehrere Nachteile. Zum einen wird direkt festgelegt, wie beim Finden eines Fehlers die entsprechende Fehlermeldung erfolgt. Eine Umstellung auf einen anderen Fehlerbenachrichtigungsmechanismus, z. B. die Generation einer HTML-Dokumentation, oder die Umstellung der Sprache (z. B. auf Deutsch) ist dementsprechend schwierig. Zum anderen tritt immer wieder der gleiche schematische Code auf, der darüber hinaus sehr technisch und wenig kurz und prägnant ist (vgl. z. B. die Zeilen 8–13 mit den Zeilen 14–19 in Code 6.1).

```
<xslt:transform version="1.0" exclude-result-prefixes="xupe"                                        1
  xmlns:xslt="http://www.w3.org/1999/XSL/Transform"                                                 2
  xmlns:xupe="http://www.informatik.uni-kiel.de/XUpdateExtend">                                     3
                                                                                                    4
<!-- Validierungsregeln des move-after-Operators -->                                                5
<xslt:template match="xupe:move-after" mode="validation">                                           6
  <xslt:variable name="name-of-context" select="'move-after'"/>                                      7
  <xslt:choose>                                                                                     8
    <xslt:when test="@from"/>                                                                       9
    <xslt:otherwise>                                                                                10
      <xslt:message>The operator 'move-after' must have a 'from' attribute.</xslt:message>          11
    </xslt:otherwise>                                                                               12
  </xslt:choose>                                                                                    13
  <xslt:choose>                                                                                     14
    <xslt:when test="@to"/>                                                                         15
    <xslt:otherwise>                                                                                16
      <xslt:message>The operator 'move-after' must have a 'to' attribute.</xslt:message>            17
    </xslt:otherwise>                                                                               18
  </xslt:choose>                                                                                    19
  <xslt:for-each select="@*">                                                                       20
    <xslt:choose>                                                                                   21
      <xslt:when test="name()='from' or name()='to'">                                               22
        <xslt:message>The attribute '<xslt:value-of select="name()"/>' is not allowed               23
          on element '<xslt:value-of select="../name()"/>'.</xslt:message>                          24
      </xslt:when>                                                                                  25
    </xslt:choose>                                                                                  26
  </xslt:for-each>                                                                                  27
  <xslt:for-each select="descendant::xupe:*[local-name(ancestor::xupe:*[1])=$name-of-context]">      28
    <xslt:message>The element '<xslt:value-of select="name()"/>' is not allowed                     29
      on the operator '<xslt:value-of select="$name-of-context"/>'.</xslt:message>                  30
  </xslt:for-each>                                                                                  31
</xslt:template>                                                                                    32
                                                                                                    33
</xslt:transform>                                                                                   34
```

Code 6.1: Mögliche Validierungsregeln für den `move-after`-Operator.

Die Idee ist, die Validierungsregeln (das Was) nicht direkt auf technischer Ebene zu implementieren (das Wie), sondern auf eine abstraktere Ebene zu verlagern. Die Spezifikation der Validierungsregeln erfolgt in einer fachlich motivierten Sprache, die spezifische, für diese Aufgabe typische Konzepte beinhaltet und von technischen Details der Umsetzungstechnologie abstrahiert. Eine solche Sprache ist bspw. eine Schemasprache zur Definition eines XML-Typsystems oder aber auch eine andere Grammatik. Über eine Transformation, welche aus einer Spezifikation Codebestandteile der konkreten Plattform (z. B. die Transformationssprache XSLT) erzeugt, werden die fachlichen Informationen mit technischen Details angereichert bzw. verfeinert. Dadurch wird das Was vom Wie entkoppelt. Der Sprachentwickler kann sich auf die Erstellung der Validierungsregeln für die neuen Operatoren konzentrieren.

Die Trennung in der oben beschriebenen Idee bewirkt eine Verringerung der Abhängigkeit von der technischen Plattform (z. B. XSLT). So können Weiterentwicklungen der Plattform oder sogar ein Austausch der Plattform in den Generatoren nachvollzogen werden. Neben der Reduzierung des Entwicklungsaufwandes durch die Generation der Operatorvalidierung kann somit auch eine höhere Flexibilität der Implementierung erreicht werden.

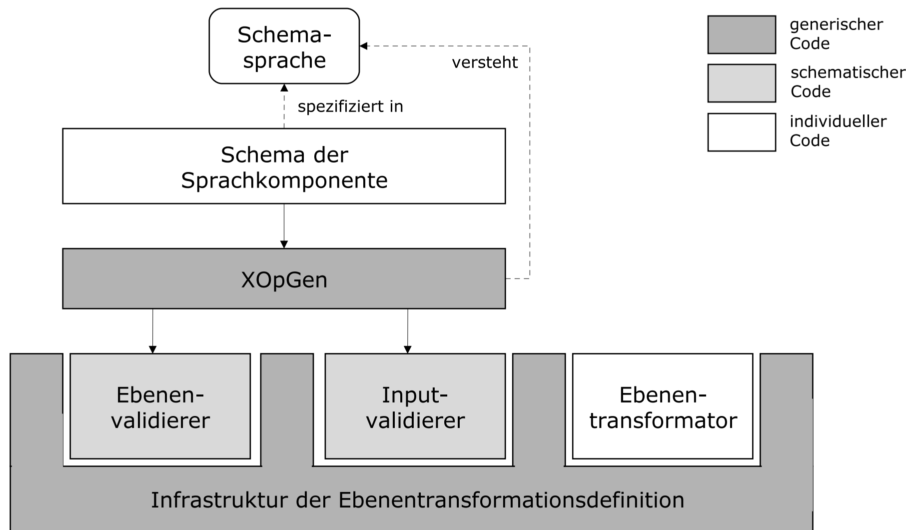


Abbildung 6.2: Konzept von XOpGen.

Abbildung 6.2 zeigt das zugrundeliegende Konzept. Aus den Informationen eines Schemas, welche in ihrer Gesamtheit die Syntax der zusätzlichen Operatoren festlegen, werden mit Hilfe von XOpGen zwei Validierer erzeugt. Dies ist zum einen ein Ebenenvalidierer, der die korrekte Benutzung der höheren Operatoren sicherstellt und damit die eigentliche Ebenentransformationsdefinition entlastet. Infolgedessen muss sich die Ebenentransformationsdefinition nicht um inkorrekte höhere Operatoren kümmern und wird somit nicht unnötig kompliziert. Eine solche klare Trennung zwischen Validierer und Transformator ist möglich, da die Validierung der höheren Operatoren völlig unabhängig davon ist, was aus ihnen erstellt wird. Zum anderen wird ein zweiter Validierer generiert, der sichert, dass der Input der zusätzlichen höheren Operatoren korrekte Datentypen aufweist. Da eine solche Überprüfung erst während der elementaren Transformation erfolgen kann, werden entsprechende Testabfragen in der elementaren Transformationsdefinition indirekt über die Ebenentransformation erstellt. Deshalb ist der generierte Inputvalidierer genau genommen gar kein Validierer, sondern vielmehr schematischer Code, der verantwortlich dafür ist, dass während der Ebenentransformation Testabfragen für die elementare Transformation konstruiert werden.

Beide generierten Validierungskomponenten werden in die vorhandene Infrastruktur für Ebenentransformationsdefinitionen eingebunden. Die Infrastruktur selbst ist ein generischer Bestandteil, der unabhängig von der Syntax konkreter Operatoren ist und daher nicht aus dem Schema abgeleitet werden muss bzw. kann. Neben den Validierungskomponenten kann in die Infrastruktur ebenso der Ebenentransformator eingebunden werden, der für die eigentliche Überführung

der neuen, höheren Operatoren in niedrigere Operatoren verantwortlich ist.

Damit nicht nur Validierungskomponenten für eine feste technische Zielplattform (vorhandene Infrastruktur und Basistransformationssprache) erzeugt werden können, bietet der Generator Schnittstellen an. Diese Schnittstellen stellen Variationspunkte zur Verfügung, mit denen spezifische, individuelle Anpassungen vorgenommen werden können (z. B. Art und Umfang der Fehlermeldungen im Fehlerfall), die entsprechend während der Generation verarbeitet werden.

6.2 Detaillierte Beschreibung

Als Ausgangsbasis für die Generation der Validierungskomponenten kommen unterschiedliche Arten von Schemasprachen in Frage. Wie in Abschnitt 2.3 geschildert, können diese grob in grammatikbasierte und regelbasierte Ansätze unterschieden werden. Mit grammatikbasierten Ansätzen kann die Struktur von XML-Dokumenten sowie Datentypen erheblich prägnanter und kürzer, als dies mit regelbasierten Ansätzen möglich ist, beschrieben werden. Dies liegt daran, dass viele Regeln implizit durch die Strukturbeschreibung aufgestellt werden. Mit regelbasierten Ansätzen hingegen können weitaus komplexere Einschränkungen, wie bspw. gegenseitige Abhängigkeiten zwischen verschiedenen Elementen und Attributen, spezifiziert werden. Zudem können Fehlermeldungen für Nichteinhaltungen der spezifizierten Einschränkungen formuliert werden. Insgesamt sind regelbasierte Ansätze mit ihren expliziten Regeln näher an der eigentlichen Umsetzung und können somit leichter implementiert werden, wie bspw. ebenso in [Jel99] oder [Nor99] festgestellt wurde. Eine Technologie für die Umsetzung wird als solches allerdings noch nicht festgelegt.

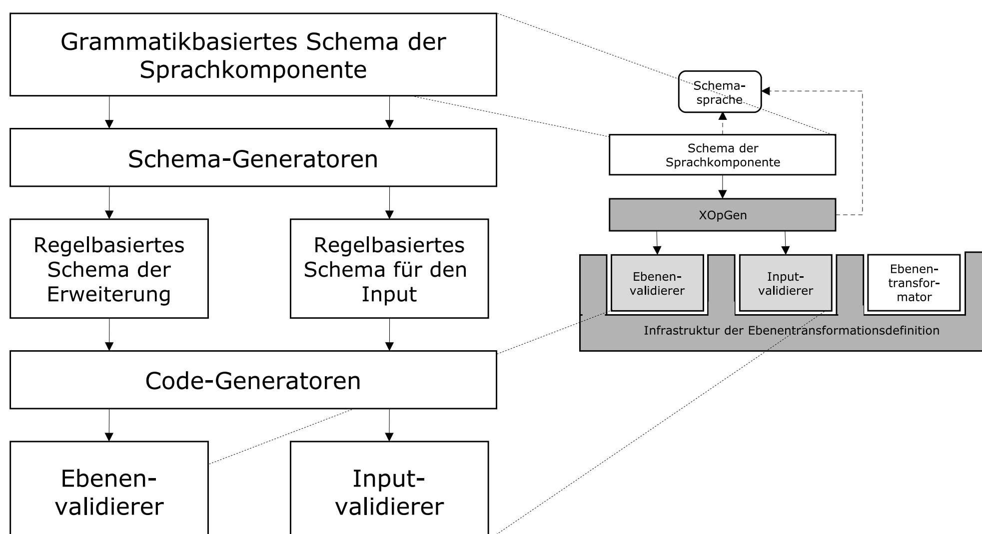


Abbildung 6.3: Generationsprozess vom grammatikbasierten Schema zu den Validierern.

Um die Vorteile der unterschiedlichen Ansätze von Schemasprachen zu nutzen, wird in [Föt07a] eine zweistufige Generierung von einem grammatikbasierten Schema über regelbasierte Schemata

zu den eigentlichen Validierungskomponenten aufgebaut (vgl. mit Abbildung 6.3). Durch diese Kaskadierung können in einem ersten Generationsschritt implizite Regeln in explizite Regeln überführt werden. Ebenso können in diesem Schritt die strukturell gleichen Fehlermeldungen generiert werden. Im zweiten Generationsschritt, bei der Überführung der expliziten Regeln in eine konkrete Implementierung, kann auf existierende Transformatoren aufgesetzt werden. Die einzelnen Generationsschritte werden im Folgenden an den Beispielen der Anwendungsszenarien vertieft. Die Implementierung wird im Abschnitt 6.3 vorgestellt.

Grammatikbasiertes Schema der Sprachkomponente

Als Ausgangspunkt dient aus den oben genannten Gründen eine grammatikbasierte Schemasprache. Da das vom W3C verabschiedete XML Schema zum einen im Vergleich zu anderen grammatikbasierten Schemasprachen sehr mächtig ist [LC00, Jel01] und ihm zum anderen die größte praktische Bedeutung zukommt, wurde dieser Sprachvorschlag für die Spezifikation der Sprachkomponenten gewählt. Andere oder zukünftige grammatikbasierte Sprachvorschläge sind grundsätzlich nicht ausgeschlossen. Für sie muss der Schema-Generator entsprechend angepasst werden. Alternativ können auch Schematransformationswerkzeuge wie bspw. trang [Tha03] zum Einsatz kommen. Dieses kann die grammatikbasierten Schemasprachen RELAX NG und DTD in XML Schema überführen.

```
1 <xsd:schema
2    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
3    targetNamespace="http://www.informatik.uni−kiel.de/XUpdateExtend">
4   <xsd:element name="move−after" type="MoveAfterType"/>
5   <xsd:complexType name="MoveAfterType">
6     <xsd:attribute name="from" type="xsd:string" use="required"/>
7     <xsd:attribute name="to" type="xsd:string" use="required"/>
8   </xsd:complexType>
9 </xsd:schema>
```

CODE 6.2: Syntaxdefinition des `move-after`-Operators in XML Schema.

Der Code 6.2 zeigt exemplarisch eine Syntaxdefinition des `move-after`-Operators aus dem einführenden Anwendungsszenario (Abschnitt 4.1.1) in der Schemasprache XML Schema. Im Wurzelelement `schema` wird zunächst der Ziel-Namensraum (`targetNamespace`) definiert, für den die Typdefinitionen und Deklarationen von Elementen spezifiziert werden. Anschließend wird das Element `move-after` deklariert und dem Typ `MoveAfterType` zugeordnet. In der Typdefinition von `MoveAfterType` (Zeilen 5–8) wird wiederum festgelegt, dass alle Elemente, die auf diesen Typ verweisen, die zwingend notwendigen Attribute `from` und `to` besitzen müssen. Beide Attribute sind vom Typ `string`.

Aus diesen Informationen kann ein regelbasiertes Schema der Sprachkomponente abgeleitet werden, welches die Zusicherungen spezifiziert, die für die neuen Operatoren in den erweiterten Transformationsdefinitionen eingehalten werden müssen. Die Zusicherungen werden später vor der eigentlichen Ebenentransformation überprüft. Zudem kann daraus ein regelbasiertes Schema für den Input der neuen Operatoren erzeugt werden, dessen Zusicherungen gewährleisten, dass

die eingehenden Daten in den höheren Operatoren den richtigen Datentypen entsprechen. Diese Zusicherungen werden letztendlich während der elementaren Transformation geprüft.

Regelbasiertes Schema der Sprachkomponente

In einem regelbasierten Schema werden die Regeln grundsätzlich explizit formuliert. Mit anderen Worten heißt dies, dass alles das, was nicht durch Zusicherungen verboten ist, hier erlaubt wird. Regelbasierte Ansätze ermöglichen ebendeshalb auf direkte Weise offene Markup-Sprachen, die neue oder unbekannte Operatoren neben den durch das Schema festgelegten Operatoren zulassen [Rog06]. Dies steht im starken Gegensatz zu grammatikbasierten Ansätzen, bei denen offene Stellen explizit definiert werden müssen.

Offene Markup-Sprachen, das Spektrum reicht von einzelnen offenen Stellen bis hin zu komplett offenen Inhalten, sind für viele XML-Technologien notwendig. Beispielsweise XHTML, BPEL, WSDL unterstützen offene Inhalte. Insbesondere aber auch XML-basierte Transformationssprachen, wie stellvertretend XSLT, wären ohne offene Inhalte nicht realisierbar.

Dementsprechend beziehen sich die aus dem grammatikbasierten Schema überführten Zusicherungen nur auf einen bestimmten Teil der nichtelementaren Transformationsdefinitionen, gerade die neuen zusätzlichen Operatoren. Für die Überführung können Informationen im grammatikbasierten Schema über

- Elementnamen,
- Informationen zur eigenen minimalen und maximalen Auftretenshäufigkeit,
- Typinformation
  - zur Reihenfolge und Anzahl von erlaubten und zwingenden Kindelementen im Elementinhalt,
  - zum Datentyp im Elementinhalt mglw. mit festen Wertvorgaben oder Nullwertfähigkeit,
  - zu den erlaubten und zwingenden Attributen mit Namen und Datentypen mglw. mit festen Wertvorgaben,
  - zu den Definitionen von Wildcard-Elementen und -Attributen inkl. spezifizierter Einschränkungen wie Anzahl, Namensraum, usw., oder
- und sonstige verarbeitbare Anmerkungen.

benutzt werden.

Zur Beschreibung der Zusicherungen wird der von der ISO standardisierte Sprachvorschlag Schematron [Int06] eingesetzt. Der Code 6.3 veranschaulicht beispielhaft, wie die Zusicherungen in Schematron formuliert werden können.

```
1  <sch:schema
2     xmlns:sch="http://www.ascc.net/xml/schematron"
3     fpi="http://www.informatik.uni-kiel.de/XUpdateExtend">
4    <sch:ns prefix="xupe" uri="http://www.informatik.uni-kiel.de/XUpdateExtend"/>
5    <sch:pattern name="generated-from-xml-schema">
6      <sch:rule id="MoveAfterType" abstract="true">
7        <sch:assert test="@from">The operator '<sch:name/>' must have a
8            'from' attribute.</sch:assert>
9        <sch:assert test="@to">The operator '<sch:name/>' must have a
10           'to' attribute.</sch:assert>
11       <sch:assert for-each="@*" test="name()='from' or name()='to'">
12           The attribute '<sch:name/>' is not allowed on the operator '<sch:name path=".."/>'.
13       </sch:assert>
14       <sch:assert
15         for-each="descendant::xupe:*[local-name(ancestor::xupe:*[1])=$name-of-context]"
16         test="false()">The element '<sch:name/>' is not allowed on the operator
17         '<sch:value-of select="$name-of-context"/>'.</sch:assert>
18     </sch:rule>
19     <sch:rule context="xupe:move-after">
20       <sch:extends rule="MoveAfterType"/>
21     </sch:rule>
22     <sch:rule context="xupe:*">
23       <sch:assert for-each="." test="local-name()='move-after'">The element
24         '<sch:name/>' is not an operator in the namespace
25         'http://www.informatik.uni-kiel.de/XUpdateExtend'.</sch:assert>
26     </sch:rule>
27   </sch:pattern>
28 </sch:schema>
```

Code 6.3: Zusicherungen für den `move-after`-Operator in Schematron.

Zunächst wird im Container-Element `schema` der Namensraum der erweiterten Operatoren `xupe` deklariert. Weiterhin enthält es benannte Regelmuster (`pattern`), in denen die expliziten Regeln (`rules`) für die erweiterten Operatoren gruppiert werden. Für jede Elementdeklaration und jede Typdefinition im grammatikbasierten Schema wird eine Regel erzeugt. Die Zeilen 19–21 beinhalten bspw. die Regel für den `move-after`-Operator. Sie verweist auf eine Regel (Zeilen 6–18), die aus der entsprechenden Typdefinition abgeleitet werden kann. In dieser abstrakten Regel, die theoretisch von anderen sowohl abstrakten als auch nichtabstrakten Regeln referenziert werden kann, werden die eigentlichen Annahmen und Ausnahmen des `move-after`-Operators spezifiziert. Die ersten beiden Annahmen sichern, dass der Operator die zwingend notwendigen Attribute `from` und `to` besitzen muss. Die dritte und vierte Zusicherung legt fest, dass keine anderen Attribute als die beiden erlaubten verwendet werden sowie keine Kindelemente existieren dürfen. Die strukturell gleichen Fehlermeldungen können ebenfalls im ersten Generationsschritt automatisch aus dem grammatikbasierten Schema erzeugt werden.

Neben der im Beispielcode 6.3 gezeigten Strukturzusicherung für Kindelemente und Attribute sind grundsätzlich ebenso Typzusicherungen (siehe Beispielcode 6.4 im nächsten Abschnitt) aus dem grammatikbasierten Schema ableitbar, die einen bestimmten Datentyp (Datum, Integer, Aufzählungstyp, usw.) für einen Attributwert oder Elementinhalt definieren.

Sämtliche Zusicherungen werden in einem weiteren Generationsschritt in eine Validierungskomponente überführt (ähnlich wie der Beispielcode 6.1), die diese vor der Ausführung der Ebenentransformation prüft. Typzusicherungen sind zu diesem Zeitpunkt aber nur prüfbar, wenn kon-

stante Werte in den neuen höheren Operatoren sowohl als Attributwert oder Elementinhalt verwendet wurden. Aus diesem Grund wird ein zweites regelbasiertes Schema für den Input der höheren Operatoren aus dem grammatikbasierten Schema abgeleitet.

Regelbasiertes Schema für den Input

Die Informationen über die zusätzlichen höheren Operatoren innerhalb des grammatikbasierten Schemas können ebenso für die Überprüfung des Inputs der Operatoren genutzt werden. So kann getestet werden, ob die selektierten Daten des Inputs für die höheren Operatoren insbesondere für deren Attribute im benötigten Wertebereich liegen. Da dieser Input bspw. von den Daten eines Eingabedokumentes abhängig sein kann, kann diese Typzusicherung erst während der elementaren Transformation, bei der das Quelldokument eingelesen wird, überprüft werden. Die Beschreibung dieser Typregeln erfolgt erneut in der Schemasprache Schematron.

Eine Typregel muss dabei genau genommen nicht nur die Menge von Werten, den Wertebereich, zu einem gegebenen Datentyp, sondern den gesamten erlaubten lexikalischen Bereich für einen Datentyp überprüfen. Jeder Wert kann durch ein oder mehrere Literale repräsentiert werden. So sind bspw. `100` und `1.0E2` zwei verschiedene Literale aus dem lexikalischen Bereich `float`, die beide den selben Wert beschreiben.

Entsprechende Typregeln können dabei definiert werden für:

- primitive Datentypen wie `string`, `boolean`, `decimal`, `float`, `double`, `duration`, `date`, `time`, `hexBinary`, usw.,
- abgeleitete Datentypen
  - durch Einschränkungen eines Datentyps, um den Wertebereich und/oder den lexikalischen Bereich auf eine Untermenge seines Basistyps einzugrenzen (`NMTOKEN`, `Name`, `ID`, `integer`, `long`, usw.),
  - durch Vereinigung eines oder mehrerer Datentypen (wie bei `anySimpleType`, der alle primitiven Datentypen vereinigt),
  - durch Auflistung von endlichen Sequenzen von Werten eines anderen Datentyps (`NMTOKENS`, `IDREFS`, `ENTITIES` usw.).

Der Codeausschnitt 6.4 zeigt exemplarisch Zusicherungen des `tr`-Operators aus dem zweiten einführenden Anwendungsszenario (Abschnitt 4.1.2). Neben Attributen (z. B. `style`, `title`, usw.) mit dem primitiven Datentyp `string` enthält der `tr`-Operator u. a. Attribute, deren Datentyp von einem Basisdatentyp eingeschränkt wurde. Beispielsweise wurde der Wertebereich der Attribute `dir`, `align` und `valign` durch Aufzählung auf bestimmte benannte Werte begrenzt (Zusicherungen der Zeilen 7–14). Ebenso wurde der Datentyp vom Attribut `char` vom Basisdatentyp `string` abgeleitet. Hier wurde die Anzahl der erlaubten Zeichen auf eines limitiert (Zusicherung der Zeilen 15–16). Auf vergleichbare Weise können bspw. Zusicherungen für Attribute mit dem primitiven Datentyp `string` (z. B. für `style`, `title`) formuliert werden, die eine

bestimmte Menge von Zeichensequenzen endlicher Länge festlegen. Dies ist gerade die Menge an Wörtern, die der `Char`-Produktion in der XML-Spezifikation [BPS+06a] entsprechen.

```
1  <sch:schema
2     xmlns:sch="http://www.ascc.net/xml/schematron"
3     fpi="http://www.informatik.uni-kiel.de/XHTML">
4    <sch:ns prefix="xhtml" uri="http://www.informatik.uni-kiel.de/XHTML"/>
5    <sch:pattern name="generated">
6      <sch:rule context="xhtml:tr">
7        <sch:report test="@dir and not(@dir='ltr' or @dir='rtl')">The value of the
8          attribute 'dir' is not allowed on the element '<sch:name/>'.</sch:report>
9        <sch:report test="@align and not(@align='left' or @align='center' or @align='right' or
10         @align='justify' or @align='char')">The value of the attribute 'align' is not allowed
11         on the element '<sch:name/>'.</sch:report>
12       <sch:report test="@valign and not(@valign='top' or @valign='middle' or
13         @valign='bottom' or @valign='baseline'">The value of the attribute 'valign'
14         is not allowed on the element '<sch:name/>'.</sch:report>
15       <sch:report test="@char and string-length(@char)!=1">The value of the attribute 'char'
16         on the element '<sch:name/>' must be the length of '1'.</sch:report>
17        ...
18     </sch:rule>
19      ...
20   </sch:pattern>
21 </sch:schema>
```

Code 6.4: Zusicherungen für den Input des `tr`-Operators in Schematron.

In einem weiteren Generationsschritt wird aus diesen Zusicherungen entsprechender Code erzeugt, der seinerseits bei der Ebenentransformation elementare Operatoren einer Basistransformationssprache generiert, die während der eigentlichen elementaren Transformation die Typprüfung des Inputs vornehmen. Abhängig von der Basistransformationssprache können diese elementaren Operatoren einfache Kontrolloperatoren sein.

6.3 Realisierung

Das Konzept wurde mit Hilfe des Generatorsystems XOpGen (*XML Operator Generator*) umgesetzt [Föt07a]. Es greift nicht in die Implementierung von XTC ein. Dadurch kann dieses Modul grundsätzlich unabhängig von XTC verwendet werden. Die wesentlichen Hauptbestandteile von XOpGen sind:

- die Schema-Generatoren für die Überführung des grammatikbasierten Schemas der Sprachkomponente in die beiden regelbasierten Schemata,
- die Code-Generatoren für die Überführung der regelbasierten Schemata in die Validierungskomponenten,

Die Generatoren müssen die Validierungskomponente so generieren, dass sie in die Infrastruktur der Ebenentransformationsdefinitionen eingebunden werden können. Der Aufbau der einzelnen Bestandteile wird in den folgenden Abschnitten vorgestellt.

6.3.1 Generatoren für die Schemaüberführungen

Sowohl das grammatikbasierte Schema (in XML Schema) als auch die regelbasierten Schemata (beide in Schematron) werden durch XML-basierte Markup-Sprachen spezifiziert. Gerade deshalb bieten sich für die Überführung spezifische XML-Transformationssprachen an. Änderungssprachen sind dabei nicht so gut geeignet, da das Quellschema eine komplett andere Struktur aufweist als die Zielsprache. Insofern sind Transformationsansätze, die ein Zieldokument vollständig neu aufbauen, zu bevorzugen. Für die Umsetzung einer solchen Überführung sind u. a. die Unterstützung von Namensräumen sowie die Möglichkeit der Erstellung von Variationspunkten essentiell. Da diese und weitere Kriterien von XSLT als Transformationstechnologie erfüllt werden, wurden die Generatoren mit XSLT implementiert.

Die Tabelle 6.1 stellt grundlegende Sprachkonstrukte von XML Schema denen von Schematron grob gegenüber. Das `schema`-Element von XML Schema wird in das `schema`-Element von Schematron mit einem `pattern`-Element, in dem sämtliche erzeugte Regeln aufgenommen werden, überführt. Aus den globalen komplexen Typdefinitionen und den globalen Elementdeklaration werden Regeln abgeleitet. Die Regeln für die Typdefinitionen sind abstrakt und können über deren eindeutige Namen (`id`) von anderen Regeln referenziert werden. Sie besitzen deswegen im Gegensatz zu konkreten Regeln, die von den Elementdeklarationen generiert werden, keinen Kontext, in dem sie angewendet werden. Beide Regelarten besitzen aber Annahmen und Ausnahmen, die entsprechend aus den restlichen Sprachkonstrukten und deren Regeln überführt werden. Lokale Elementdeklarationen werden ebenso in explizite Regeln überführt. Zur Definition des Kontextes wird ein rekursiver Algorithmus angewendet, der alle möglichen Vaterelemente sucht, um die Kontextbezogenheit der lokalen Elementdeklaration gerecht zu werden.

Besitzt bspw. ein Attribut `space` einen festen Wert (`fixed=`“`preserve`“), so wird dies in Schematron mit der Ausnahme

```
<sch:report test="@space and not(@space='preserve') and not(starts-with(@space,'{') and
 substring(@space, string-length(@space))='}')">
  The value of the attribute 'space' must be the fixed value 'preserve'
  on element '<sch:name/>' defined by the respective XML Schema.
</sch:report>
```

gesichert.[2] Neben dieser und den in der Tabelle 6.1 gezeigten 1-zu-1-Abbildungen, müssen ebenso explizite Annahmen und Ausnahmen generiert werden, die aus mehreren Sprachkonstrukten im XML Schema abgeleitet werden. Beispielsweise muss für eine Elementdeklaration, in der mehrere Attribute deklariert werden, eine Annahme erzeugt werden, die sichert, dass keine anderen Attribute als die deklarierten in diesem Element vorkommen.

Um den Schema-Generator für spezielle Bedürfnisse einfach anpassen zu können, trennt XOpGen zwischen festen, unveränderlichen Verhalten und variablen, veränderlichen Verhalten. Für den offenen, variablen Teil verwendet der Generator ein Entwurfsmuster, welches das „Hollywood-Prinzip“ [Swe85] verfolgt (*don’t call us, we’ll call you*). Es stellt eine Vielzahl sogenannter Hooks bereit, die genutzt werden können, um eigenen spezifischen Code hinzuzufügen. Diese Hooks

[2] Der dritte konjunktive Ausdruck ist notwendig, da in den höheren Operatoren Attribut-Wert-Templates erlaubt werden sollen. Diese beginnen und enden mit einer geschweiften Klammer.

XML Schema	Schematron
Schema <xsd:schema targetNamespace=“...“> ... </xsd:schema>	Schema <sch:schema fpi=“...“> ... <sch:pattern name=“generated-from-xml-schema“> ... </sch:pattern> </sch:schema>
Elementdeklaration <xsd:element name=“...“ type=“...“ vordef. simpler Typ benutzerdef. simpler Typ komplexer Typ fixed=“true“ minOccurs=“...“ maxOccurs=“...“ nillable=“false“ substitutionGroup=“...“> ... </xsd:element>	Regeldeklaration inkl. Annahmen und Ausnahmen <sch:rule context=“...“ > <sch:assert test=“...“>...</sch:assert> Annahmen und Ausnahmen der simplen Typdefinition <sch:extends rule=“...“/> <sch:report test=“...“>...</sch:report> <sch:assert test=“...“>...</sch:assert> <sch:assert test=“...“>...</sch:assert> <sch:assert test=“...“>...</sch:assert> Annahmen und Ausnahmen der ersetzten Elementgruppe ... </sch:rule>
Attributdeklaration <xsd:attribute name=“...“ type=“...“ fixed=“true“ use=“required“/>	Annahmen und Ausnahmen <sch:assert test=“...“>...</sch:assert> <sch:report test=“...“>...</sch:report> <sch:assert test=“...“>...</sch:assert>
simple Typdefinition <xsd:simpleType name=“...“ ... </xsd:simpleType>	Annahmen <sch:assert test=“...“>...</sch:assert>
komplexe Typdefinition <xsd:complexType name=“...“ mixed=“false“/> ... </xsd:complexType>	abstrakte Regeldeklaration inkl. Annahmen <sch:rule id=“...“ abstract=“true“ > <sch:assert test=“...“>...</sch:assert> ... </sch:rule>
Disjunktive Elementgruppen <xsd:choice minOccurs=“...“ maxOccurs=“...“/> ... </xsd:choice>	Annahmen <sch:assert test=“...“>...</sch:assert> <sch:assert test=“...“>...</sch:assert> <sch:assert test=“...“>...</sch:assert>
Sequentielle Elementgruppen <xsd:sequence minOccurs=“...“ maxOccurs=“...“/> ... </xsd:sequence>	Annahmen <sch:assert test=“...“>...</sch:assert> <sch:assert test=“...“>...</sch:assert> <sch:assert test=“...“>...</sch:assert>
Konjunktive Elementgruppen <xsd:all minOccurs=“...“ maxOccurs=“...“/> ... </xsd:all>	Annahmen <sch:assert test=“...“>...</sch:assert> <sch:assert test=“...“>...</sch:assert> <sch:assert test=“...“>...</sch:assert>

TABELLE 6.1: Schematische Abbildung von Sprachkonstrukten des XML Schemas nach Schematron.

werden während der Überführung des grammatikbasierten Schemas in regelbasierte Schemata automatisch aufgerufen. In XSLT werden solche Hooks durch benannte Templates realisiert. Für die benannten Templates stellt XOpGen eine Standardimplementierung zur Verfügung, die mit Hilfe gleichnamiger, benutzerspezifischer Templates überschrieben werden kann. Die Tabelle 6.2 gibt einige Beispiele existierender Erweiterungspunkte an.

Template-Name	**Verwendung**
`process-prolog`	Es wird am Start der Generation aufgerufen und ist sinnvoll, um andere Transformationsdefinitionen zu importieren oder Variablen zu setzen.
`process-schema`	Es wird beim Wurzelelement `schema` aufgerufen. Die Standardeinstellung bewirkt nichts.
`process-...`	Nach dem gleichen Prinzip wird für nahezu jedes Sprachkonstrukt von XML Schema ein Hook zum Hinzufügen neuer Annahmen und Ausnahmen angeboten.
`assert-attribute-type-integer`	Es wird bei jedem Attribut aufgerufen, dessen Wertebereich auf ganze Zahlen beschränkt wurde (vordefinierter simpler Datentyp `integer`). Die Standardeinstellung sieht einen entsprechenden Test vor.
`report-restriction-element-enumeration`	Es wird bei jedem Element aufgerufen, dessen Wertebereich auf bestimmte benannte Werte beschränkt wurde (Aufzählung). Die Standardeinstellung sieht einen entsprechenden Test vor.
`assert-...`,`report-...`	Nach dem gleichen Prinzip wird für viele Attribute jedes Sprachkonstruktes von XML Schema ein Hook angeboten, welcher es erlaubt, den Test für die Zusicherungen anzupassen.
`message-attribute-type-integer`	Es wird ebenfalls bei jedem Attribut aufgerufen, dessen Wertebereich auf ganze Zahlen beschränkt wurde (vordefinierter simpler Datentyp `integer`). Die Standardeinstellung sieht eine entsprechende Fehlermeldung vor.
`message-restriction-element-enumeration`	Es wird ebenfalls bei jedem Element aufgerufen, dessen Wertebereich auf bestimmte benannte Werte beschränkt wurde (Aufzählung). Die Standardeinstellung sieht eine entsprechende Fehlermeldung vor.
`message-...`	Nach dem gleichen Prinzip können für alle bereits definierten Annahmen und Ausnahmen mit Hilfe dieser Hooks die Fehlermeldungen angepasst werden.

Tabelle 6.2: Übersicht über die wichtigsten Hooks des Schema-Generators.

Die Erweiterungspunkte werden u. a. dafür benutzt, um die beiden zum Teil unterschiedlichen regelbasierten Schemata umzusetzen. So wird bspw. der Test für die Ausnahme auf Seite 131 zu festen Werten in Attributen im regelbasierten Schema der Sprachkomponente aus folgender Template-Definition generiert:

```
<xsl:template name="report-attribute-fixed">
  <xsl:param name="fixed"/>
  <xsl:param name="name"/>
  <xsl:text>@</xsl:text>
  <xsl:value-of select="$name"/>
  <xsl:text> and not(@</xsl:text>
  <xsl:value-of select="$name"/>
  <xsl:text>='</xsl:text>
  <xsl:value-of select="$fixed"/>
  <xsl:text>') and not(starts-with(@</xsl:text>
  <xsl:value-of select="$name"/>
  <xsl:text>,'{') and substring(@</xsl:text>
  <xsl:value-of select="$name"/>
  <xsl:text>,string-length(@</xsl:text>
  <xsl:value-of select="$name"/>
  <xsl:text>) = '}')</xsl:text>
</xsl:template>
```

Diese Template-Definition entspricht der Standardkonfiguration des Schema-Generators.

Bei der Überprüfung des Inputs dürfen hingegen keine Attribut-Wert-Templates auftreten. Aus diesem Grund ist der entsprechende dritte konjunktive Ausdruck überflüssig. Mit Hilfe dieses Hooks kann der entsprechende Code angepasst werden, indem die alte Definition überschrieben wird.

Der folgende Codeausschnitt zeigt die entsprechende Ausnahme für das regelbasierte Schema für den Input.

```
<xsl:template name="report-attribute-fixed">
  <xsl:param name="fixed"/>
  <xsl:param name="name"/>
  <xsl:text>@</xsl:text>
  <xsl:value-of select="$name"/>
  <xsl:text> and not(@</xsl:text>
  <xsl:value-of select="$name"/>
  <xsl:text>='</xsl:text>
  <xsl:value-of select="$fixed"/>
  <xsl:text>')</xsl:text>
</xsl:template>
```

Die Vielzahl von Hooks kann darüber hinaus verwendet werden, um bspw. Fehlermeldungen an die Terminologie der Benutzergruppe individuell anzupassen. Zudem können neue Annahmen und Ausnahmen hinzugefügt oder entfernt werden. Dadurch kann der Grad der Validierung und damit die Offenheit oder Spezifität der höheren Transformationsdefinitionen reguliert werden.

6.3.2 Generatoren zur Erzeugung der Validierungskomponenten

Auch die für die Erzeugung der Validierungskomponenten notwendigen Generatoren können mit einer spezifischen XML-Transformationssprache realisiert werden, da die im ersten Generationsschritt erzeugten regelbasierten Schemata in XML-Syntax vorliegen. Wie das Eingangsbeispiel 6.1 mustergültig zeigt, lassen sich auch die im regelbasierten Schemata vorliegenden Zusicherungen ausgezeichnet mit XSLT implementieren. Diese Beobachtung wurde ebenso in [Nor99] oder [Jel99] gemacht. XSLT kann somit für die Realisierung der Generationskomponente (Meta-Transformationskomponente), die aus regelbasierten Schemata Validierungskomponenten

erzeugt, als auch für die Realisierung der Validierungskomponente selbst, die die zusätzlichen neuen Operatoren überprüft, dienen.

Tabelle 6.3 zeigt die Abbildung der Kernsprachkonstrukte von Schematron nach XSLT. Annahmen (assert) und Ausnahmen (report) werden in einfache XSLT-Kontrollblöcke überführt. Von den Regeln (rules) werden individuelle Templates abgeleitet, deren match-Attribut direkt vom Kontext der Regel übernommen wird. Der Modus der Templates wird vom Regelmuster (pattern) entnommen, in dem diese Regeln und andere Regeln gruppiert werden. Weitere Details können in [Dod01] nachgelesen werden.

Schematron	**XSLT**
Schema <sch:schema name="..."> ... </sch:schema	Definition des Wurzel-Templates <xsl:template match="/" mode="..."> ... </xsl:template
Regelmuster <sch:pattern name="..."> ... </sch:pattern	Anwendung von Templates <xsl:apply-templates select="/" mode="..." />
Regel <sch:rule context="..."> ... </sch:rule>	Definition von Templates <xsl:templates match="..." mode="..." priority="..."> ... </xsl:templates
Annahme <sch:assert test="..."> ... </sch:assert>	Bedingte Verarbeitung <xsl:choose> <xsl:when test="..." /> <xsl:otherwise ... </xsl:otherwise> </xsl:choose>
Ausnahme <sch:assert test="..."> ... </sch:assert>	Bedingte Verarbeitung <xsl:if test="..."> ... </xsl:if>

TABELLE 6.3: Abbildung von Sprachkonstrukten des Schematrons nach XSLT.

Der Generator wurde basierend auf einem offenen Schematron-XSLT-Compiler [Sch01] realisiert. Das Grundgerüst dieses offenen Compilers wurde für die Generation der Validierungskomponenten erweitert. Beispielsweise wurden entsprechende Schnittstellen implementiert, mit denen die Validierungskomponenten in die Ebenentransformationsinfrastruktur eingebunden werden können. Darüber hinaus wurden Hooks definiert, die es erlauben, auch diesen Generator anzupassen. Die folgende Tabelle 6.4 gibt eine Übersicht über die wichtigsten Hooks.

Die Tabelle 6.4 ist keinesfalls eine vollständige Liste aller Variationspunkte. Es werden Hooks für nahezu alle Sprachkonstrukte in Schematron angeboten. Viele der Templates werden mit einer Vielzahl an Parametern aufgerufen, die aus den Attributen der Elemente von Schematron entstammen und für die Anpassung des Generators genutzt werden können.

Template-Name	Verwendung
`process-prolog`	Es wird am Start der Generation aufgerufen und ist sinnvoll, um andere Transformationsdefinitionen zu importieren oder Variablen zu setzen.
`process-schema`	Es wird beim Wurzelelement (`schema`) aufgerufen und enthält in der Standardeinstellung `xsl:apply-templates` für jedes `pattern`.
`process-pattern`	Es wird bei jedem Regelmuster (`pattern`) aufgerufen. Die Standardeinstellung bewirkt nichts.
`process-rule`	Es wird bei jeder Regel (`rule`) aufgerufen. Die Standardeinstellung bewirkt ein weiteres Durchlaufen der Regel.
`process-assert`	Es wird bei jeder Annahme (`assert`) aufgerufen und ruft in der Standardeinstellung das Template `process-message` mit entsprechenden Parametern auf.
`process-report`	Es wird bei jeder Ausnahme (`report`) aufgerufen und ruft in der Standardeinstellung das Template `process-message` mit entsprechenden Parametern auf.
`process-message`	Es wird indirekt bei jeder Annahme (`assert`) oder bei jeder Ausnahme (`report`) aufgerufen. In der Standardeinstellung werden die Fehlermeldungen mit zusätzlichen Informationen (z. B. Ursache des Fehlers) auf die Konsole ausgegeben.
`process-...`	Nach dem gleichen Prinzip wird für nahezu jeden Sprachkonstrukt in Schematron ein Hook zur Anpassung angeboten.
`generate-attribute-value-test`	Es wird aufgerufen, wenn der Code erzeugt wird, der verantwortlich ist, dass während der Ebenentransformation Kontrollblöcke für die Typüberprüfung von Attributwerten generiert werden. In der Standardeinstellung wird Metacode für spätere XSLT-Kontrollblöcke erzeugt.
`generate-element-content-test`	Es wird aufgerufen, wenn der Code erzeugt wird, der verantwortlich ist, dass während der Ebenentransformation Kontrollblöcke für die Typüberprüfung von Elementinhalten generiert werden. In der Standardeinstellung wird Metacode für spätere XSLT-Kontrollblöcke erzeugt.

TABELLE 6.4: Übersicht über die wichtigsten Hooks des Code-Generators.

Insbesondere für den zweiten Inputvalidierer sind diese Variationspunkte entscheidend. Wie bereits erwähnt, ist dieser Inputvalidierer genau genommen gar kein Validierer, sondern vielmehr schematischer Code, der während der Ebenentransformation Testabfragen für die elementare Transformation konstruiert. Erst diese Testabfragen bei der eigentlichen elementaren Transformation sichern, dass korrekte Datentypen verwendet werden. Die Variationspunkte können nun u. a. dafür verwendet werden, um elementare Operatoren einer anderen Basistransformationssprache (quasi indirekt) über die Ebenentransformation zu erzeugen. Sie bilden somit die Grundlage für die Unabhängigkeit von XOpGen von einer konkreten Basistransformationssprache.

6.3.3 Infrastruktur, Ebenentransformator und Integration

Der generische Infrastrukturcode ist verantwortlich für die Steuerung der Ebenentransformation. Der Aufbau der Komponente wird im Code 6.5 skizziert. Zunächst werden zwei Traversierungen gestartet: im `validation`-Modus (Zeile 9) und im `transformation`-Modus (Zeile 22). Innerhalb der ersten Traversierung wird die Syntax der neuen Operatoren geprüft. Es wird entsprechend bei der Dokumentwurzel begonnen. Ist die erste Traversierung beendet und es sind keine Fehler

aufgetreten, wird die zweite Traversierung gestartet. Die zweite Traversierung führt die eigentliche Ebenentransformation wieder von der Dokumentwurzel beginnend durch.

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">  1
                                                                                 2
  <xsl:output method="xml" omit-xml-declaration="no" indent="yes"/>              3
                                                                                 4
  <xsl:template match="/">                                                       5
    <xsl:message>Starting the validation                                         6
      of the higher-level transformation definitions.</xsl:message>              7
    <xsl:variable name="errors">                                                 8
      <xsl:apply-templates select="/" mode="validation"/>                        9
    </xsl:variable>                                                              10
    <xsl:if test="string-length($errors)!=0">                                    11
      <xsl:message terminate="yes">                                              12
        <xsl:value-of select="string-length($errors)"/>                          13
        <xsl:text> error(s) detected.</xsl:text>                                 14
      </xsl:message>                                                             15
    </xsl:if>                                                                    16
    <xsl:if test="string-length($errors)=0">                                     17
      <xsl:message>No error(s) detected.</xsl:message>                           18
    </xsl:if>                                                                    19
    <xsl:message>Starting the transformation                                     20
      of the higher-level transformation definitions.</xsl:message>              21
    <xsl:apply-templates select="/" mode="transformation"/>                      22
  </xsl:template>                                                                23
  ...                                                                            24
</xsl:stylesheet>                                                                25
```

Code 6.5: Generische Infrastrukturkomponente von XOpGen.

Die Integration der Infrastruktur (`infrastructure.xslt`), der Validierungskomponenten (`level-validator.xslt` und `input-validator.xslt`) sowie des Ebenentransformators (`level-transformer.xslt`) erfolgt separat. Da die Generierung der Validierungskomponenten und Programmierung des Ebenentransformators gegen die festen Schnittstellen der Infrastruktur erfolgt bzw. erfolgen sollte, reicht ein einfaches Importieren der entsprechenden Dateien aus, wie im Code 6.6 gezeigt wird (Zeilen 2–5).

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">  1
  <xsl:import href="level-validator.xslt"/>                                      2
  <xsl:import href="input-validator.xslt"/>                                      3
  <xsl:import href="level-transformer.xslt"/>                                    4
  <xsl:import href="infrastructure.xslt"/>                                       5
  ...                                                                            6
</xsl:stylesheet>                                                                7
```

Code 6.6: Integration der einzelnen Komponenten und der Infrastruktur.

Durch die Einbindung übernimmt der Ebenenvalidierer im `validation`-Modus bei passenden Operatoren in der höheren Transformationsdefinition die weitere Kontrolle und gibt bei Fehlern entsprechende Fehlermeldungen aus. Ähnliches gilt für die Ebenentransformation, die im `transformation`-Modus gestartet wird. Hier erzeugt der Inputvalidierer bei passenden höheren Operatoren zunächst elementare Kontrolloperatoren für die Typprüfung und anschließend

wird der Hook (`perform-transformation`) für die Implementierung der eigentlichen Ebenentransformation aufgerufen. Die Implementierung wird in einer eigenen Ebenentransformationskomponente realisiert. Durch diesen Mechanismus wird die Überführung der neuen, höheren Operatoren in niedrigere Operatoren von der Erzeugung der Operatoren für die Typprüfung konsequent getrennt.

Der Ebenentransformator kann allerdings nicht aus dem Schema der Sprachkomponente direkt oder indirekt generiert werden. Er muss im Allgemeinen manuell umgesetzt werden. Nur in besonderen Fällen, wenn die Syntax der höheren Transformationsoperatoren mit der Syntax der Sprachkonstrukte der Zielsprachen übereinstimmt, kann sich einer generischen Implementierung bedient werden. Dieser generische Code ist für alle diese Spezialfälle identisch. Code 6.7 zeigt, wie eine solche Überführung verwirklicht werden kann.

```
1  <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
2    <xsl:template name="perform-transformation">
3      <xsl:param name="mode"/>
4      <xsl:choose>
5        <xsl:when test="namespace-uri(.)=$source-uri">
6          <xsl:element name="{name(.)}" namespace="{$target-uri}">
7            <xsl:for-each select="@*">
8              <xsl:attribute name="{name()}"><xsl:value-of select="."/></xsl:attribute>
9            </xsl:for-each>
10           <xsl:call-template name="apply-templates">
11             <xsl:with-param name="mode" select="$mode"/>
12           </xsl:call-template>
13         </xsl:element>
14       </xsl:when>
15       <xsl:otherwise>
16         <xsl:copy>
17           <xsl:copy-of select="@*"/>
18           <xsl:call-template name="apply-templates">
19             <xsl:with-param name="mode" select="$mode"/>
20           </xsl:call-template>
21         </xsl:copy>
22       </xsl:otherwise>
23     </xsl:choose>
24   </xsl:template>
25 </xsl:stylesheet>
```

Code 6.7: Ebenentransformator bei analoger Syntax von höheren Transformationsoperatoren und Sprachkonstrukten der Zielsprache.

Als erstes muss unterschieden werden, ob das derzeit aktive Quellelement zum zu transformierenden Namensraum gehört (Zeile 5) oder nicht (Zeile 15). Gehört es dazu, wird es überführt, indem ein Element mit gleichem Namen (Zeile 6) und identischen Attributen (Zeilen 7–9) aber anderem Namensraum (Zeile 6) erzeugt wird. Der Elementinhalt wird mittels des benannten Templates `apply-templates` verarbeitet (Zeilen 10–12). Es wird innerhalb des Inputvalidierers definiert und ist verantwortlich für das weitere Durchlaufen der höheren Transformationsdefinitionen. Für eine korrekte Traversierung muss beim Template-Aufruf der Parameter `mode` übergeben werden. Gehört das aktuelle Quellelement nicht zum zu transformierenden Namensraum, so wird das Element (Zeile 16) und dessen Attribute (Zeile 17) kopiert und zur weiteren Traversierung wiederum das Template `apply-templates` aufgerufen (Zeilen 18–20).

Kapitel 7

Entwicklungsvorgehen

Bisher wurde das technischen Framework vorgestellt, welches die Konzepte der Operatorhierarchie und Generationstechniken für eine effiziente Entwicklung von Sprachkomponenten umsetzt. Dazu wurden die Ausführungsumgebung XTC und das Generatorsystem XOpGen aufgebaut. Ein Entwicklungsvorgehen wurde dabei zwar konstruktiv, aber noch nicht methodisch systematisiert.

In diesem Kapitel werden Vorgehensmodelle vorgestellt, wie höhere Sprachkomponenten basierend auf den nun zur Verfügung stehenden Technologien entwickelt werden können. Bezugnehmend auf die grundverschiedenen Motivationen werden ein Top-down-Vorgehensmodell und ein Bottom-up-Vorgehensmodell zur Entwicklung einer höheren Sprachkomponente vorgeschlagen.

Der Bedarf für eine höhere Sprachkomponente kann bspw. von einer Anwenderzielgruppe getrieben sein, die für ihre Domäne spezifische Notation favorisiert oder fordert. In der Regel geht eine solche Anwenderorientierung mit weiteren Eingrenzungen des Transformationsbereiches einher. Ein Beispiel dafür ist die Einschränkung auf eine bestimmte Quell- und/oder Zielsprache. Für solche Neuentwicklungen ist das Top-down-Vorgehensmodell vorgesehen.

Höhere Sprachkomponenten können aber auch als Weiterentwicklungen aus existierenden Sprachvorschlägen entstehen, weil diese bestimmte Defizite aufweisen. Die Defizite treten typischerweise beim Anwenden der Transformationssprachen auf und können bspw. immer wiederholende Operatorkombinationen wie im Beispielszenario des Abschnittes 4.1.1 sein. Für diese Fälle soll das Bottom-up-Vorgehensmodell definiert werden.

Die einzelnen Aktivitäten beider Vorgehensmodelle werden im Folgenden in eigenen Abschnitten erläutert. Abschließend werden organisatorische Strukturen besprochen.

7.1 Top-down-Vorgehensmodell

Dieses Vorgehensmodell enthält Aktivitäten, die notwendig sind, um eine domänenspezifische Sprachkomponente zu erstellen. Bei der domänenspezifischen Sprachkomponente kann es sich, je nach Umfang der in ihr definierten Operatoren, um eine vollständig eigenständige Transforma-

tionssprache handeln. Im Vorgehensmodell wird der zeitliche Zusammenhang zwischen den Aktivitäten abgebildet (vgl. Abbildung 7.1). Dabei wird lediglich der Normalfall modelliert, da alles darüber hinaus erfahrungsgemäß das grundsätzliche Vorgehen eher in den Hintergrund stellt. Insbesondere sind keine Iterationszyklen innerhalb des Vorgehensmodells dargestellt. Ebenso wurde auf Testaktivitäten verzichtet, die nach jeder Aktivität deren Ergebnisse prüfen.

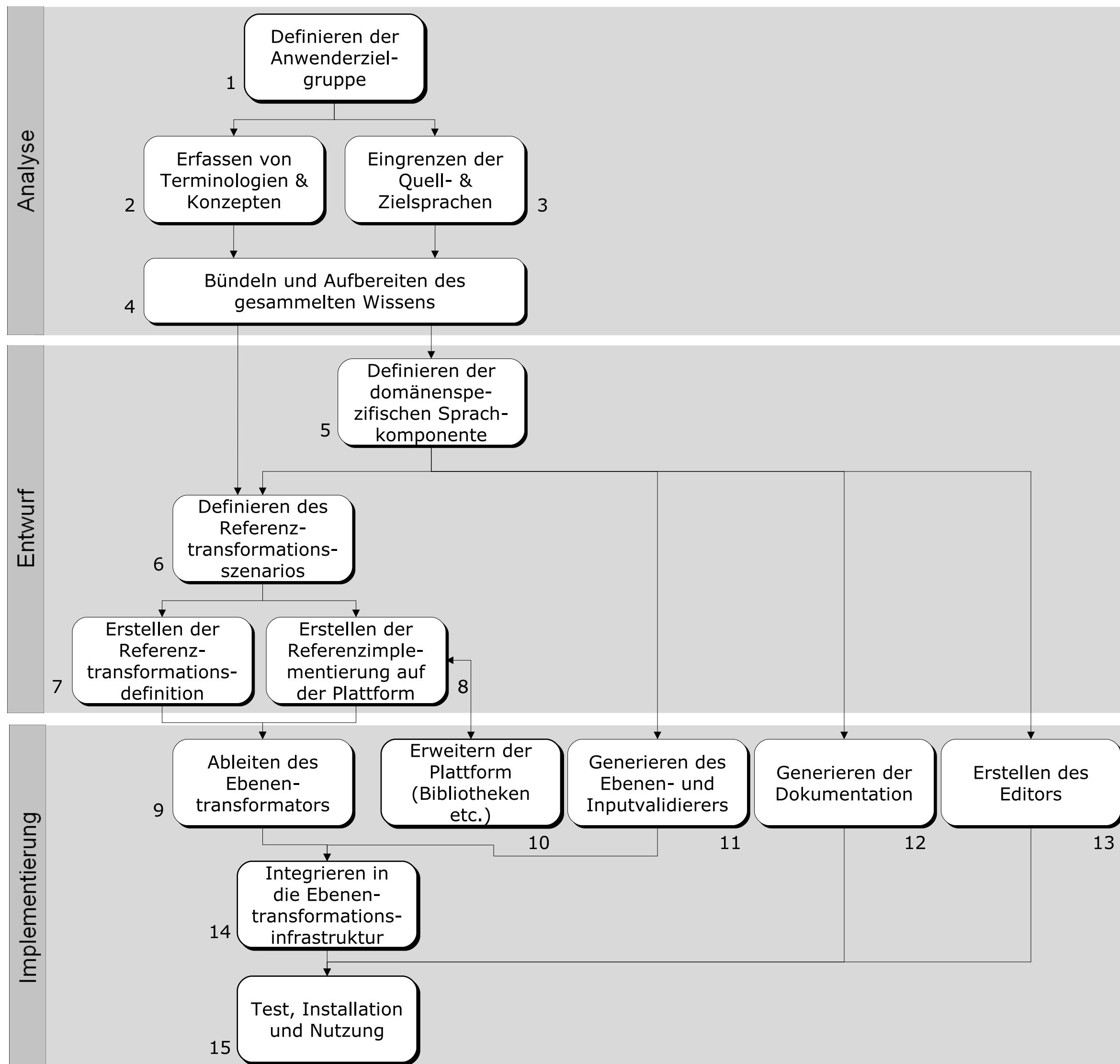


ABBILDUNG 7.1: Das Top-down-Vorgehensmodell.

Die wichtigsten Phasen des Vorgehensmodells sind Analyse, Entwurf und Implementierung. Die Phasen selbst können bei Bedarf in Zyklen ablaufen. Zu Beginn eines Projektes sollten alle Phasen zumindest einmal komplett von Anfang bis Ende durchlaufen werden. Zu späteren Projektzeitpunkten können bei einer erneuten Iteration des gesamten Vorgehensmodells die Aktivitäten, die der Analysephase zugeordnet werden können, schneller durchlaufen bzw. übersprungen werden. Die wichtigsten Aktivitäten und ihre Ergebnistypen der Phasen im Einzelnen sind:

Aktivität 1: Definieren der Anwenderzielgruppe

Im ersten Schritt wird die Zielgruppe, für die die domänenspezifische Sprachkomponente entworfen wird (z. B. Kunden, Domänenexperten, Entwickler, Programmierer), bestimmt. Eine Zielgruppe können mehrere Personen, aber auch Institutionen sein. Zunächst müssen deshalb alle Interessenbeteiligten (*Stakeholder*) erfasst werden und anschließend deren Bedürfnisse und Ansprüche identifiziert werden. Da die Bedürfnisse i. d. R. gegenläufig und widersprüchlich sind, werden diese mit den (Unternehmens-)Zielen, die mit dem Projekt verfolgt werden, gewichtet, verglichen und priorisiert. Anhand dieser Informationen wird ein Anforderungskatalog mit Zielen und Randbedingungen für die Sprachkomponente aufgestellt. Dieser umfasst sowohl funktionale als auch nichtfunktionale Anforderungen.

Zweck	Bestimmung und Priorisierung von Anwenderzielgruppen zum Erfassen der Ziele, Präferenzen und Randbedingungen
Rollen	- Domänenanalytiker (verantwortlich) - Domänenexperte - Projektleiter
Input-Artefakte	- Marktstudien - Meinungsumfragen - Business Plan für aktuelle und zukünftige Anwendungsentwicklungen
Output-Artefakte	- Anforderungskatalog für Sprachkomponenten
Methoden/Werzeuge	- Stakeholder Management

Tabelle 7.1: Aktivität 1: Definieren der Anwenderzielgruppe.

Aktivität 2: Erfassen von Terminologien und Konzepten

Ziel dieser Aktivität ist es, ein Vokabular von Begriffen zu definieren und die wesentlichen Konzepte zusammenzutragen, die im Zusammenhang mit dem aufgestellten Anforderungskatalog stehen. Zunächst müssen zu diesem Zweck potentielle Quellen für das Sammeln von Domäneninformationen identifiziert werden. Als Quellen können vorhandene Systeme und Prototypen, Handbücher und andere technische Berichte dienen. Ebenso können (externe) Domänenexperten befragt werden, um deren Erfahrungen zu dokumentieren. Insbesondere bei Befragungen können Kreativmethoden zum Einsatz kommen. Aus diesem Wissen wird ein entsprechendes Begriffsrepository aufgebaut.

Aktivität 3: Eingrenzen der Quell- und Zielsprachen

In dieser Aktivität werden die für die domänenspezifischen Sprachkomponente relevanten Quell- und Zielsprachen auf Grundlage des Anforderungskatalogs der ersten Aktivität erfasst. Bei der Bestimmung der Quell- und Zielsprachen müssen Konstrukte, die nicht sinnvoll abgebildet werden können, ausgegrenzt werden, damit überhaupt eine konsistente Überführung möglich ist. Das erlaubte Vokabular der eingeschränkten Quell- und Zielsprache wird, falls noch nicht vorhanden, durch eine formale Definition (z. B. ein Schema) festgelegt. Dies soll nicht eine iterative Ab- bzw. Eingrenzung ausschließen. Jedoch sollten zu jedem Zeitpunkt die Sprachen

Zweck	Zusammentragen von Begrifflichkeiten und wesentlichen Konzepten
Rollen	- Domänenanalytiker (verantwortlich) - Domänenexperte - Endanwender
Input-Artefakte	- Anforderungskatalog - vorhandene Systeme - Prototypen und andere technische Berichte - Handbücher
Output-Artefakte	- Glossar von Begriffen - Übersicht über wesentliche Konzepte
Methoden/Werkzeuge	- Kreativmethoden: intuitive und diskursive Methoden - Repräsentationsmethoden: Mind Map, Knowledge Maps, Ontologien

TABELLE 7.2: Aktivität 2: Erfassen von Terminologien und Konzepten.

Zweck	formale Erfassung der relevanten Quellsprachen und/oder Zielsprachen
Rollen	- Domänenanalytiker (verantwortlich) - Domänenexperte - Endanwender
Input-Artefakte	- Anforderungskatalog - Spezifikationen der Sprachen - Dokumentationen oder sonstige Beschreibungen
Output-Artefakte	- Schema der Quellsprachen - Schema der Zielsprachen
Methoden/Werkzeuge	- Schema-Editoren - Schema-Konverter

TABELLE 7.3: Aktivität 3: Eingrenzen der Quell- und Zielsprachen.

klar definiert sein. Diese Aktivität kann parallel zu der Aktivität Erfassen von Terminologien und Konzepten durchgeführt werden.

Aktivität 4: Bündeln und Aufbereiten des gesammelten Wissens

Während bei der zweiten und dritten Aktivität die schrittweise Erfassung von Wissen erfolgt, wird in diesem Schritt das Wissen kategorisiert, priorisiert und u. U. abstrahiert. Ziel der Konsolidierung ist es, die Voraussetzungen für das Finden passender Operatoren für die Beschreibung der Überführung der Quellsprache(n) in Zielsprache(n) unter Berücksichtigung der Anwenderzielgruppe mit deren Präferenzen zu schaffen. Ebenso soll mit dieser Aktivität der Anforderungskatalog durch weitere, vor allem funktionale, Anforderungen ergänzt werden.

Für diesen kreativen Prozess können Methoden, die zur Erfassung von Produktfamilien entstammen, angewendet werden. Beispielsweise können die aus der Methode FODA bekannten Merkmaldiagramme genutzt werden, um Spracheigenschaften systematisch zu dokumentieren, die die Zielsprache haben muss, kann oder darf. Eine Priorisierung, Kombinierung, Multiplizität und gegenseitige Ausschlüsse können darüber hinaus spezifiziert werden. Diese Methode sagt absolut noch nichts über die spätere Umsetzung der Merkmale aus. Hauptaufgabe dieser Methode ist es, konzeptionelle Gemeinsamkeiten und Unterschiede zwischen den Ausprägungen von Sprachbestandteilen zu erfassen. Wie diese Merkmale implementiert werden, wird erst in

der Entwurfsphase festgelegt. Insbesondere Abhängigkeiten zwischen Merkmalen, die sich gegenseitig ausschließen oder bedingen, sind für die spätere Definition der höheren Operatoren wichtig. Ebenso sollten Standardeinstellungen an dieser Stelle definiert werden.

Zweck	Erkennung, Gruppierung und Beschreibung von Strukturen und konzeptionellen Zusammenhängen
Rollen	- Domänenanalytiker (verantwortlich) - Domänenexperte - Sprachdesigner - Endanwender
Input-Artefakte	- Anforderungskatalog - Schemata der Quell- und Zielsprachen - Glossar von Begriffen - Übersicht über wesentliche Konzepte
Output-Artefakte	- Begriffsnetze/Domänenmodelle - verfeinerter Anforderungskatalog
Methoden/Werkzeuge	- FODA, ODM

Tabelle 7.4: Aktivität 4: Bündeln und Aufbereiten des gesammelten Wissens.

Aktivität 5: Definieren der domänenspezifischen Sprachkomponente

Für die Sprachfindung sind im Wesentlichen die variablen Merkmale interessant. Gemeinsame Merkmale werden durch feste Operatorkombinationen in den Ebenentransformationsdefinitionen realisiert. Die variablen, offenen Stellen müssen durch Informationen aus den Transformationsdefinitionen der höheren Operatoren der domänenspezifischen Sprachkomponente gefüllt werden. Eine höhere Transformationsdefinition stellt somit im gewissen Sinne eine Konfiguration der Ebenentransformation dar.

Die entworfenen Operatoren der höheren, problemspezifischen Sprachkomponente sollten dabei immer die relevanten Konzepte und Termini der Transformationsdomäne aufnehmen. Zudem sollte eine möglichst vollständige Überdeckung dieser erreicht werden. Technische, von einer Basistransformationssprache abhängige, Operatoren sind sowohl für das fachliche Verständnis der Anwenderzielgruppe als auch für die technische Weiterentwicklung, z. B. für eine spätere Migration, nachteilig. Des Weiteren sollte darauf geachtet werden, dass die Operatoren der Sprachkomponente so kompakt wie möglich sind und keine unnötigen Redundanzen aufweisen.

Die entworfene domänenspezifische Sprachkomponente unterliegt typischerweise einer Evolution. Beispielsweise ist es nicht ungewöhnlich, dass bestimmte Operatoren im ursprünglichen Entwurf nicht abstrakt genug sind und daher angepasst werden müssen. Grundlegende Änderungen sowohl in der Struktur als auch in der Bedeutung von Operatoren können weitreichende Änderungen der Plattform und somit auch der Ebenentransformationsdefinitionen nach sich ziehen. Insbesondere die Ebenentransformationsdefinitionen sollten daher erst abgeleitet werden, wenn das Referenzszenario mit höheren Transformationsdefinitionen und deren Implementierung sowohl bei Anwendern als auch beim Architekten plausibel und für gut befunden werden.

Nach dem kreativen Prozess der Sprachfindung ist das Erstellen eines Schemas eine systematische Tätigkeit. Ein Schema legt die Syntax der Sprachkomponente formal fest. Ebenso wird ein

Namensraum definiert, mit dem die Sprachkomponente eindeutig identifiziert und somit von anderen Sprachkomponenten unterschieden werden kann. Da XML Schema eine weit verbreitete, mächtige und relativ leicht weiterverarbeitbare Schemasprache ist, wird diese als Beschreibungsform bevorzugt. Nichtsdestotrotz können andere Vorschläge für Schemasprachen ebenso verwendet werden. Diese müssen anschließend in eine Schemadefinition in XML Schema überführt werden, um das Generatorsystem XOpGen in der jetzigen Implementierung nutzen zu können.

Neben der Strukturbeschreibung können innerhalb der Annotation von XML Schema weitere Informationen, bspw. die Bedeutung der Sprachkonstrukte, in textueller Form hinterlegt werden. Dadurch kann ebenso die Semantik informell in textueller Form aufgenommen werden. Mit Hilfe der Annotation ist es aber ebenso möglich, zusätzliche, nicht mit dem XML Schema ausdrückbare Einschränkungen hinzuzufügen. Solche Einschränkungen sind bspw. gegenseitige Abhängigkeiten von Elementen und/oder Attributen. Hierzu eignen sich die Annahmen und Ausnahmen, die mit Schematron spezifiziert werden können, ideal.

Bei der Erstellung sollte mit der Deklaration des Wurzelelementes begonnen werden und anschließend das Metamodell systematisch in einer zuvor festgelegten Abarbeitungsreihenfolge abgearbeitet werden. Die Typdefinitionen können dabei parallel zur oder erst nach dem Abschluss der Elementdeklaration erfolgen. Verschiedene Werkzeuge, sowohl proprietäre als auch offene, helfen bei der Erstellung von Schemadefinitionen in XML Schema.

Zweck	formale Beschreibung der Operatoren der Sprachkomponente
Rollen	- Sprachdesigner (verantwortlich) - Domänenanalytiker - Domänenexperte - Endanwender
Input-Artefakte	- verfeinerter Anforderungskatalog - Schemata der Quell- und Zielsprachen - Glossar von Begriffen - Übersicht über wesentliche Konzepte
Output-Artefakte	- annotiertes Schema der Sprachkomponente
Methoden/Werkzeuge	- Schema-Editor

Tabelle 7.5: Aktivität 5: Definieren der domänenspezifischen Sprachkomponente.

Aktivität 6: Definieren des Referenztransformationsszenarios

Um eine domänenspezifische Sprachkomponente nicht „auf der grünen Wiese“ zu entwerfen, sollte am besten ein Beispielfall betrachtet werden, an dem man die Ergonomie und Angepasstheit des Sprachentwurfs kontinuierlich messen kann. Ein Referenzszenario dient u. a. für diesen Zweck. Der konkrete fachliche Gehalt eines Referenzszenarios ist im Grunde irrelevant. Vielmehr geht es darum, die Transformationsaufgaben in einem Anwendungsbereich wiederzuspiegeln und somit eine hinreichende Überdeckung von Operatoren und ihren Kombinationen zu erzielen. Dennoch wird i. d. R. ein fachlich einigermaßen sinnvoller Anwendungsfall im Kleinen formuliert und später umgesetzt, um die Zweckdienlichkeit zu ermitteln.

Ein echter Anwendungsfall sollte zumindest in den frühen Phasen vermieden werden. Zwar

kann für diesen u. U. bereits eine bestehende Referenzimplementierung verwendet und somit der zusätzliche Entwicklungs- und Pflegeaufwand für das Referenzszenario vermieden werden, allerdings besteht die Gefahr, den Überblick zu verlieren. Ein echter Anwendungsfall ist eben nicht mehr fachlich minimalistisch. Zudem fehlt oftmals die Überdeckung wichtiger Operatoren, da sie zur Lösung dieses Anwendungsfalls nicht benötigt werden.

Zweck	Wiederspiegelung der Transformationsaufgaben innerhalb des Anwendungsbereiches
Rollen	- Sprachdesigner (verantwortlich) - Domänenanalytiker - Domänenexperte - Plattformarchitekt - Referenzimplementierer - Plattformentwickler - Ebenentransformationsentwickler
Input-Artefakte	- Schemata der Sprachkomponente, Quell- und Zielsprachen - bestehende Dokumentationen, Implementierungen, etc.
Output-Artefakte	- detaillierte Beschreibung des Referenztransformationsszenarios
Methoden/Werkzeuge	- Schema-Viewer - Code-Editor

Tabelle 7.6: Aktivität 6: Definieren des Referenztransformationsszenarios.

Aktivität 7: Erstellen der Referenztransformationsdefinition

Mit den Mitteln der domänenspezifischen Sprachkomponente wird das Transformationsbeispiel des Referenzszenarios ausgedrückt. Für die Referenztransformationsdefinition werden entsprechende, im Idealfall alle möglichen, Kombinationen von Operatoren verwendet. Gemeinsam mit der Referenzimplementierung wird die Syntax und indirekt die Semantik bis auf Detailebene exemplarisch gezeigt.

Eine Evolution der domänenspezifischen Sprachkomponente, insbesondere in den frühen Phasen der Findung und Stabilisierung, wird in aller Regel Rückwirkungen auf die Referenztransformationsdefinition der Sprachkomponente und gegebenenfalls auf deren Referenzimplementierung und deren Plattform haben. Dies ist mit einem iterativen inkrementellen Entwicklungsansatz gleichzusetzen.

Zweck	Anwendung der Operatoren am Referenztranformationsszenario zur Überprüfung der Zweckdienlichkeit der definierten Operatoren
Rollen	- Sprachdesigner (verantwortlich) - Referenzimplementierer
Input-Artefakte	- Schemata der Sprachkomponente, Quell- und Zielsprachen - detaillierte Beschreibung des Referenztransformationsszenarios
Output-Artefakte	- Transformationsdefinition des Referenzszenarios mit den Mitteln der Sprachkomponenten
Methoden/Werkzeuge	- Schema-Viewer - Code-Editor

Tabelle 7.7: Aktivität 7: Erstellen der Referenztransformationsdefinition.

Aktivität 8: Erstellen der Referenzimplementierung auf der Plattform

Die Referenzimplementierung kann aus existierenden Beispielimplementierungen extrahiert oder vollkommen neu von Hand entwickelt werden. Unabhängig davon hat sie aber einen höheren Anspruch als ein einfaches, isoliertes Beispiel. Im Zusammenspiel mit der Referenztransformationsdefinition zeigt sie die Anwendung und Umsetzung der domänenspezifischen Sprachkomponente. Diese zweiteilige Referenz gibt den Übergang vom Abstrakten (höhere, fachliche Operatoren) zum Konkreten (niedrigere, eher technische Operatoren) auf eine bestimmte Plattform wieder. Aus beiden kann die Ebenentransformationsdefinition abgeleitet werden. Zugleich zeigt die Referenzimplementierung ebenso die Verwendung der Plattform. Im Zuge der Erstellung der Referenzimplementierung kann die Plattform selbst erweitert werden.

Sobald die Ebenentransformationsdefinitionen vorhanden sind, wird die Referenzimplementierung auf die manuell entwickelte Fachlogik reduziert. Die restlichen Bestandteile werden dann aus den höheren Transformationsdefinitionen generiert.

Zweck	Umsetzung des Referenzszenarios auf der Plattform
Rollen	- Referenzimplementierer (verantwortlich) - Plattformentwickler
Input-Artefakte	- Schemata der Quell- und Zielsprachen - detaillierte Beschreibung des Referenztransformationsszenarios - evtl. bestehende Implementierungen und Dokumentationen
Output-Artefakte	- Transformationsdefinition des Referenzszenarios mit den Mitteln der Plattform
Methoden/Werkzeuge	- Schema-Viewer - Code-Editor

Tabelle 7.8: Aktivität 8: Erstellen der Referenzimplementierung auf der Plattform.

Aktivität 9: Erweitern der Plattform

Die Plattform hat die Aufgabe, die Umsetzung der domänenspezifischen Sprachkomponente zu stützen. Unter dem Begriff Plattform werden deshalb in diesem Zusammenhang sowohl Bibliotheken in einer Basistransformationssprache als auch die Basistransformationssprache selbst verstanden. Darüber hinaus werden auch bereits implementierte, höhere Operatoren oder Sprachkomponenten dazugezählt. Die Plattform stellt somit die Grundlage für eine Implementierung dar. Um die Ebenentransformationsdefinitionen möglichst einfach zu halten, bildet die Erweiterung der Zielplattform bspw. durch weitere Bibliotheken ein wichtiges Mittel. Je mächtiger die Plattform ist, umso einfacher ist die Ebenentransformationsdefinition zu erstellen. Im Extremfall kann die Pattform die Rolle eines Interpreters einnehmen, so dass die Ebenentransformation selbst trivial ist. Umgekehrt gilt, je größer der Unterschied zwischen den Konzepten der domänenspezifischen Sprachkomponente und den Konzepten der Zielplattform ist, desto komplexer sind die notwendigen Ebenentransformationen. Dies sollte vermieden werden, zumal Komplexität auf der Metaebene schwerer zu beherrschen ist als auf der konkreten Ebene der Plattform. Generell gilt, dass feste generische oder generalisierbare Anteile Teil der Zielplattform sein sollten.

Die Grenzen zwischen Plattform und Generat der Ebenentransformation können sich mit zunehmenden Verständnis über die Domäne und mit der einhergehenden Evolution der domänenspezifischen Sprachkomponente verschieben – sowohl in die eine als auch in die andere Richtung. Ebendeshalb sollte die Entwicklung der Plattform einem iterativen, inkrementellen Entwicklungsansatz folgen.

Zweck	Vereinfachung der Ebenentransformationsdefinition von den höheren Operatoren der Sprachkomponente zu den Operatoren der Plattform
Rollen	- Plattformentwickler (verantwortlich) - Ebenentransformationsentwickler
Input-Artefakte	- Schemata der Quell- und Zielsprachen - detaillierte Beschreibung des Referenztransformationsszenarios - evtl. bestehende Implementierungen und Dokumentationen
Output-Artefakte	- Code der Plattformerweiterung - Dokumentation der Erweiterung
Methoden/Werkzeuge	- Schema-Viewer - Code-Editor

Tabelle 7.9: Aktivität 9: Erweitern der Plattform.

Aktivität 10: Ableiten des Ebenentransformators

Hauptaufgabe dieser Aktivität ist die Entwicklung einer Ebenentransformationsdefinition. Die Ebenentransformation überführt eine mit den Mitteln der domänenspezifischen Sprachkomponente gegebene Transformationsdefinition in eine Transformationsdefinition, die entweder elementar und deshalb direkt ausführbar ist oder ihrerseits durch bereits existierende Ebenentransformationsdefinitionen in eine ausführbare Form einer Basistransformationssprache transformiert wird. Somit definiert die Ebenentransformationsdefinition die Semantik (die Interpretation) bzgl. einer konkreten Zielplattform.

Die Ebenentransformationsdefinitionen können zunächst aus der Referenztransformationsdefinition und der Referenzimplementierung abgeleitet werden. Wie die Aktivität im Einzelnen abläuft, ist von der verwendeten Transformationsmethode abhängig. Wird bspw. für diesen Schritt der Template-basierte Ansatz gewählt, so können die statischen Teile der Templates direkt aus der Referenzimplementierung übernommen werden. Für die dynamischen Teile werden in den Templates offene Stellen angelegt, die bei der Initialisierung durch Informationen aus der Referenztransformationsdefinition gefüllt werden. Die offenen Stellen der Templates können Metacode enthalten, der die weitere Verarbeitungsreihenfolge durch Iterationen oder Aufrufe anderer Templates steuert.

Losgelöst davon sollte, wann immer möglich, „gut verständlicher“ Code erzeugt werden. Es sollten die gleichen Qualitätsmaßstäbe wie bei nichtgeneriertem Code angelegt werden, denn es ist in vielen Fällen unrealistisch anzunehmen, dass Entwickler den generierten Code nie zu sehen bekommen. Zwar muss die aus der Ebenentransformation erzeugte niedrigere Transformationsdefinition nicht geändert, aber mglw. in weiteren Ebenentransformationsschritten in eine elementare Transformationsdefinition überführt werden. Um die Verständlichkeit des Generates zu verbessern, können bspw.

- Kommentare generiert werden, die auf Informationen der höheren Transformationsdefinition basieren,
- nachgelagerte *Pretty Printer* benutzt werden, die richtige Einrückungen zur Strukturierung des Generates vornehmen, und
- speziell für die Fehlersuche Metadaten erzeugt werden, die bspw. die verwendete Transformationsregel in der Ebenentransformationsdefinition sowie den Sprachkonstrukt in der höheren Transformationsdefinition identifizieren. Informationen wie Zeitstempel, Version und Ort der Ebenentransformationsdefinition sind ebenso sinnvolle Ergänzungen.

Die nahezu einzige Ausnahme besteht bei der Erzeugung von performanzoptimierten Transformationsdefinitionen. Hier spielt die Verständlichkeit eine untergeordnete Rolle.

Neben der Einhaltung von Stilrichtlinien für die erzeugte Transformationsdefinition ist es ratsam, für die Ebenentransformationsdefinitionen selbst Grundprinzipien der Software-Entwicklung anzuwenden, wie Strukturierung, Modularisierung, regelmäßiges Refaktorisieren und Versionieren. Beispielsweise führt eine Modularisierung der Ebenentransformationsdefinitionen zu einer besseren Wiederverwendbarkeit. Je nach Transformationssprache, die zur Beschreibung der Ebenentransformation eingesetzt wird, kann dies durch Subroutinen, Module oder lose gekoppelte Komponenten erreicht werden, die für die Erzeugung klar getrennter Bestandteile zuständig sind. Diese können dann einzeln ausgetauscht werden. Wie im Abschnitt 5.3.4 gezeigt, können für die Ebenentransformation selbst höheren Operatoren bzw. Sprachkomponenten und somit ihre Vorteile verwendet werden.

Zweck	Entwicklung des Ebenentransformators
Rollen	- Ebenentransformationsentwickler (verantwortlich) - Plattformentwickler
Input-Artefakte	- Schema der Sprachkomponente - Transformationsdefinition des Referenzszenarios mit den Mitteln der Sprachkomponente und mit den Mitteln der Plattform - evtl. bestehende Dokumentationen der Plattform
Output-Artefakte	- Ebenentransformator - Dokumentation des Ebenentransformators
Methoden/Werkzeuge	- Schema-Viewer - Code-Editor

TABELLE 7.10: Aktivität 10: Ableiten des Ebenentransformators.

Aktivität 11: Generieren des Ebenen- und Inputvalidierers

Dieser Schritt ist ein rein mechanischer Vorgang, der auf der Grundlage des Schemas und der Konfiguration des Generatorsystems XOpGen erfolgt. Es werden Validierungskomponenten erzeugt, die in die Infrastruktur der Ebenentransformationsdefinition über feste Schnittstellen eingebunden werden.

Die Konfiguration von XOpGen ist notwendig, wenn Anpassungen bzgl. der Standardkonfiguration durchgeführt werden müssen. Solche Änderungen können die Art und den Umfang der

Fehlermeldungen für die zu erzeugenden Validierungskomponenten betreffen. Insbesondere für den Inputvalidierer, der erst während der elementaren Transformation ausgeführt wird und daher auf eine Basistransformationssprache abgebildet werden muss, besteht die Möglichkeit die Zielplattform einzustellen. Gerade diese Justierung ermöglicht die Unabhängigkeit von einer konkreten Zielplattform. In XOpGen werden konfigurierbare Variationspunkte mit Hilfe von Hooks, die überschrieben werden können, realisiert (siehe Abschnitt 6.3.1 und Abschnitt 6.3.2).

Zweck	Erstellung der Validierungskomponenten, die entsprechende Überprüfungen vornehmen
Rollen	- Plattformarchitekt (verantwortlich)
Input-Artefakte	- Schema der Sprachkomponente
Output-Artefakte	- Validierungskomponenten
Methoden/Werkzeuge	- XOpGen

Tabelle 7.11: Aktivität 11: Generieren des Ebenen- und Inputvalidierers.

Aktivität 12: Generieren der Dokumentation

Die Dokumentation der Syntax und Semantik einer Sprachkomponente ist essentiell für deren Anwendung. Das Schema der Sprachkomponente bietet alle Informationen, um eine Dokumentation der Syntax zu generieren. Wird die Bedeutung der Operatoren im Schema annotiert, so können ebenso diese semantischen Informationen an den entsprechenden Stellen der Dokumentation generiert werden. Diverse Dokumentationsgenerationswerkzeuge existieren bereits. Sie unterstützen unterschiedliche Repräsentationsformen und Formate. Einige Werkzeuge (z. B. DocFlex/XML-XSDDoc [Rud08], xnsdoc [bul07], xsddoc [Kur05]) erzeugen bspw. eine JavaDoc-ähnliche Repräsentation im HTML-Format. Wiederum andere Werkzeuge (z. B. XS3p [DST02], Document! X [Inn07]) unterstützen eigene proprietäre Repräsentationsformen und können andere Formate (z. B. RTF) erzeugen.

Zweck	Erzeugung der Dokumentation zum Finden und Verstehen der definierten Operatoren
Rollen	- Plattformarchitekt (verantwortlich)
Input-Artefakte	- Schema der Sprachkomponente
Output-Artefakte	- Dokumentation der Sprachkomponente
Methoden/Werkzeuge	- DocFlex/XML, xnsdoc, xsddoc

Tabelle 7.12: Aktivität 12: Generieren der Dokumentation.

Aktivität 13: Erstellen des Editors

Da die Operatoren von höheren Sprachkomponenten XML-basiert sind, können standardmäßige XML-Editoren zum Einsatz kommen. Einige Standardwerkzeuge wie das Eclipse WTP (*Web Tool Project*) [ecl08b] können über Erweiterungspunkte angepasst werden. Beispielsweise können bestimmte Zuweisungen von Namensräumen und Referenzen auf Schemata (in XML Schema oder DTD) angelegt werden. Danach kann der angepasste Editor entsprechend den Einschränkungen der Schemata während des Eingebens auf Wunsch Vorschläge, Hinweise und Fehlermeldungen für die Operatoren der Sprachkomponente anzeigen.

Wird durch die domänenspezifische Sprachkomponente eine spezielle Zielgruppe angesprochen, ist es durchaus üblich und angebracht, eine eigene Umgebung für die Eingabe zu erstellen, um bspw. die Ergonomie und damit die Effizienz weiter zu steigern. Ein Generatorwerkzeug für einen solchen Ansatz ist das EMF (*Eclipse Modeling Framework*) [ecl08a]. Es generiert in der Standardeinstellung eine einfache Baumeingabe in der neue, nur vom Schema erlaubte Knoten hinzugefügt werden können.

Zweck	Anpassen des Editors an die definierte Sprachkomponente
Rollen	- Plattformarchitekt (verantwortlich)
Input-Artefakte	- Schema der Sprachkomponente
Output-Artefakte	- Konfigurations- und Implementierungsdateien
Methoden/Werkzeuge	- Eclipse WTP, EMF

TABELLE 7.13: Aktivität 13: Erstellen des Editors.

Aktivität 14: Integrieren in die Ebenentransformationsinfrastruktur

Die Generation der Validierungskomponenten wird gegen feste Schnittstellen der generischen Infrastruktur vorgenommen. Ebenso wird die eigentliche Beschreibung der Ebenentransformation gegen feste Schnittstellen programmiert. Deshalb reicht in den meisten Fällen ein einfaches Importieren, um diese Komponenten zu verbinden. Nichtsdestotrotz kann es Fälle geben, in denen bspw. bereits existierende Codebestandteile integriert und somit entsprechende Mapping-Komponenten für die Infrastruktur entwickelt werden müssen.

Zweck	Verkoppelung der Infrastruktur mit den erstellten und generierten Komponenten
Rollen	- Plattformarchitekt (verantwortlich) - Ebenentransformationsentwickler
Input-Artefakte	- Ebenentransformator - Validierungskomponenten - generische Infrastruktur der Transformationsdefinition
Output-Artefakte	- Integrationsdatei
Methoden/Werkzeuge	- Code-Editor

TABELLE 7.14: Aktivität 14: Integrieren in die Ebenentransformationsinfrastruktur.

Aktivität 15: Test, Installation und Nutzung

In dieser komplexen Aktivität erfolgt zunächst der Test, dann die Installation und anschließend die Nutzung. Obwohl es angebracht ist, nach jeder Durchführung einer Aktivität das Ergebnis gegen die aufgestellten Anforderungen zu prüfen, erfolgt in diesem Schritt ein expliziter und umfassender Test. Dieser überprüft, ob die Umsetzung der Sprachkomponente sowohl alle funktionalen als auch alle nichtfunktionalen Anforderungen erfüllt. Bei Nichterfüllung werden Gründe ermittelt und Lösungen gesucht. Unter Umständen wird eine erneute Iteration dieses Vorgehensmodells instanziiert.

Wird der Test erfolgreich beendet, müssen mglw. Installationsroutinen zusammengestellt werden, die zum einen die benötigten Dateien, Bibliotheken und Komponenten zur Verfügung stellen und

zum anderen den Vorgang der Installation selbst unterstützen. Die Unterstützung umfasst bspw. die Prüfung über die Eignung der (System-)Umgebung, das eigentliche Kopieren der Daten sowie die Konfiguration abhängig von der Umgebung bzw. von den Benutzereingaben.

Die Nutzung und Überwachung des aktiven Anwenderentwicklungsbetriebs kann wertvolle Hinweise auf mögliche Verbesserungen und Optimierungspotenziale geben. Diese Informationen können eine Überarbeitung der Sprachkomponente im Rahmen einer neuen Instanz des Vorgehensmodells erfordern.

Zweck	Vorbereitung für die Freigabe und Nutzung
Rollen	- Plattformarchitekt (verantwortlich) - Qualitätssicherer - Endanwender - Projektleiter
Input-Artefakte	- sämtliche erstellten und generierten Komponenten (inkl. Dokumentation, Editorkonfigurationen, etc.) - Referenzimplementierung und andere Anwendungsbeispiele zum Testen
Output-Artefakte	- Testprotokoll - Installationsroutinen
Methoden/Werkzeuge	- Testumgebungen

Tabelle 7.15: Aktivität 15: Test, Installation und Nutzung.

7.2 Bottom-up-Vorgehensmodell

Das Bottom-up-Vorgehensmodell setzt bestehende Implementierungen mit einer Transformationssprache oder Sprachkomponente voraus. Diese werden untersucht und neue oder zu verändernde Operatoren identifiziert (Analyse), Operatoren definiert bzw. angepasst (Entwurf) und umgesetzt (Implementierung). Abbildung 7.2 zeigt den zeitlichen Ablauf der einzelnen Aktivitäten innerhalb dieser Phasen. Es wurde erneut bewusst nur der Normalfall modelliert und bspw. auf Iterationszyklen und implizite Testaktivitäten im Vorgehensmodell verzichtet. Die einzelnen Aktivitäten und deren Ergebnistypen werden nun im Detail diskutiert.

Aktivität 1: Identifizieren potentieller Operatoren

Ziel dieser Aktivität ist es, auf Grundlage bereits entwickelter Transformationsdefinitionen potentielle Operatoren zu identifizieren, um immer wiederkehrenden, schematisch gleichen Code zu vermeiden. Für die Identifikation potentieller Operatoren sollten zwei Fragen gestellt werden:

Sind in den vorhandenen Transformationsdefinitionen Operatorkombinationen enthalten, die immer wieder gleich auftreten oder gleich strukturiert sind, aber noch nicht als höhere Operatoren ganz oder nur teilweise erfasst wurden? Wäre ein höherer Operator für diesen schematischen Code einfach und kompakt oder würde dieser mglw. keine wesentliche Vorteile mit sich bringen, da diese Lösung nicht verständlicher oder kürzer als die bisherige wäre?

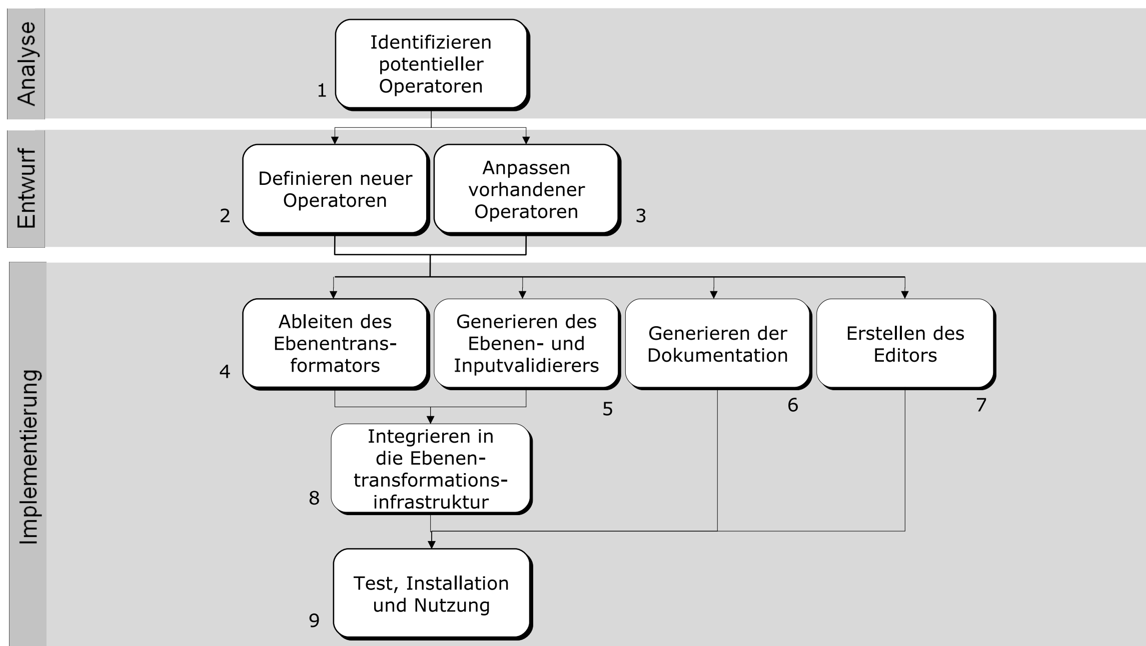


ABBILDUNG 7.2: Das Bottom-up-Vorgehensmodell.

Da diese Abwägung oftmals subjektiv ist, setzt sie einiges an Erfahrung voraus. In den zunächst identifizierten Operatorkombinationen werden statische und variable Bestandteile bestimmt. Die variablen Bestandteile sind für die Definition der Operatoren von besonderer Bedeutung, da sie gerade in den höheren Operatoren konfiguriert werden. Die statischen Bestandteile sind für die Ableitung der Ebenentransformationsdefinition relevant. Sind in den potentiell zu ersetzenden Operatorkombinationen bereits höhere Operatoren enthalten, können diese u. U. weiter vervollständigt und damit abstrahiert werden. Auch hier ist Fingerspitzengefühl gefragt, da sich eine zu große Granularität negativ auf den Grad der Wiederverwendbarkeit auswirken kann. Ein neuer, eigener Operator kann dann sinnvoller sein.

Zweck	Finden von repetitiven Operatorkombinationen, da diese den Code aufblähen und aufgrund ihrer Redundanzen den Code fehleranfällig machen
Rollen	- Plattformarchitekt (verantwortlich)
Input-Artefakte	- bestehende Implementierung
Output-Artefakte	- Beschreibung der zu ersetzenden Operatorkombinationen
Methoden/Werkzeuge	- Code-Editor

TABELLE 7.16: Aktivität 1: Identifizieren potentieller Operatoren.

Aktivität 2: Definieren neuer Operatoren

Ein neuer Operator ist immer Teil einer Sprachkomponente. Zunächst muss in dieser Aktivität eine solche Sprachkomponente, die funktionell oder inhaltlich ähnliche Operatoren enthält, gefunden werden. Existiert keine solche Sprachkomponente muss diese inkl. ihres Namensraums neu definiert werden. Stehen bei der Neudefinition die Anwenderzielgruppe bzw. eine spezifische

Transformationsdomäne im Vordergrund, sollte eine Instanz des Top-down-Vorgehensmodells gebildet und ein entsprechendes Projekt durchgeführt werden.

Die bereits bestimmten variablen Codebestandteile bilden das Fundament des zu erstellenden Operators, denn der neue Operator muss gerade Informationen bereitstellen, um diese zu füllen oder konfigurieren zu können. Die Syntax des Operators wird durch ein Schema formal festgelegt. Mit Hilfe von Annotationen können textuelle Informationen über die Semantik und weitere mit XML Schema nicht formulierbare Einschränkungen hinterlegt werden.

Zweck	formale Beschreibung der hinzuzufügenden Operatoren
Rollen	- Sprachdesigner (verantwortlich) - Plattformarchitekt
Input-Artefakte	- Beschreibung der zu ersetzenden Operatorkombinationen - bestehende Dokumentationen und Schemata von Sprachkomponenten
Output-Artefakte	- annotiertes Schema der Sprachkomponente mit hinzugefügten Operatoren
Methoden/Werkzeuge	- Schema-Editor

Tabelle 7.17: Aktivität 2: Definieren neuer Operatoren.

Aktivität 3: Anpassen vorhandener Operatoren

In dieser Aktivität werden bereits vorhandene Operatoren vervollständigt oder erweitert, da die ursprüngliche Definition bestimmte Kriterien oder Notwendigkeiten der Anwender nicht erfüllt. Oftmals tritt bspw. der Fall ein, dass der bisherige Entwurf nicht abstrakt genug definiert wurde. Ein schwerer zu identifizierendes Problem ist der entgegengesetzte Fall. In diesem sind die Operatoren zu grobgranular und infolgedessen nicht oder nur selten anwendbar.

Unabhängig vom Grund muss die Änderung des Operators im Schema vorgenommen werden, in dem die Syntax des ursprünglichen Operators festgelegt wurde. Sind darin weitere Annotationen enthalten, müssen sie ebenso angepasst werden.

Zweck	formale Beschreibung des anzupassenden Operatoren
Rollen	- Sprachdesigner (verantwortlich) - Plattformarchitekt
Input-Artefakte	- Beschreibung der zu ersetzenden Operatorkombinationen - bestehende Dokumentationen und Schemata von Sprachkomponenten
Output-Artefakte	- annotiertes Schema der Sprachkomponente mit angepassten Operatoren
Methoden/Werkzeuge	- Schema-Editor

Tabelle 7.18: Aktivität 3: Anpassen vorhandener Operatoren.

Aktivität 4: Ableiten des Ebenentransformators

Aus den identifizierten statischen und variablen Codebestandteilen sowie aus der (angepassten) Definition des Operators wird die Ebenentransformationsdefinition abgeleitet. Wie die Ableitung im Konkreten umgesetzt wird, ist von der eingesetzten Transformationsmethode abhängig.

Losgelöst davon muss die Ebenentransformationsdefinition die statischen Bestandteile erzeugen und die variablen Bestandteile durch Werte aus den höheren Operatoren belegen oder füllen.

Die Ebenentransformationsdefinition sollte so angelegt sein, dass die durch sie erzeugte Implementierung gleiche Stilrichtlinien einhält, die für nichtgenerierten Code verwendet werden. Ebenso sollte bei der Entwicklung der Ebenentransformationsdefinitionen von den Grundprinzipien der Software-Entwicklung Gebrauch gemacht werden (siehe Ableitung des Ebenentransformators in Abschnitt 7.1).

Zweck	Anpassen des Ebenentransformators
Rollen	- Ebenentransformationsentwickler (verantwortlich) - Plattformentwickler
Input-Artefakte	- Schema der Sprachkomponente mit hinzugefügten oder angepassten Operatoren - Beschreibung der zu ersetzenden Operatorkombinationen - evtl. dazugehörige Anwendungsbeispiele der hinzugefügten oder angepassten Operatoren
Output-Artefakte	- Ebenentransformator - Dokumentation des Ebenentransformators
Methoden/Werkzeuge	- Schema-Viewer - Code-Editor

TABELLE 7.19: Aktivität 4: Ableiten des Ebenentransformators.

Aktivität 5: Generieren des Ebenen- und Inputvalidierers

Diese Aktivität entspricht der Generation des Ebenen- und Inputvalidierers von Abschnitt 7.1. Zunächst wird das Generatorsystem XOpGen konfiguriert und anschließend die automatische Generierung durchgeführt.

Zweck	Anpassen der Validierungskomponenten, die entsprechende Überprüfungen vornehmen
Rollen	- Plattformarchitekt (verantwortlich)
Input-Artefakte	- Schema der Sprachkomponente
Output-Artefakte	- Validierungskomponenten
Methoden/Werkzeuge	- XOpGen

TABELLE 7.20: Aktivität 5: Generieren des Ebenen- und Inputvalidierers.

Aktivität 6: Generieren der Dokumentation

Diese Aktivität entspricht der Generation der Dokumentation von Abschnitt 7.1. Die Generation erfolgt auf Basis der Informationen, die im Schema niedergelegt wurden.

Aktivität 7: Erstellen des Editors

Neben der Dokumentation kann ebenso der Editor für die neu definierten oder angepassten Operatoren modifiziert werden. Dies wird sicherlich nur in Ausnahmefällen nach der Definition ein-

Zweck	Anpassen der Dokumentation zum Finden und Verstehen der hinzugefügten oder angepassten Operatoren
Rollen	- Plattformarchitekt (verantwortlich)
Input-Artefakte	- Schema der Sprachkomponente
Output-Artefakte	- Dokumentation der Sprachkomponente
Methoden/Werkzeuge	- DocFlex/XML, xnsdoc, xsddoc

Tabelle 7.21: Aktivität 6: Generieren der Dokumentation.

zelner zusätzlicher Operatoren geschehen. Die Aktivität "Erstellen des Editors" in Abschnitt 7.1 kann dennoch auch hier analog angewendet werden.

Zweck	Anpassung des Editors an die veränderte Sprachkomponente
Rollen	- Plattformarchitekt (verantwortlich)
Input-Artefakte	- Schema der Sprachkomponente
Output-Artefakte	- Konfigurations- und Implementierungsdateien
Methoden/Werkzeuge	- Eclipse WTP, EMF

Tabelle 7.22: Aktivität 7: Erstellen des Editors.

Aktivität 8: Integrieren in die Ebenentransformationsinfrastruktur

Die generierten Validierungskomponenten und der abgeleitete Ebenentransformator werden in dieser Aktivität in die generische Infrastruktur integriert. Da Validierungskomponenten und Ebenentransformator gegen feste Schnittstellen generiert bzw. programmiert werden, reicht ein einfaches Importieren aus, um eine Verknüpfung herzustellen. Sollen weitere Komponenten integriert werden, müssen in diesem Schritt entsprechende Mapping-Komponenten entwickelt werden.

Zweck	Verkoppelung der Infrastruktur mit den erstellten und generierten Komponenten
Rollen	- Plattformarchitekt (verantwortlich) - Ebenentransformationsentwickler
Input-Artefakte	- Ebenentransformator - Validierungskomponenten - generische Infrastruktur der Transformationsdefinition
Output-Artefakte	- Integrationsdatei
Methoden/Werkzeuge	- Code-Editor

Tabelle 7.23: Aktivität 8: Integrieren in die Ebenentransformationsinfrastruktur.

Aktivität 9: Test, Installation und Nutzung

In dieser Aktivität erfolgt zunächst ein abschließender Test sowie die Installation und Nutzung. Der Test sollte sowohl die einzelne Überprüfung des Operators (Operatortest) als auch eine Überprüfung mit anderen Operatoren (Integrationstest) umfassen. Für den Operatortest kann überprüft werden, ob die ursprünglichen Implementierungen (statische und variable Codebestandteile) aus den neu definierten bzw. angepassten Operatoren erzeugt werden können. Das

Zusammenspiel mit anderen Operatoren kann bspw. mit den bereits existierenden Transformationsdefinitionen erprobt werden. Diese können nun mit den neuen Operatoren geschrieben und anschließend das Ergebnis der Ebenentransformation gegen die ursprünglichen Transformationsdefinitionen getestet werden. Dies kann und darf jedoch nicht weitere Integrationstests mit anderen Operatoren ersetzen, da gerade die neuen Operatoren aus diesen ursprünglichen Transformationsdefinitionen erstellt wurden.

Erst nach einer Installation und unter Nutzung der neuen Operatoren in realen Problemstellungen kann der Reifegrad der Operatoren vollständig ermittelt werden (vgl. mit der Aktivität Test, Installation und Nutzung im Abschnitt 7.1). Ein hieraus resultierendes Feedback der Anwender kann eine weitere Iteration des Vorgehensmodells notwendig machen.

Zweck	Vorbereitung für die Freigabe und Nutzung
Rollen	- Plattformarchitekt (verantwortlich) - Qualitätssicherer - Endanwender
Input-Artefakte	- sämtliche erstellten und generierten Komponenten (inkl. Dokumentation, Editorkonfigurationen, etc.) - Anwendungsbeispiele zum Testen
Output-Artefakte	- Testprotokoll - Installationsroutinen
Methoden/Werkzeuge	- Testumgebungen

Tabelle 7.24: Aktivität 9: Test, Installation und Nutzung.

7.3 Rollenverteilung

Generell ist es notwendig, dass die Entwicklung von konkreten Anwendungen und die Entwicklung der Sprachkomponenten getrennt wird. Hierfür können mehrere Gründe angeführt werden. Zunächst einmal sind Anwendungsentwicklungen i. d. R. sowohl einem starken Termin- als auch Kostendruck untergeordnet. Somit kann bei einer verschmolzenen Entwicklung nicht auf die eventuellen Schwierigkeiten der Spracherweiterung Rücksicht genommen werden, so dass die Qualität, die entscheidend für den Erfolg ist, leidet. Zudem sind unterschiedliche Kenntnisse für Anwendungsentwicklung und Sprachentwicklung vonnöten.

In diesem Abschnitt wird auf die spezifische Rollenverteilung bei der (Weiter-)Entwicklung von Sprachkomponenten eingegangen. Für eine erfolgreiche Sprachentwicklung ist eine große Bandbreite an Erfahrungen, Kenntnissen und Fähigkeiten notwendig, welche sich typischerweise auf eine Gruppe von Personen verteilt. Einige Rollen unterscheiden sich nicht von traditionellen Anwendungsprojekten, während andere Rollen eine besondere Bedeutung einnehmen. Auf diese Rollen wird in erster Linie eingegangen.

Abbildung 7.3 zeigt die spezifischen Rollen, die im Top-down-Vorgehensmodell und im Bottom-up-Vorgehensmodell Aktivitäten durchführen. Da das Top-down-Vorgehensmodell für die Entwicklung von fachlich motivierten Sprachkomponenten angewendet wird, werden hier neben den

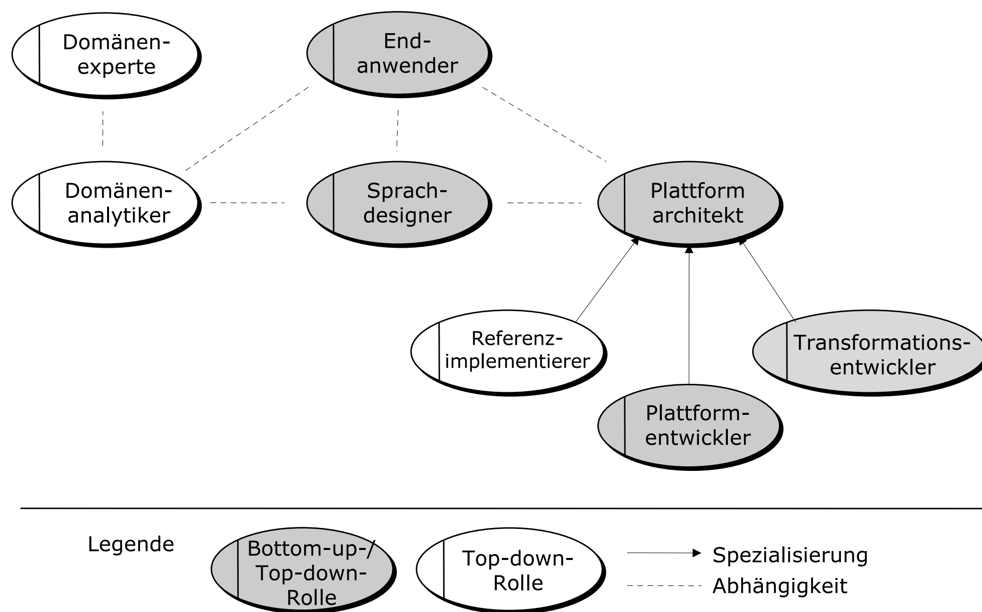


ABBILDUNG 7.3: Spezifische Rollen bei der (Weiter-)Entwicklung von Sprachen.

Rollen des Bottom-up-Vorgehensmodells (grau hinterlegte Rollen) zusätzliche Rollen (weiß hinterlegte Rollen) benötigt. Zu jeder Rolle werden im Folgenden kurz die Hauptaufgaben und wichtigsten Tätigkeiten beschrieben. Eine Person kann dabei natürlich mehrere Rollen übernehmen und umgekehrt kann eine Rolle auch von mehreren Personen ausgefüllt werden.

Domänenexperten

Um eine gute eigenständige domänenspezifische Transformationssprache zu erstellen oder lediglich domänenspezifische Sprachkomponenten zu entwerfen, wird ein Expertenwissen über die Domäne benötigt. Domänenexperten sind Personen, die bereits mehrere Anwendungen in der Domäne realisiert haben oder potentielle Anwender und Kunden, die Experten in Teilen der Domäne sind.

Domänenanalytiker

Der Domänenanalytiker ist verantwortlich für sämtliche Aktivitäten der Domänenanalyse (siehe Abschnitt 7.1). Dies lässt sich am besten durch eine Reihe von Workshops mit Domänenexperten, potentiellen Endanwendern, Anforderungsanalytikern und Plattformarchitekten iterativ erarbeiten.

Sprachdesigner

Der Sprachdesigner ist ein Experte im Finden von Operatoren und im Zusammenfügen dieser zu einer Sprachkomponente abhängig oder unabhängig von einer bestimmten Domäne. Er definiert die abstrakte und die konkrete Notationsform der Operatoren. Er muss dahingehend besondere

Fähigkeiten besitzen. Er ist das Bindeglied zwischen Domänenanalytikern auf der einen Seite und dem Plattformarchitekten auf der anderen Seite.

Plattformarchitekt

Der Plattformarchitekt ist für die Realisierung der gesamten technischen Architektur verantwortlich. Dazu gehören die Definition der Zielplattform, das Erstellen der Referenzimplementierung und die Umsetzung der Ebenentransformation. Entsprechend den Aufgaben lassen sich spezifische Ausprägungen des Plattformarchitekten benennen (Referenzimplementierer, Plattformentwickler und Ebenentransformationsentwickler, etc.).

Referenzimplementierer

Referenzimplementierer bilden die neuen Sprachkonstrukte auf eine gegebenen Zielplattform ab. Es werden die Referenztransformationsdefinitionen des Sprachdesigners aus der Domänenanalyse verwendet. Der Referenzimplementierer braucht somit sowohl Kenntnisse von der Domäne, Kenntnisse von der Zielplattform als auch Fähigkeiten für die Überführung des Referenzszenarios. Er leistet eine wichtige Vorarbeit für den Ebenentransformationsentwickler.

Plattformentwickler

Für die Rolle des Plattformentwicklers sind Personen geeignet, die weitgehende Erfahrungen mit dem Erstellen von Bibliotheken für Sprachen im Allgemeinen und Transformationssprachen im Besonderen haben. Sie stellen die Zielplattform zur Verfügung und müssen deshalb eng mit den Referenzimplementierern zusammenarbeiten, da die Referenzimplementierung sich auf die gegebene Zielplattform stützt und diese exemplarisch verwendet.

Ebenentransformationsentwickler

Ebenentransformationsentwickler müssen sowohl Transformationssprachen als auch Generierungswerkzeuge beherrschen. Sie definieren die Ebenentransformation, die bspw. unter zur Hilfenahme der Referenzimplementierung abgeleitet werden kann. Diese Transformationsdefinition erfolgt in Abstimmung mit der Anpassung der Zielplattform, so dass weder die Komplexität der Zielplattform noch die Komplexität der Ebenentransformation zu hoch wird und deshalb außer Kontrolle gerät.

Weitere Rollen

Neben den bereits erwähnten Rollen sind ebenso traditionelle Rollen relevant. Der Endanwender, es könnte auch ein Kunde sein, sollte so früh und so weit wie möglich in die Sprachentwicklung einbezogen werden. Anforderungsanalytiker, die zur Analyse und Beschreibung der anwendungsspezifischen Kundenanforderungen eingesetzt werden, bilden eine indirekte Schnittstelle zwischen Kunden der Anwendungsentwicklung und Domänenanalytikern bzw. Sprachdesignern.

Qualitätssicherer sind verantwortlich für die Entwicklung von Teststrategien und anderen Maßnahmen zur Qualitätssicherung. Projektmanager koordinieren den gesamte Sprachentwicklungsprozess mit Blickpunkt auf aktuelle Anwendungsentwicklungen und zukünftige Anwendungsentwicklungen für potentielle Kunden.

Die Matrix in Tabelle 7.25 ordnet abschließend den Aktivitäten der vorgeschlagenen Vorgehensmodelle Rollen zu.

Aktivität / **Rolle**	Domänenexperte	Domänenanalytiker	Sprachdesigner	Plattformarchitekt	Referenzimplementierer	Plattformentwickler	Ebenentransformationsentwickler	Qualitätssicherer	Endanwender	Projektleiter
Top-down-Vorgehensmodell										
Definieren der Anwenderzielgruppen	m	v								m
Erfassen von Terminologien und Konzepten	m	v							m	
Eingrenzen der Quell- und Zielsprache	m	v							m	
Bündeln und Aufbereiten des gesammelten Wissens	m	v	b						b	
Definieren der domänenspezifischen Sprachkomponente	b	b	v						b	
Definieren des Referenztransformationsszenarios	b	m	v	m	b	b	b			
Erstellen der Referenztransformationsdefinition			v		b					
Erstellen der Referenzimplementierung auf der Plattform					v	b				
Ableiten des Ebenentransformators						b	v			
Erweitern der Plattform (Bibliotheken etc.)						v	b			
Ableiten des Ebenentransformators						b	v			
Generieren des Ebenen- und Inputvalidierers				v						
Generieren der Dokumentation				v						
Erstellen des Editors				v						
Integrieren in die Ebenentransformationsinfrastruktur				v			m			
Test, Installation und Nutzung				v				m	m	b
Bottom-up-Vorgehensmodell										
Identifizieren potentieller Operatoren				v						
Definieren neuer Operatoren			v	m						
Anpassen vorhandener Operatoren			v	m						
Ableiten des Ebenentransformators						m	v			
Generieren des Ebenen- und Inputvalidierers				v						
Generieren der Dokumentation				v						
Erstellen des Editors				v						
Integrieren in die Ebenentransformationsinfrastruktur				v			m			
Test, Installation und Nutzung				v				m	m	

Legende: v = verantwortlich, m = mitwirkend, b = beratend

TABELLE 7.25: Aktivitäten/Rollen-Matrix des Top-down-Vorgehensmodells und des Bottom-up-Vorgehensmodells.

Teil III

Evaluation

Der dritte Teil dieser Arbeit widmet sich der Anwendung des vorgeschlagenen Frameworks. Das Framework wurde genutzt, um bspw. die Transformationssprache XSLT durch zusätzliche Sprachkomponenten zu erweitern. Einige von ihnen werden in diesem Teil vorgestellt. Die dabei bisher gewonnenen Erfahrungen werden diskutiert und anschließend einer Kosten/Nutzen-Analyse unterzogen.

Kapitel 8

Anwendung

Bevor Nutzen und Kosten im nächsten Kapitel ausführlich erörtert werden, wird die Anwendung der vorgeschlagenen Konzepte und deren Implementierungen exemplarisch an der spezifischen Transformationssprache XSLT gezeigt. Es wird sowohl die Erweiterung von XSLT als auch der Aufbau einer eigenständigen Transformationssprache, die auf XSLT aufsetzt, demonstriert.

Im ersten Abschnitt dieses Kapitels wird XSLT mit verschiedenen Sprachkomponenten erweitert. Ziel ist es, losgelöst von einer konkreten Problemstellung, generell eine größere Anzahl von potentiellen Spracheigenschaften zu unterstützen. Zur Veranschaulichung werden Operatoren einer Sprachkomponente sukzessive mit Hilfe des Bottom-up-Vorgehensmodells konstruiert. Im zweiten Abschnitt werden reale Anwendungsbeispiele vorgestellt. Auf eines wird dabei näher eingegangen. Für dieses wird mittels des Top-down-Vorgehensmodells eine problemspezifische Transformationssprache basierend auf XSLT entworfen.

8.1 Erweiterung von XSLT

In diesem Abschnitt steht die Erweiterung einer Transformationssprache im Fokus. Als Beispiel werden zusätzliche Sprachkomponenten für XSLT konzipiert. Grundsätzlich kann das gleiche Entwicklungsvorgehen ebenso bei anderen Transformationssprachen angewendet werden. Da die höheren Operatoren innerhalb der Sprachkomponenten jedoch grundlegend von der Transformationsmethode im Allgemeinen sowie den elementaren Operatoren einer Basistransformationssprache im Besonderen abhängen, können für zwei Basistransformationssprachen u. U. nicht die gleichen höheren Operatoren gebildet werden.

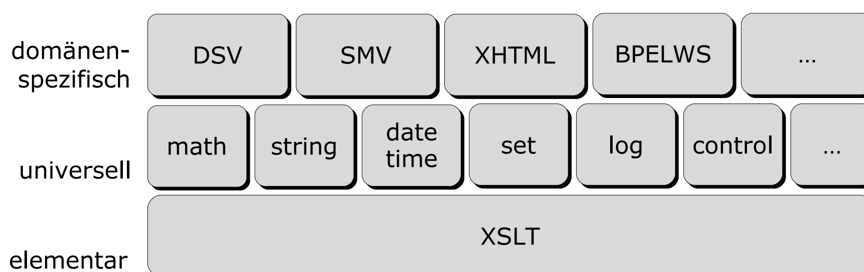


Abbildung 8.1: Zusätzliche Sprachkomponenten für XSLT.

Abbildung 8.1 stellt Beispiele von Sprachkomponenten vor, die für XSLT entwickelt wurden. Es wurden Sprachkomponenten mit universellen Operatoren und Sprachkomponenten mit domänenspezifischen Operatoren erstellt. Universelle Operatoren wurden dabei mit einer Bottom-up-Komposition aus elementaren XSLT-Operatoren zusammengesetzt. Domänenspezifische Operatoren, bei denen die Zielanwendergruppe im Vordergrund steht, wurden mit einem Top-down-Entwicklungsvorgehen definiert. Sie bauen auf universelle und elementare Operatoren bzw. deren Sprachkomponenten auf.

8.1.1 Universelle Operatoren

Die Ebene der universellen Operatoren bietet Funktionalität für kleine Transformationsaufgaben an, deren Lösungen schematisch immer wieder gleich aufgebaut sind. Die wiederverwendbaren Operatoren gestalten den Code wesentlich kompakter und führen letztendlich zu einer verständlicheren Transformationsdefinition. Die Operatoren werden inhaltlich und/oder funktionell in Sprachkomponenten zusammengefasst. Beispiele für Sprachkomponenten auf diesen Ebenen sind (vgl. mit [FP07a]):

- **math**
  Diese Sprachkomponente bietet Operatoren für mathematische Berechnungen, z. B. Berechnung des Minimums, Maximums, Durchschnitts, Summe, Multiplizität einer selektierten Menge von Werten oder Suche des kleinsten und größten Wertes aus einer selektierten Menge von Werten.
- **string**
  Diese Sprachkomponente bietet Operatoren für Zeichenkettenmanipulationen, z. B. Ersetzungen, Teilungen, Zusammenführungen oder Ausrichtungen von einer oder mehreren selektierten Zeichenketten.
- **date/time**
  Diese Sprachkomponente bietet Operatoren für Datums- und Zeitformatierungen und -umrechnungen, z. B. Ausgabe eines Datums, Zeit, Dauer, Jahr, Monat, Name eines Monats, Tag, Name eines Tags, Stunde, Minute, Sekunde aus einem selektierten Wert oder Addieren oder Subtrahieren von selektierten Datums- oder Zeitangaben von anderen Werten.
- **set**
  Diese Sprachkomponente bietet Operatoren für Manipulationen von Knotenmengen, z. B. die Differenz, Schnittmenge, Vereinigungsmenge, Ausschlussmenge, Gleichheit oder Anzahl von zwei oder mehreren selektierten Knoten.
- **log**
  Diese Sprachkomponente bietet Operatoren für das Logging, z. B. Ausgabe des Pfades oder Teilbaums zum aktuellen Kontextknoten, unterschiedliche Arten von Fehlermeldungen, die von Hinweisen bis hin zu groben Fehlern mit Programmabbruch reichen.
- **control**
  Diese Sprachkomponente bietet Operatoren für die Steuerung des Kontrollflusses, z. B.

Operatoren für Funktionsdefinitionen und -aufrufe, Schleifen (`for`, `while`, `do-while`) oder zur Festlegung von expliziten Traversierungsstrategien.

Um das Prinzip zu demonstrieren, wird im Folgenden besonderes Augenmerk auf die `control`-Sprachkomponente gelegt. Dabei wird speziell auf die Erstellung des `function`- und `call-function`-Operators eingegangen. Zunächst wird die Erstellung dieser Operatoren motiviert, anschließend die Syntax und Semantik beschrieben und abschließend die Implementierung und Verwendung erläutert. Dies zeigt exemplarisch eine vollständige Umsetzung mit Hilfe des Bottom-up-Entwicklungsvorgehens.

Problembeschreibung

XSLT bietet einen sehr einfachen, eingeschränkten Mechanismus, um Funktionen zu definieren und aufzurufen. Um eine Funktion in XSLT zu realisieren, wird ein benanntes Template definiert, welches über diesen Namen aufgerufen werden kann.

```
<xsl:template name="template-x">
  <xsl:param name="x"/>
  <xsl:param name="y"/>
  ...
</xsl:template>
```

Dieser Mechanismus hat einige, für eine funktionale Sprache ungewöhnliche Restriktionen. Um das benannte Template in XSLT identifizieren zu können, muss der Name des aufzurufenden Templates eindeutig sein. Es ist deshalb ein Fehler, wenn eine Transformationsdefinition mehr als ein Template mit dem gleichen Namen enthält.[1] Benannte Templates bieten zwar die Möglichkeit formale, benannte Parameter zu definieren (z. B. Parameter `x` und `y`), das Parameterübergabekonzept enthält aber Vereinfachungen, die leicht zu ungewolltem Verhalten führen können.

```
<xsl:call-template name="template-x">
  <xsl:with-param name="x" select="node()"/>
  <xsl:with-param name="z" select="."/>
  ...
</xsl:template>
```

Werden beim Aufruf eines Templates die dort definierten Parameter nicht übergeben (z. B. Parameter `y`) oder andere Parameter übergeben, die gar nicht definiert wurden (z. B. Parameter `z`), so kann der Parameter laut Spezifikation [Cla99a] einfach und ohne entsprechende Fehlermeldung ignoriert werden.

Mit Hilfe der Operatorhierarchie wird nun ein flexiblerer und zugleich sicherer Funktionsmechanismus entworfen, der auf den gezeigten elementaren Operatoren aufbaut. Der Anwender wird mit den neuen Operatoren in doppelter Hinsicht entlastet, indem er einerseits nicht mehr manuell erwähnte Fehlerquellen bspw. bei der Parameterübergabe aufspüren muss und andererseits ungebundener bei der Funktionsdefinition wird.

Abbildung 8.2 gibt eine Übersicht über die einzelnen Aktivitäten und deren Ergebnisdokumente für das Entwickeln einer neuen `control`-Sprachkomponente. Der `control`-Sprachkomponente

[1] Eine Ausnahme stellen gleich benannte Templates dar, die importiert wurden. Sie werden überdeckt von Templates mit höherer Importpriorität.

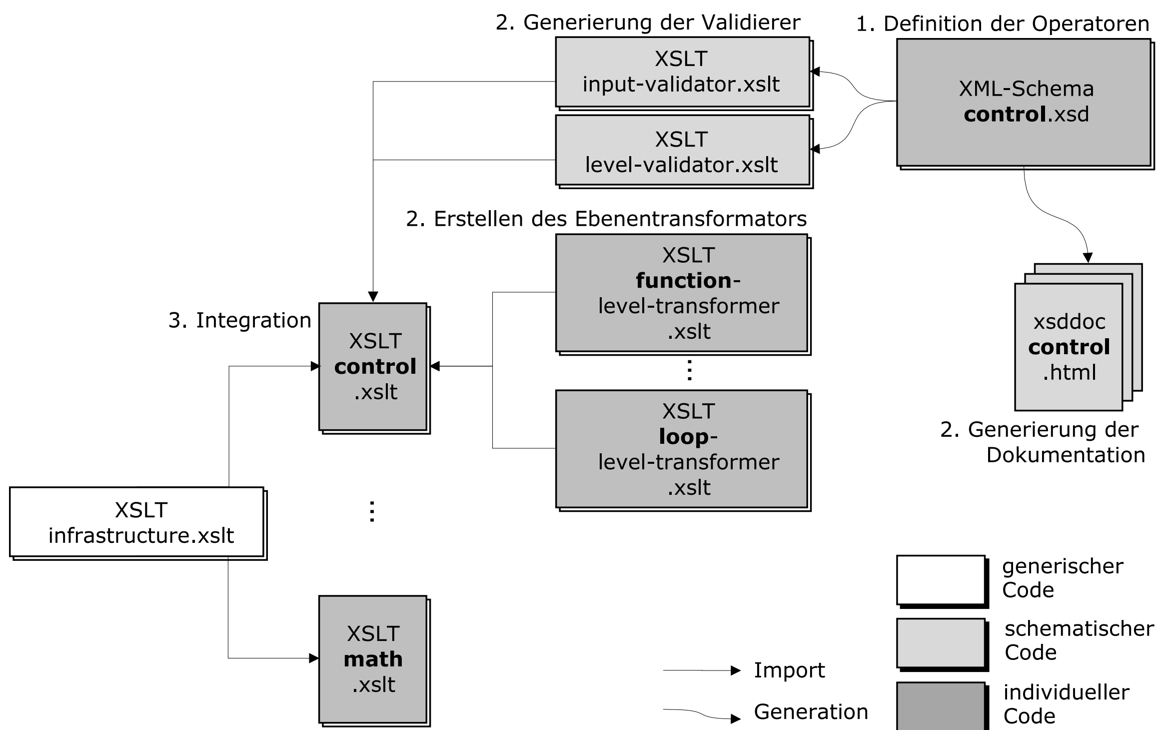


ABBILDUNG 8.2: Übersicht über Ergebnisse des Erstellungsprozesses.

sollen Operatoren hinzugefügt werden, welche den neuen Funktionsmechanismus implementieren. Wie im Bottom-up-Vorgehensmodell vorgeschlagen, folgt nach dem Identifizieren potentieller Operatoren die exakte Definition der neuen Operatoren. Hierbei entsteht ein Schema, aus dem im nächsten Schritt die Validierungskomponenten indirekt und die Dokumentation direkt generiert werden können. Parallel dazu kann der Ebenentransformator erstellt werden. Da die Ebenentransformation mit Hilfe der Sprache XSLT umgesetzt wird, werden konforme Transformationsdefinitionen erstellt bzw. generiert, die im letzten Schritt zusammen mit der Infrastrukturkomponente lediglich importiert werden müssen.

Wird die `control`-Sprachkomponente durch einen weiteren Operator (z. B. Schleifenoperator) ergänzt, so muss zunächst das Schema vervollständigt und die Validierungskomponenten bzw. Dokumentation erneut generiert werden. Werden separate Ebenentransformatoren geschrieben, müssen diese noch integriert werden. Alternativ können bestehende Ebenentransformatoren ebenso erweitert werden.

Die einzelnen Aktivitäten und ihre Ergebnisse für die Entwicklung des Funktionsmechanismus werden nun im Einzelnen genauer beschrieben.

Definieren des `function`- und `call-function`-Operators

Eine Funktion wird mit dem gleichlautenden `function`-Operator definiert. Eine Funktion hat einen qualifizierten Namen, der mit dem `name`-Attribut festgehalten wird. Der Name ist zwin-

gend notwendig. Eine Funktion kann endlich viele formale Parameter besitzen, die jeweils mit dem `param`-Element spezifiziert werden. Jeder Parameter muss dabei mit einen Namen ausgezeichnet werden. Eine optionale Standardbelegung, wie bei den Parametern der benannten Templates, ist nicht erforderlich, da der formale Parameter beim Funktionsaufruf immer mit tatsächlichen Werten belegt wird. Nach der Parameterdefinition folgt der problemspezifische Funktionsrumpf, der die eigentliche Verarbeitungsvorschrift beinhaltet.

```
<ctrl:function name=qname>
  <ctrl:param name=qname/>*
  <!-- Inhalt -->
</ctrl:function>
```

Der Gegenpart der Funktionsdefinition – der Funktionsaufruf – wird mit dem `call-function`-Operator beschrieben. Dafür wird der entsprechende qualifizierte Name benötigt.[2] Die Argumente des Funktionsaufrufs (`with-param`) werden mit einem Namen im `name`-Attribut und mit der Wertebindung (entweder durch `select`-Attribut oder Argumentinhalt) angegeben.

```
<ctrl:call-function name=qname | fctref>
  <ctrl:with-param name=qname select=expression?>
    <!-- Inhalt -->?
  </ctrl:with-param>*
</ctrl:call-function>
```

Im Kontrast zu den benannten Templates besteht jetzt die Signatur einer Funktion aus dem Namen und den zwingend erforderlichen Parametern. Insofern können Funktionen mit gleichen Namen und unterschiedlichen Parametern definiert werden. Da Parameter benannt sind, können sogar gleichnamige Funktionen mit gleicher Parameteranzahl eine unterschiedliche Signatur aufweisen.

Abschließend wird die obige EBNF-angelehnte Pseudonotation in eine Schemadefinition in XML Schema ausformuliert bzw. zu dem existierenden Schema der `control`-Sprachkomponente hinzugefügt. Code 8.1 zeigt einen Ausschnitt dieses Schemas. Die vollständige Definition der `call-function`- und `function`-Operatoren ist im Anhang A.1 zu finden. Der Ausschnitt beschränkt sich auf das `with-param`-Element, mit dem man Argumente bei einem Funktionsaufruf übergeben kann. Die oben beschriebene Syntax wird dabei innerhalb des `complexType`-Elementes festgelegt (Zeilen 8–15). Sie wird in diesem Schema durch eine Erweiterung des zuvor definierten `anyType`-Typs beschrieben. Der `anyType`-Typ lässt die Benutzung von Attributen außerhalb des Namensraums der `control`-Sprachkomponente zu. Wie im Codeausschnitt zu sehen, wird die Semantik im `annotation`-Element notiert. Innerhalb des `documentation`-Elementes erfolgt die textuelle Beschreibung. Diese ist bspw. für die spätere Generierung der Dokumentation wichtig. Innerhalb des `appinfo`-Elementes werden weitere Einschränkungen, die nicht mit XML Schema möglich sind, mittels Schematron-Regeln vorgenommen. Mit der Annahme (Zeile 5) wird gesichert, dass das `with-param`-Element entweder ein `select`-Attribut oder einen Inhalt besitzt.

[2] Um Funktionen höherer Ordnung zu unterstützen, kann der Name des `call-function`-Operators auch eine Referenz zu einer Funktion sein.

```
1  <element name="with−param">
2    <annotation>
3      <documentation>Parameter are passed to functions using the with−param element. The required name
         attribute specifies the name of the parameter. The name of parameter is specified in the same way as
         the corresponding parameter name in the function definition. The value of the passed parameter is
         defined by either a select attribute or the content of the element.</documentation>
4      <appinfo>
5        <sch:assert xmlns:sch="http://www.ascc.net/xml/schematron" test="not(@select and ∗)">Element '
           with−param' may have either a 'select' attribute or a content.</sch:assert>
6      </appinfo>
7    </annotation>
8    <complexType>
9      <complexContent mixed="true">
10       <extension base="ctrl:anyType">
11         <attribute name="name" type="string" use="required"/>
12         <attribute name="select" type="string" use="optional"/>
13       </extension>
14     </complexContent>
15   </complexType>
16 </element>
```

Code 8.1: Ausschnitt aus dem Schema (`control.xsd`) – das `with-param`-Element des `call-function`-Operators.

Erstellen des Ebenentransformators

In der Ebenentransformation werden die neu definierten Operatoren in die ursprünglichen Operatoren überführt. Um die Ebenentransformation der `function`- und `call-function`-Operatoren leichter verstehen zu können, wird der gesamte erforderliche Code vorgestellt (siehe Code 8.2). Zunächst werden die notwendigen Namensräume (Zeilen 2–4) und deren Eigenschaften, wie das Löschen des Namensraums im Zieldokument (Zeile 5) oder das Setzen eines Alias-Namensraums zur Vermeidung von Namensraumkonflikten (Zeile 7), festgelegt. Danach (Zeilen 9–14) wird das benannte Template `perform-transformation` mit dem Parameter `mode` definiert. Es bildet die Schnittstelle zum Inputvalidierer. Der Inputvalidierer erzeugt zunächst Code zur Überprüfung des Inputs und ruft anschließend dieses Template auf. Es kann sich somit hier auf die reine Überführung konzentriert werden, zumal die syntaktische Korrektheit des `function`- und `call-function`-Operators durch den generierten Ebenenvalidierer bereits als gesichert angenommen werden kann. In dieser Implementation des Ebenentransformators werden im Inneren des `perform-transformation`-Templates externe Templates im Modus `perform-transformation` angestoßen (Zeilen 11–13).

```
1  <xsl:transform version="1.0"
2    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
3    xmlns:axsl="http://www.w3.org/1999/XSL/Transform/Alias"
4    xmlns:ctrl="http://www.informatik.uni−kiel.de/Control"
5    exclude−result−prefixes="ctrl">
6
7  <xsl:namespace−alias stylesheet−prefix="axsl" result−prefix="xsl"/>
8
9  <xsl:template name="perform−transformation">
10   <xsl:param name="mode"/>
```

```
  <xsl:apply-templates select="." mode="perform-transformation">              11
    <xsl:with-param name="mode" select="$mode"/>                               12
  </xsl:apply-templates>                                                       13
</xsl:template>                                                                14
                                                                               15
<xsl:variable name="id" select="generate-id()"/>                               16
                                                                               17
<xsl:template match="ctrl:function" mode="perform-transformation">             18
  <xsl:param name="mode"/>                                                     19
  <xsl:variable name="template-name">                                          20
    <xsl:value-of select="@name"/>                                             21
    <xsl:for-each select="ctrl:param">                                         22
      <xsl:sort select="@name"/>                                               23
      <xsl:text>-</xsl:text>                                                   24
      <xsl:value-of select="@name"/>                                           25
    </xsl:for-each>                                                            26
    <xsl:text>-</xsl:text>                                                     27
    <xsl:value-of select="$id"/>                                               28
  </xsl:variable>                                                              29
  <axsl:template name="{$template-name}">                                      30
    <xsl:for-each select="ctrl:param">                                         31
      <axsl:param name="{@name}"/>                                             32
    </xsl:for-each>                                                            33
    <xsl:call-template name="apply-templates">                                 34
      <xsl:with-param name="mode" select="$mode"/>                             35
    </xsl:call-template>                                                       36
  </axsl:template>                                                             37
</xsl:template>                                                                38
                                                                               39
<xsl:template match="ctrl:call-function" mode="perform-transformation">        40
  <xsl:param name="mode"/>                                                     41
  <xsl:variable name="call-template-name">                                     42
    <xsl:value-of select="@name"/>                                             43
    <xsl:for-each select="ctrl:with-param">                                    44
      <xsl:sort select="@name"/>                                               45
      <xsl:text>-</xsl:text>                                                   46
      <xsl:value-of select="@name"/>                                           47
    </xsl:for-each>                                                            48
    <xsl:text>-</xsl:text>                                                     49
    <xsl:value-of select="$id"/>                                               50
  </xsl:variable>                                                              51
  <axsl:call-template name="{$call-template-name}">                            52
    <xsl:for-each select="ctrl:with-param">                                    53
      <xsl:choose>                                                             54
        <xsl:when test="@select">                                              55
          <axsl:with-param name="{@name}" select="{@select}"/>                 56
        </xsl:when>                                                            57
        <xsl:otherwise>                                                        58
          <axsl:with-param name="{@name}">                                     59
            <xsl:call-template name="apply-templates">                         60
              <xsl:with-param name="mode" select="$mode"/>                     61
            </xsl:call-template>                                               62
          </axsl:with-param>                                                   63
        </xsl:otherwise>                                                       64
      </xsl:choose>                                                            65
    </xsl:for-each>                                                            66
  </axsl:call-template>                                                        67
</xsl:template>                                                                68
                                                                               69
</xsl:transform>                                                               70
```

CODE 8.2: Ebenentransformationsdefinition des `function`- und `call-function`-Operators.

Alternativ können auch mehrseitige Auswahlbedingungen eingesetzt werden. Die in Code 8.2 gewählte Implementation hat jedoch den Vorteil, dass sie leichter erweitert werden kann. Es muss für einen neuen Operator lediglich ein neues entsprechendes Template hinzugefügt werden.

Wie ein solches Template für den `function`-Operator aussehen kann, zeigen die Zeilen 18–38 im Code 8.2. Als Vorbereitung wird ein eindeutiger Name des Templates generiert und an die Variable `template-name` gebunden (Zeilen 20–29). Der Name setzt sich aus dem Funktionsnamen konkateniert mit dem Namen der Parameter und einer eindeutigen Kennzahl als Suffix zusammen. Mit Hilfe dieser Variable und mit Informationen über die Parameter des `function`-Operators wird anschließend ein benanntes Template mit entsprechenden Parametern erzeugt (Zeilen 30–37). Um den Inhalt der Funktionsdefinition zu verarbeiten, wird das Template `apply-templates` aufgerufen. Es wird im Inputvalidierer definiert und koordiniert das weitere Durchlaufen (Zeilen 34–36). Die Übergabe des Modus ist wichtig, um den Einsprungspunkt wiederzufinden.

In analoger Weise wird die Überführung des `call-function`-Operators erstellt (Zeilen 39–67). Die Anwendung gleicher Namenskonventionen sichert hierbei, dass die Beziehung zwischen Funktionsdefinition und Funktionsaufruf erhalten bleibt.

Generieren des Ebenen- und Inputvalidierers

Mit Hilfe des Generatorsystems XOpGen können in einem mehrstufigen Generationsprozess aus dem Schema die Validierungskomponenten automatisch produziert werden. Im ersten Schritt wird das grammatikbasierte Schema in regelbasierte Schemata überführt. Zusätzliche Annotationen, wie die Annahme in Zeile 5 des Codeausschnittes 8.1, werden dabei entsprechend verwebt. Im zweiten Schritt werden daraus die Validierungskomponenten erzeugt. Eine Konfiguration des Generators ist nicht notwendig, da in der Standardeinstellung, wie benötigt, XSLT-Code generiert wird.

Code 8.3 zeigt den Aufbau des generierten Ebenenvalidierers. Zu Beginn werden wieder die benötigten Namensräume und deren Eigenschaften festgelegt (Zeilen 2–5). Danach wird in den Zeilen 7–9 der Aufruf der Infrastrukturkomponente entgegengenommen und eigene Templates im entsprechenden Modus angestoßen. Anschließend wird beispielhaft vorgeführt (Zeilen 11–47), wie die Überprüfung des `with-param`-Elementes vom `call-function`-Operator mit XSLT implementiert werden kann. Die Zeilen 13–19 zeigen den Codeteil, der aus der Annotation (Zeile 5 des Codeausschnittes 8.1) generiert wurde. Er garantiert, dass entweder ein `select`-Attribut oder ein Inhalt im `with-param`-Element verwendet wird. Mit den Zeilen 20–26 wird gesichert, dass das zwingend erforderliche `name`-Attribut vorhanden ist. Der Test in den Zeilen 27–37 und 38–47 schließt unerlaubte Attribute bzw. Kindelemente aus.

Neben der Validierungskomponente für die Ebenentransformation wird ebenso die Validierungskomponente für die elementare Transformation, die den Input prüft, generiert. Der Inputvalidierer enthält für das `with-param`-Element lediglich Code, der XSLT-Kontrollblöcke erzeugt, die validieren, ob der Wert des `name`-Attributes vom Typ `QName` ist.

```
<xsl:stylesheet version="1.0"                                                    1
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"                                2
  xmlns:sch="http://www.ascc.net/xml/schematron"                                  3
  xmlns:ctrl="http://www.informatik.uni-kiel.de/Control"                          4
  exclude-result-prefixes="ctrl">                                                 5
                                                                                  6
<xsl:template match="/" mode="validation">                                        7
  <xsl:apply-templates select="/" mode="levelValidation1"/>                       8
</xsl:template>                                                                   9
                                                                                  10
<xsl:template match="ctrl:with-param" priority="3996" mode="levelValidation1">    11
  <xsl:variable name="name-of-context" select="'with-param'"/>                    12
  <xsl:choose>                                                                    13
    <xsl:when test="not(@select and *)"/>                                         14
    <xsl:otherwise>                                                               15
      <xsl:message>Error: Element 'with-param' may have either a 'select' attribute or a content. Test:   16
        not(@select and *)</xsl:message>
      <xsl:text>.</xsl:text>                                                      17
    </xsl:otherwise>                                                              18
  </xsl:choose>                                                                   19
  <xsl:choose>                                                                    20
    <xsl:when test="@name"/>                                                      21
    <xsl:otherwise>                                                               22
      <xsl:message>Error: The operator '<xsl:value-of select="local-name(.)"/>' must have a 'name'   23
        attribute. Test: @name</xsl:message>
      <xsl:text>.</xsl:text>                                                      24
    </xsl:otherwise>                                                              25
  </xsl:choose>                                                                   26
  <xsl:for-each select="@*">                                                      27
    <xsl:choose>                                                                  28
      <xsl:when test="not(namespace-uri()='http://www.informatik.uni-kiel.de/Control' or   29
        namespace-uri()='') or local-name()='name' or local-name()='select' or false()"/>   30
      <xsl:otherwise>                                                             31
        <xsl:message>Error: The attribute '<xsl:value-of select="local-name(.)"/>' is not allowed on the   32
          operator '<xsl:value-of select="local-name(..)"/>'. Test: not(namespace-uri()='http://www.
          informatik.uni-kiel.de/Control' or namespace-uri()='') or local-name()='name' or local-name()
          ='select' or false()
        </xsl:message>                                                            33
        <xsl:text>.</xsl:text>                                                    34
      </xsl:otherwise>                                                            35
    </xsl:choose>                                                                 36
  </xsl:for-each>                                                                 37
  <xsl:for-each                                                                   38
    select="descendant::ctrl:*[local-name(ancestor::ctrl:*[1])=$name-of-context]">   39
    <xsl:choose>                                                                  40
       <xsl:when test="false()"/>                                                 41
       <xsl:otherwise>                                                            42
         <xsl:message>Error: The element '<xsl:value-of select="local-name(.)"/>' is not allowed on the   43
           operator '<xsl:value-of select="$name-of-context"/>'.Test: false() </xsl:message>
         <xsl:text>.</xsl:text>                                                   44
       </xsl:otherwise>                                                           45
    </xsl:choose>                                                                 46
  </xsl:for-each>                                                                 47
  ...                                                                             48
  <xsl:apply-templates mode="levelValidation1"/>                                  49
</xsl:template>                                                                   50
...                                                                               51
</xsl:stylesheet>                                                                 52
```

CODE 8.3: Ausschnitt aus dem Ebenenvalidierer (`level-validator.xslt`) – das `with-param`-Element des `call-function`-Operators.

Generieren der Dokumentation

Aus dem erstellten Schema der `control`-Sprachkomponente kann eine Dokumentation automatisch erzeugt werden. Abbildung 8.3 illustriert eine JavaDoc-ähnliche Darstellung des `with-param`-Elementes vom `call-function`-Operator. Sie wurde mit Hilfe des frei erhältlichen Werkzeugs xsddoc [Kur05] generiert und entspricht dem Codeausschnitt 8.1. Die generierten HTML-Dateien können mit einem Web-Browser angesehen werden.

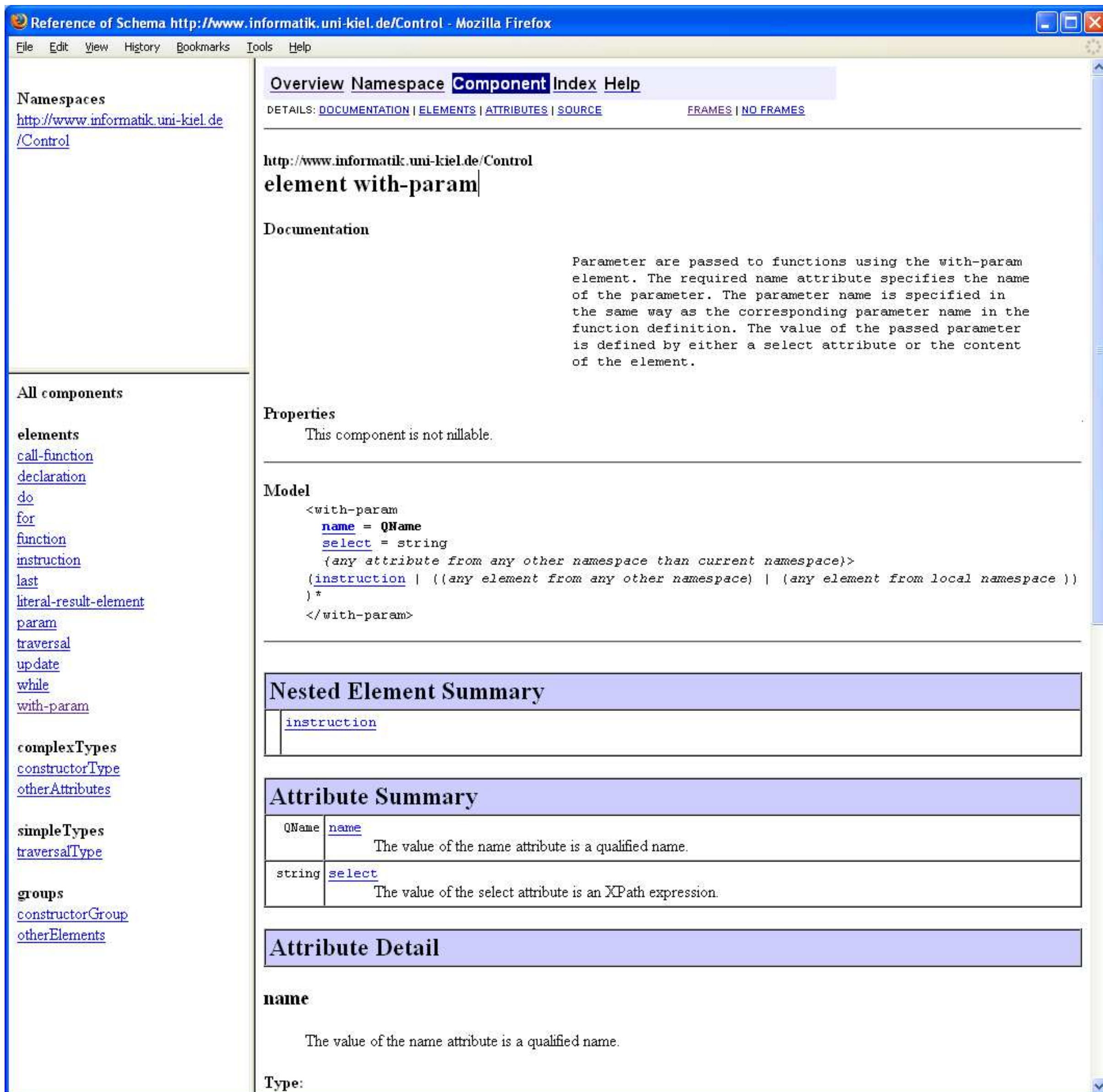


ABBILDUNG 8.3: JavaDoc-ähnliche Dokumentation der `control`-Sprachkomponente.

Andere Darstellungen und Zielformate sind möglich bzw. werden ebenso unterstützt. Unabhängig davon erlaubt eine wie in Abbildung 8.3 dargestellte Dokumentation erst das Verstehen der erstellten Operatoren und ist daher die Grundvoraussetzung für deren Anwendung.

Integration

Um die `control`-Sprachkomponente nutzen zu können, müssen die manuell geschriebene Ebenentransformationskomponente, die generische Infrastrukturkomponente und die generierten Validierungskomponenten miteinander verknüpft werden. Da bei der Generierung die Standardeinstellung verwendet wurde und der manuelle Teil die Standardschnittstellen unterstützt, kann die Integration durch einfaches Importieren vorgenommen werden (vgl. mit Code 6.6). Diese Datei (z. B. `control.xslt` in Abbildung 8.2) ist der Einstiegspunkt zur Nutzung der `control`-Sprachkomponente.

Installation, Test und Nutzung

Für eine Nutzung der Erweiterungssprachkomponente mit XTC muss die Hauptdatei (z. B. `control.xslt`) registriert werden. Dies wird durch eine Eintragung in eine externe Registry-Datei vollzogen (siehe Beispielcode 8.4).

```
<?xml version="1.0" encoding="UTF-8" ?>                                                              1
<level-transformation-definition-library>                                                             2
  ...                                                                                                  3
  <level-transformation-definition component="control" stand-alone="no">                              4
    <namespace>http://informatik.uni-kiel.de/Control</namespace>                                      5
    <location>level-library/XSLT/control.xslt</location>                                              6
    <used-language version="1.0>http://www.w3.org/1999/XSL/Transform</used-language>                 7
    <generated-language version="1.0">http://www.w3.org/1999/XSL/Transform</generated-language>      8
     ...                                                                                               9
  </level-transformation-definition>                                                                  10
   ...                                                                                                 11
</level-transformation-definition-library>                                                            12
```

Code 8.4: Beispielregistrierung der `control`-Sprachkomponente in XTC.

Durch die Eintragung werden XTC Metainformationen bspw. über den Namensraum der Sprachkomponente und deren Operatoren (Zeile 5) und der Ort der entsprechenden Hauptdatei (Zeile 6) mitgeteilt. Darüber hinaus werden Metainformationen über die benötigte Transformationssprache für die Ebenentransformation (Zeile 7) sowie über die erzeugten Transformationssprachen (Zeile 8) übermittelt. Hieraus kann XTC die grundsätzliche Durchführbarkeit errechnen und den Ebenentransformationsprozess koordinieren.

An dieser Stelle sollte die neue Sprachkomponente funktionsbereit sein. Es sollte aber unbedingt getestet werden, ob die neuen höheren Operatoren verarbeitet werden oder einfach, wie beim Template-basierten Ansatz üblich, in die Zieldatei kopiert werden. Dies kann auf eine fehlerhafte oder unvollständige Registrierung der Sprachkomponente bzw. deren Hauptdatei in XTC zurückzuführen sein (z. B. falsche Angabe des Namensraums). Ebenso kann ein inkorrektes Importieren von einzelnen Komponenten in die Hauptdatei der Sprachkomponente (`control.xslt`) das unerwünschte Verhalten verursachen. Nach dem generellen Funktionstest können die Operatoren zunächst alleine (Operatortest) und anschließend zusammen getestet werden (Integrationstest). Beim Operatortest sollte mit dem Prüfen, ob noch falsche Eingaben für den Operator möglich sind, begonnen werden. Zudem muss geprüft werden, ob der Operator nur an den gewünschten

Stellen anwendbar ist. Ein `function`-Operator kann bspw. nur als Kind des Wurzelelementes (Top-Level-Element) eingesetzt werden. Der `call-function`-Operator darf hingegen nur lokal und nicht als Top-Level-Element auftreten. In diesen Fällen, muss das Schema vervollständigt werden und der Generationsprozess für den Ebenenvalidierer sowie für die anderen Generate erneut angestoßen werden. Erst wenn alle diese Tests erfolgreich absolviert sind, sollte die eigentliche Ebenentransformationsdefintion validiert werden. Hier sind zwei Dinge zu prüfen. Wurde eine syntaktisch richtige Implementierung erzeugt und ist diese darüber hinaus semantisch korrekt? Das erste sichert, dass das Ergebnis der Ebenentransformation überhaupt verarbeitbar ist. Das zweite soll sichern, dass die Implementierung auch das macht, wofür sie vorgesehen wurde.

Das folgende minimale Beispiel zeigt, wie die neuen Operatoren genutzt werden können. Angenommen es sei eine Liste von Bestellungen als Quelle gegeben. Sämtliche Zeittypen sollen dabei in ein anderes Format konvertiert werden.

```
Quelle
<OrderArray xmlns="http://developer.ebay.com/webservices/latest/ebaySvc.xsd">
  <Order>
    <OrderID>2476679</OrderID>
    <OrderStatus>Completed</OrderStatus>
    <AdjustmentAmount currencyID="USD">0.0</AdjustmentAmount>
    <AmountSaved currencyID="USD">0.99</AmountSaved>
    <CheckoutStatus>
      <eBayPaymentStatus>PayPalPaymentInProcess</eBayPaymentStatus>
      <LastModifiedTime>20070515T08:11:51.000Z</LastModifiedTime>
      <PaymentMethod>PayPal</PaymentMethod>
      <Status>Complete</Status>
    </CheckoutStatus>
     ...
  </Order>
  <Order>
    <OrderID>2698789</OrderID>
    <OrderStatus>Active</OrderStatus>
    <AdjustmentAmount currencyID="USD">0.0</AdjustmentAmount>
    <AmountSaved currencyID="USD">0.0</AmountSaved>
    <CheckoutStatus>
      <eBayPaymentStatus>NoPaymentFailure</eBayPaymentStatus>
      <LastModifiedTime>20070520T18:31:55.000Z</LastModifiedTime>
      <PaymentMethod>None</PaymentMethod>
      <Status>Incomplete</Status>
    </CheckoutStatus>
     ...
  </Order>
</OrderArray>

Ziel
<order xmlns="http://clientA.de>
    ...
   <lastChange>2007−05−15</lastChange>
    ...
   <lastChange>2007−05−20</lastChange>
    ...
</order>
```

Code 8.5: Beispieltransformationsaufgabe.

Dies könnte bspw. notwendig sein, um zwei Systeme zu verknüpfen (z. B. der eBay-Marktplatz [eba08] mit einem Warenwirtschaftssystem). Mögliche Beispieldaten für die Quelle und das

Ziel sind im Code 8.5 angegeben. Die Integration der vorhandenen Systeme setzt einen korrekten Datenaustausch voraus. Die Transformation in das gewünschte Datenformat soll mit dem erweiterten XSLT vollzogen werden. Wie zuvor enthält dann eine Transformationsdefinition normale elementare XSLT-Operatoren gemischt mit neuen Operatoren, wie bspw. den `function`-Operator, der die Konvertierungsfunktion implementiert (Zeilen 7–13 in Code 8.6). Die definierte Konvertierungsfunktion kann mit dem entsprechenden `call-function`-Operator aufgerufen werden (Zeilen 17–19).

```
<xsl:transform                                                                      1
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"                                 2
   xmlns:ctrl="http://www.informatik.uni-kiel.de/Control"                           3
   xmlns:in="http://developer.ebay.com/webservices/latest/ebaySvc.xsd"              4
   xmlns:out="http://clientA.de">                                                   5
  ...                                                                               6
  <ctrl:function name="convertTime">                                                7
    <ctrl:param name="oldTimestamp"/>                                               8
    <xsl:value-of                                                                   9
       select ="concat(substring($oldTimestamp,'1','4'),'-',                        10
                       substring($oldTimestamp,'5','2'),'-',                        11
                       substring($oldTimestamp,'7','2'))"/>                         12
  </ctrl:function>                                                                  13
  ...                                                                               14
  <xsl:template match="//in:Order/in:CheckoutStatus">                               15
    <out:lastChange>                                                                16
      <ctrl: call-function name="convertTime">                                      17
        <ctrl:with-param name="oldTimestamp" select="in:LastModifiedTime"/>         18
      </ctrl: call-function>                                                        19
    </out:lastChange>                                                               20
  </xsl:template>                                                                   21
</xsl:transform>                                                                    22
```

Code 8.6: Nutzung des `function`- und `call-function`-Operators.

Würde diese erweiterte Transformationsdefinition zusammen mit der zu transformierenden Quelldatei an XTC übergeben, würde zunächst die Ebenentransformation gestartet und daraufhin die eigentliche elementare Transformation durchgeführt. Voraussetzung ist, dass die erstellte Ebenentransformationsdefinition bei der *Level Library* registriert und ein den W3C-Standard unterstützender XSLT-Prozessor in XTC korrekt eingebunden wurde. Abbildung 8.4 zeigt den Ebenentransformationsprozess für den `function`- bzw. `call-function`-Operator aus der `control`-Sprachkomponente, wobei die Quelldokumente grau gekennzeichnet sind. Der Ebenentransformationsprozess besitzt im vorgestellten Beispiel nur eine Stufe, da `function`- und `call-function`-Operatoren direkt in XSLT-Code überführt werden können. Die Ebenentransformationsdatei `control.xslt` wird dabei von XTC automatisiert aufgrund der bestehenden Namensraumzugehörigkeit identifiziert und ausgeführt.

Gleichermaßen können höhere Operatoren auch auf Operatoren anderer Erweiterungssprachkomponenten basieren. Dementsprechend müssen adäquate Ebenentransformationsdefinitionen aufgerufen werden.

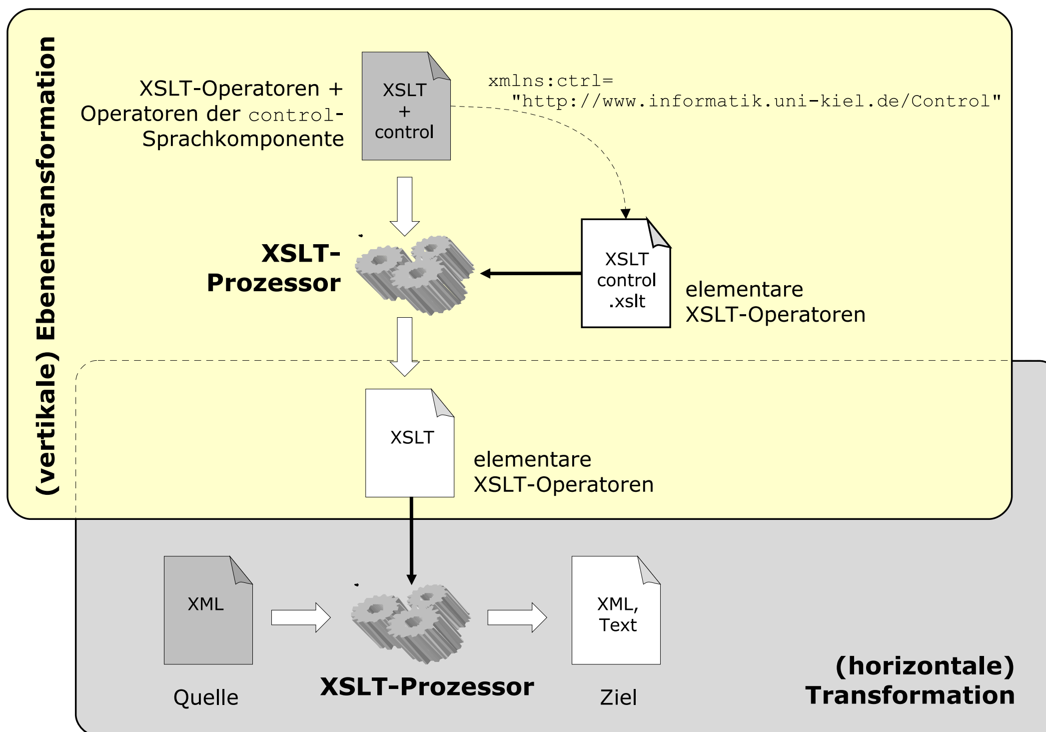


ABBILDUNG 8.4: Ebenentransformation für `control`-Sprachkomponente.

8.1.2 Domänenspezifische Operatoren

Technisch gesehen werden domänenspezifische Operatoren nach dem gleichen Prinzip gebildet wie universelle Operatoren. Es gibt jedoch zwei wesentliche Unterschiede. Zum einen werden domänenspezifische Operatoren fachlich für eine ganz bestimmte Domäne entworfen. Eine bestimmte Domäne zeichnet sich in aller Regel durch einen strikt abgegrenzten Transformationsbereich aus, in dem bspw. die Zielsprache klar festgelegt wurde. Ebenso sind womöglich Präferenzen und Abneigungen der Zielanwendergruppe zu beachten. Beides impliziert, dass diese Operatoren nicht mehr allgemein anwendbar sind. Zum anderen, um bspw. alle Sprachkonstrukte einer festgelegten Zielsprache erzeugen zu können, braucht es eine Menge an domänenspezifischen Operatoren, die enger miteinander in Beziehung stehen als die größtenteils lose gekoppelten universellen Operatoren. Beispielsweise können die Schleifenoperatoren (z. B. `while` oder `for`) selbst unabhängig, aber auch unabhängig vom im vorherigen Abschnitt definierten Funktionsmechanismus eingesetzt werden. Betrachtet man zielsprachenspezifische Operatoren, so müssen diese in richtiger Reihenfolge und Schachteltiefe angewendet werden, um korrekte Zieldokumente zu erzeugen. Resultierend aus diesen Unterschieden wird ein Top-down-Entwicklungsvorgehen, wie im Vorgehensmodell im Abschnitt 7.1 beschrieben, empfohlen.

Wie die Vielzahl der XML-Transformationssprachen, bietet XSLT bspw. keine Operatoren an, um eine bestimmte Klasse von Zieldokumenten zu erzeugen. XSLT stellt lediglich Operatoren für die Erzeugung von Textliteralen, XML-Literalen oder XML-Informationseinheiten (z. B.

Elemente, Attribute, Verarbeitungsanweisungen) bereit. Spezielle Zielsprachen, ob XML-basiert oder nicht-XML-basiert, werden nicht unterstützt. Mit Hilfe der Operatorhierarchie können für beide Kategorien von Zielsprachen entsprechende domänenspezifische Operatoren entworfen werden, die XSLT geeignet erweitern. Dies kann so weit getrieben werden, dass eine definierte Menge an domänenspezifischen Operatoren eine eigenständige Sprache bildet, die letztendlich nur noch auf XSLT basiert, aber keine XSLT-Operatoren als solches zulässt (siehe Abschnitt 5.2).

Beispiele von domänenspezifischen Operatoren oder genauer gesagt dessen Sprachkomponente werden anhand von konkreten Fallbeispielen aus Forschungsprojekten und Praxis im kommenden Abschnitt erläutert.

8.2 Fallbeispiele

Nachdem der Schwerpunkt bisher vorwiegend auf der technische Umsetzung lag, wird sich in diesem Abschnitt mehr der Analyse- und Entwurfsphase gewidmet. Anhand eines konkreten Anwendungsproblems wird eine problemangepasste Sprache entworfen. Am Ende des Abschnittes werden weitere Fallbeispiele kurz angeführt.

8.2.1 Datenaustausch am Beispiel von Truitions XML-Export – Die Transformationssprache XML2DSV

Bevor mit der eigentlichen Sprachentwicklung begonnen wird, werden vorab einige Hintergrundinformationen gegeben, um einen besseren Einblick in die Problemstellung und das Problemumfeld zu bekommen. Zunächst werden die Firma und ihre Ziele, Kunden und vorhandene Systemlandschaften kurz geschildert.

Einführung und Motivation

Truition ist eine Firma, die sich als ein E-Commerce-Anbieter versteht, der durch seine speziellen On-Demand-Software-Lösungen das effiziente Auslagern (*Outsourcing*) von wesentlichen Teilen der E-Commerce-Aktivitäten seiner Kunden unterstützt und ermöglicht [Tru08]. Die Produktfamilie wird als *Commerce Management System* (CMS) bezeichnet. Die Produkte werden in Form von Diensten betrieben und dem Kunden über öffentliche Netze angeboten. Truition tritt dabei als *Application Service Provider* (ASP) auf, der sich um die gesamte Administration, wie Backups, das Einspielen von Patches usw., kümmert. Mit CMS kann ein Kunde eine Vielzahl von Online-Vertriebskanälen mit einer E-Commerce-Plattform nutzen. So können mit Hilfe des CMS Kundenprodukte über einen eigenen Online-Shop, Marktplatz oder Auktionen angeboten werden. Weiterhin ist es möglich, die Produkte in etablierten Online-Marktplätzen wie bspw. eBay oder ElectronicScout24 einzustellen (siehe Abbildung 8.5).

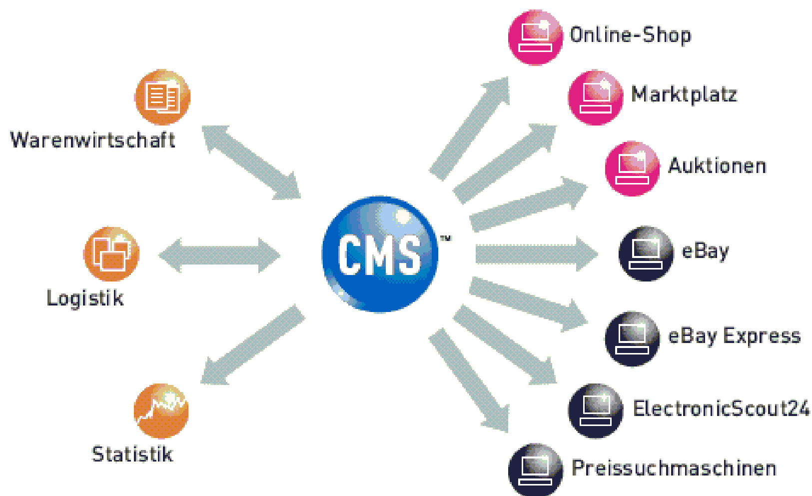


ABBILDUNG 8.5: Truitions Commerce Management System (CMS) [Tru08].

Um dies zu ermöglichen, werden notwendige Produktdaten (z. B. Name, Beschreibung, Preis, Verfügbarkeit, etc.), die durch Warenwirtschafts- oder Logistiksysteme den Kunden zur Verfügung gestellt werden, in eine interne CMS-Struktur überführt. Anschließend werden die Daten den Schnittstellen entsprechend an die Online-Vertriebskanäle übertragen. Wird innerhalb eines solchen Vertriebskanals eine Transaktion (z. B. eine Bestellung eines Produkts) ausgelöst, wird diese Information an die Kundensysteme zurückgeliefert und dort die entsprechende Verarbeitungsroutine gestartet. Die Routine wiederum stößt ihrerseits mglw. weitere Interaktionsprozesse mit dem Vertriebskanal über das CMS an (z. B. Bezahlprozesse).

Wie das Szenario skizziert, müssen mit verschiedenen Schnittstellen Daten ausgetauscht werden. Etablierte Online-Marktplätze definieren eigene spezielle APIs, über die das Erstellen bzw. Einstellen eines Shops und dessen Produkten sowie die gesamte Auftragsabwicklung mit Bestellbestätigung und Verkaufsabwicklung erfolgen. Obwohl die Schnittstellen dabei nicht fest sind, sondern sich wie bspw. beim eBay API [eba08] i. d. R. alle 24 Wochen ändern, kann mit dem CMS relativ flexibel darauf reagiert werden. Gründe für die häufige Anpassung liegen neben den üblichen Wartungs- und Weiterentwicklungsprozessen vor allem in der überdurchschnittlichen Dynamik im elektronischen Handel. Dieser Markt ist durch immer wieder neue Verkaufs- und Marketingstrategien gekennzeichnet (z. B. Cross- und Upselling), um sich von der zunehmenden Konkurrenz abzugrenzen und hervorzuheben [Amo02].

Problematischer als die Marktplatzseite stellen sich allerdings die Schnittstellen auf Kundenseite dar. Sie unterliegen zwar einem nicht ganz so starken Änderungsprozess, doch variieren die Schnittstellen von Kunde zu Kunde erheblich. Eine Untersuchung der Kunden von Truition in [FFSD06] ergab bspw., dass 46 Prozent (38% proprietär und 8% Web Service) einen direkten Zugriff auf die Kundensysteme anbieten. Die Mehrheit (54 Prozent) erlaubt keinen direkten Zugriff. Vielmehr werden gebündelte, komplette Datensätze entweder online oder offline übertragen. Dabei kommen unterschiedlichste Datenformate, wie z. B. ASCII Text, Excel, XML und DSV (*Delimiter-Separated Value*) zum Einsatz. Das Letztere ist das am meisten genutzte

Format von Truitions Kunden (53,7%).

Um flexibel auf Änderungen reagieren zu können, soll für die Überführung von XML in DSV-Datenformate eine eigene, spezifische Transformationssprache entworfen werden. DSV ist ein einfaches Datenaustauschformat, dass von vielen Altsystemen und nahezu allen Tabellenkalkulationswerkzeugen unterstützt wird. Das DSV-Format verwendet bestimmte Trennzeichen, um Datenfelder und Datensätze voneinander abzugrenzen. Das bekannteste und standardisierte Format unter ihnen ist CSV (*Comma-Separated Values*) [Sha05]. Es trennt die Datenfelder und Datensätze mit Kommas bzw. Zeilenumbrüchen und verwendet Anführungsstriche als Maskierungszeichen der Datenfelder.

Analyse

Zunächst muss definiert werden, für welchen Anwendungsbereich die Sprache geeignet sein soll. Um diese Frage systematisch anzugehen, könnte man bereits hier Merkmalmodellierung einsetzen, um festzuhalten, welche Teile der Transformationsdomäne adressiert werden sollen. Im Folgenden wird für die erste Phase des *Domain Scoping* ein pragmatischer Ansatz gewählt. Erst nachdem gezielt bestimmte Szenarien ausgeschlossen wurden, wird festgelegt, welche Merkmale die Transformationssprache besitzt und welche variabel sind. Diese Zweiteilung hilft, das aufwendige Vorgehen der systematischen Merkmalmodellierung zu minimieren und den Fokus auf das Wesentliche zu richten.

Die Sprache soll speziell für die Überführung von XML-Daten in CSV-ähnliche Repräsentationen der Daten entworfen werden. Die Zielanwendergruppe soll vorwiegend der technische Entwickler sein. Damit aber ebenso weniger versierte Entwickler (z. B. auf Kundenseite) diese Sprache nutzen können, werden folgende Entwicklungsziele verfolgt:

- Die Transformationssprache soll verständlich und einfach sein. Dies impliziert, dass die Anzahl der optionalen Features, die keinen signifikanten Mehrnutzen mit sich bringen, minimiert wird.

- Die Transformationssprache soll bzgl. der Struktur von DSV möglichst flexibel sein. Sie soll bspw. verschiedene Trennzeichen unterstützen. Dadurch kann sie trotz des eingegrenzten Anwendungsbereiches für eine Vielzahl von Problemen angewendet werden.

- Die Transformationssprache soll kompatibel mit XML sein. Die Kompaktheit der Operatoren soll somit einen nachrangigen Stellenwert haben.

- Stattdessen soll die Transformationssprache gut durch Anwendungen (wie Editoren, Parser usw.) unterstützt werden können. Es soll bspw. einfach sein, Programme zu schreiben, die die Transformationssprache sowohl generieren als auch weiterverarbeiten können.

- Der Entwurf der Transformationssprache soll formal und präzise sein. Dies beinhaltet eine nützliche Dokumentation.

Im Gegensatz zum Zielformat, dass auf das DSV-Format eingeschränkt ist, soll das Quellformat lediglich XML-basiert sein. Eine weitere Einschränkung, z. B. auf eine bestimmte Markup-Sprache, soll nicht erfolgen.

Innerhalb der Sprache sollen für die Sprachkonstrukte übliche Bezeichnungen, wie sie im Zielformat anzutreffen sind, verwendet werden. Gebräuchliche Bezeichnungen sind Trennzeichen (*delimiter*), Maskierung (*escape*), Daten (*data*), Datensätze (*record*) oder Datenfelder (*field*).

Nachdem weitestgehend die Transformationsdomäne eingegrenzt wurde, kann die Transformationssprache analysiert werden. Das Merkmaldiagramm in Abbildung 8.6 beschreibt die Eigenschaften der Sprache detaillierter. Zum einen soll sie erlauben, sämtliche Trenn- und Maskierungszeichen festlegen zu können. Werden keine Zeichen explizit konfiguriert, so gilt die Standardeinstellung für das entsprechende Zeichen. Die Standardeinstellung entspricht dem CSV-Format [Sha05].

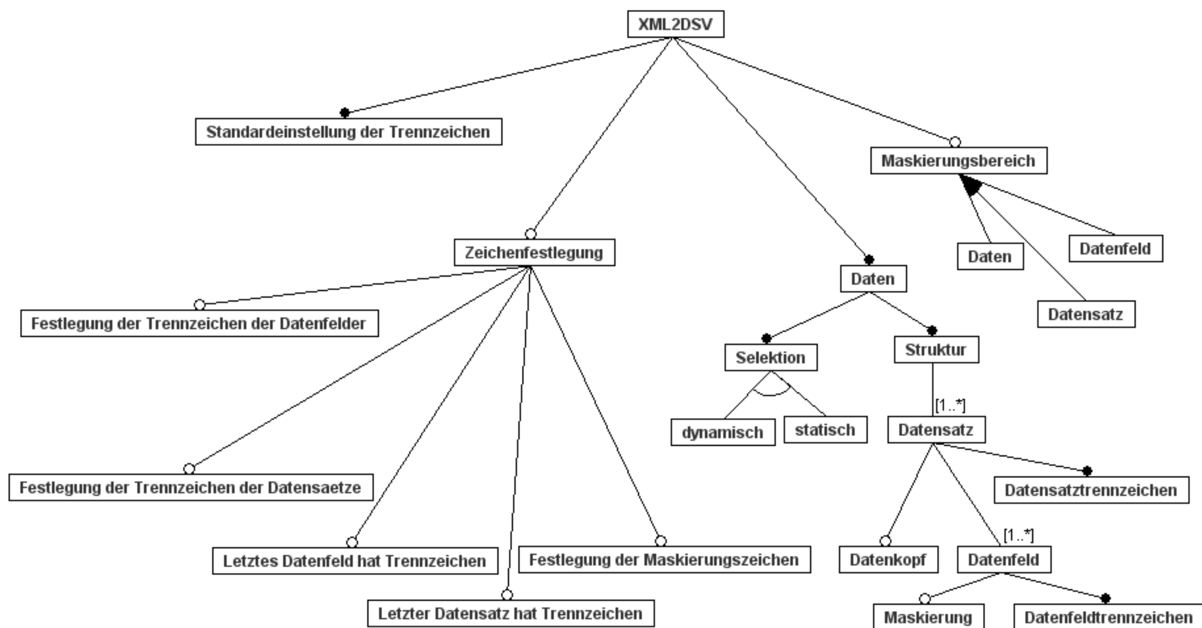


ABBILDUNG 8.6: Merkmale von XML2DSV.

Zum anderen werden die Daten in Datensätze strukturiert, die durch konfigurierte Trennzeichen getrennt werden. Jeder Datensatz besitzt wiederum durch Trennzeichen abgegrenzte Datenfelder. Datenfelder können maskiert werden. Dies ist bspw. notwendig, wenn in ihnen Trennzeichen enthalten sind. Ob Datenfelder maskiert werden oder nicht, kann global für alle Daten oder lokal für einen ganzen Datensatz oder für ein einziges Datenfeld angegeben werden. Lokalere Konfigurationen überdecken globalere Konfigurationen. Die Daten selbst können aus den XML-Quelldokumenten (dynamisch) selektiert werden oder sie sind (statische) Textliterale, die unabhängig von den Quellen sind.

Entwurf

Die Syntax der spezifischen Transformationssprache wird durch ein Schema in XML Schema definiert. Es wird mit textuell verfasster natürlicher Sprache ergänzt, die die Semantik der Operatoren beschreibt und bspw. bei der Dokumentationsgenerierung genutzt wird. Das vollständige Schema ist im Anhang A.2 zu finden.

Insbesondere in den frühen Phasen des Sprachfindens ist es sinnvoll, eine Referenztransformationsdefinition fortzuschreiben. Anhand dieser kann die Syntax und zusammen mit der Referenzimplementierung die Semantik bis auf Detailebene exemplarisch gezeigt werden. An dieser Stelle soll mit Hilfe der Referenztransformationsdefinition, welche in Code 8.7 abgebildet ist, die Syntax der domänenspezifischen Sprache erläutert werden.

Ein XML2DSV-Programm [FP07b] wird durch einen `dsv`-Wurzeloperator in einem XML-Dokument repräsentiert. Der `dsv`-Operator kann eine Trennzeichenbeschreibung und muss eine Beschreibung der Daten haben. Die optionale Trennzeichenbeschreibung, die durch den `separators`-Operator spezifiziert wird, ermöglicht die Trennzeichen zu definieren. Mit dem `field-separator`-Operator bspw. kann das Trennzeichen für die Datenfelder festgelegt werden und ob das letzte Datenfeld ein abschließendes Trennzeichen hat. Wird die Trennzeichenbeschreibung weggelassen, so gilt die Standardeinstellung, die in den Zeilen 3–5 im Code 8.7 konfiguriert wurde. Sie erzeugt ein CSV-konformes Zieldokument.

```
<dsv xmlns="http://www.informatik.uni-kiel.de/DSV"                                   1
      xmlns:in="http://developer.ebay.com/webservices/latest/ebaySvc.xsd" >          2
  <separators>                                                                       3
    <field-separator ending-field-break="yes" delimiter=","/>                        4
    <field-escape delimiter=""/>                                                     5
    <record-separator ending-record-break="no" delimiter="&#xA;"/>                   6
  </separators>                                                                      7
  <data escaped-field="yes">                                                         8
    <header escaped-field="no">                                                      9
      <field>ID</field>                                                              10
      <field>STATUS</field>                                                          11
      <field>AMOUNT</field>                                                          12
      <field>PAYMENT</field>                                                         13
      <field>CHECKOUT</field>                                                        14
    </header>                                                                        15
    <record match="//in:Order">                                                      16
      <field select="in:OrderID"/>                                                   17
      <field select="in:OrderStatus[text()='Active' or text()='Completed']"/>        18
      <field select="in:AmountSaved"/>                                               19
      <field select="in:CheckoutStatus/in:PaymentMethod"/>                           20
      <field>yes</field>                                                             21
    </record>                                                                        22
  </data>                                                                            23
</dsv>                                                                               24
```

Code 8.7: Anwendung der XML2DSV-Transformationssprache.

Der `data`-Operator wird eingesetzt, um Daten zu beschreiben. Die Selektion von Daten wird mittels des `record`-Operators zusammen mit den `field`-Operatoren spezifiziert. Das optionale `match`-Attribut innerhalb des `record`-Operators erlaubt es, mittels eines XPath-Ausdrucks

auf ein oder mehrere Quelldokumente zuzugreifen und darin bestimmte Knoten zu markieren (Zeile 16). Für jeden dieser markierten Kontextknoten wird im Zieldokument ein Datensatz angelegt. Abhängig von den Kontextknoten werden konkrete Dateneinträge durch das `select`-Attribut im `field`-Operator selektiert (Zeilen 17–20). Wenn die Datenfelder lediglich an statische Zeichenketten gebunden werden (Zeilen 10–14), darf das `match`-Attribut im `record`-Operator weggelassen werden. Es darf jedoch umgekehrt kein `select`-Attribut benutzt werden, wenn das `match`-Attribut fehlt. Ein besonderer Datensatz ist der Kopfdatensatz. Dieser wird durch den `header`-Operator, der nur einmal und nur als erster Datensatz auftreten darf, beschrieben. Außer diesen zwei Restriktionen entspricht der `header`-Operator dem `record`-Operator.

Sämtliche Operatoren im `data`-Operator und der `data`-Operator selbst unterstützen ein `escaped-field`-Attribut. Mit diesem wird festgelegt, ob die Datenfelder maskiert oder nicht maskiert werden. Die Standardeinstellung ist Letzteres. Eine Festlegung überdeckt eine andere Festlegung, wenn sie an einem Punkt auftritt, an dem die andere ebenso sichtbar ist.

Die Implementierung der XML2DSV-Transformationssprache wird in XSLT mit der `control`-Sprachkomponente umgesetzt. Grundsätzlich kann eine dynamische oder statische Realisierung gewählt werden. Bei der dynamischen Variante werden die Trenn- und Maskierungszeichen in jeweils eigenen Variablen abgelegt und an den notwendigen Stellen referenziert (siehe Code 8.8). In der statischen Variante werden dagegen die Zeichen gleich an Ort und Stelle als Textliterale eingefügt. Die letztere Realisierung ist kürzer, aber weniger flexibel.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <transform version="1.0"
3    xmlns="http://www.w3.org/1999/XSL/Transform"
4    xmlns:in="http://developer.ebay.com/webservices/latest/ebaySvc.xsd" >
5    xmlns:ctrl="http://www.informatik.uni-kiel.de/Control">
6
7    <output method="text"/>
8    <variable name="field-separator-delimiter">,</variable>
9    <variable name="ending-field-break">yes</variable>
10   <variable name="field-escape-delimiter">"</variable>
11   <variable name="record-separator-delimiter">
12                                                   </variable>
13   <variable name="ending-record-break">no</variable>
14
15   <template match="text()"/>
16
17   <template match="/">
18     <text>ID</text>
19     <value-of select="$field-separator-delimiter"/>
20     <text>SATUS</text>
21     <value-of select="$field-separator-delimiter"/>
22     <text>AMOUNT</text>
23     <value-of select="$field-separator-delimiter"/>
24     <text>PAYMENT</text>
25     <value-of select="$field-separator-delimiter"/>
26     <text>CHECKOUT</text>
27     <if test="$ending-field-break='yes'">
28       <value-of select="$field-separator-delimiter"/>
29     </if>
30     <value-of select="$record-separator-delimiter"/>
31     <for-each select="//in:Order">
32       <variable name="path" select="concat('//in:Order[',position() ,']/') "/>
33       <value-of select="$field-escape-delimiter"/>
```

```
      <ctrl: call −function name="check−escaped−field">                                    34
        <ctrl:with−param name="field" select="in:OrderID"/>                                35
        <ctrl:with−param name="selection" select="concat($path,'in:OrderID')"/>            36
      </ctrl: call −function>                                                              37
      <value−of select="$field−escape−delimiter"/>                                         38
      <value−of select="$field−separator−delimiter"/>                                      39
      <value−of select="$field−escape−delimiter"/>                                         40
      <ctrl: call −function name="check−escaped−field">                                    41
        <ctrl:with−param name="field" select="in:OrderStatus[text()='Active' or text()='Completed']"/>  42
        <ctrl:with−param name="selection" select="concat($path,'in:OrderStatus')"/>        43
      </ctrl: call −function>                                                              44
      <value−of select="$field−escape−delimiter"/>                                         45
      <value−of select="$field−separator−delimiter"/>                                      46
      <value−of select="$field−escape−delimiter"/>                                         47
      <ctrl: call −function name="check−escaped−field">                                    48
        <ctrl:with−param name="field" select="in:AmountSaved"/>                            49
        <ctrl:with−param name="selection" select="concat($path,'in:AmountSaved')"/>        50
      </ctrl: call −function>                                                              51
      <value−of select="$field−escape−delimiter"/>                                         52
      <value−of select="$field−separator−delimiter"/>                                      53
      <value−of select="$field−escape−delimiter"/>                                         54
      <ctrl: call −function name="check−escaped−field">                                    55
        <ctrl:with−param name="field" select="in:CheckoutStatus/in:PaymentMethod"/>        56
        <ctrl:with−param name="selection" select="concat($path,'in:CheckoutStatus/in:PaymentMethod')  57
          "/>
      </ctrl: call −function>                                                              58
      <value−of select="$field−escape−delimiter"/>                                         59
      <value−of select="$field−separator−delimiter"/>                                      60
            <value−of select="$field−escape−delimiter"/>                                   61
      <text>yes</text>                                                                     62
      <value−of select="$field−escape−delimiter"/>                                         63
      <if test="$ending−field−break='yes'">                                                64
        <value−of select="$field−separator−delimiter"/>                                    65
      </if>                                                                                66
      <choose>                                                                             67
        <when test="$ending−record−break='yes'">                                           68
          <value−of select="$record−separator−delimiter"/>                                 69
        </when>                                                                            70
        <otherwise>                                                                        71
          <if test="not(position()=last())">                                               72
            <value−of select="$record−separator−delimiter"/>                               73
          </if>                                                                            74
        </otherwise>                                                                       75
      </choose>                                                                            76
    </for−each>                                                                            77
  </template>                                                                              78
                                                                                           79
  <ctrl:function name="check−escaped−field">                                               80
    <ctrl:param name="field"/>                                                             81
    <ctrl:param name="selection"/>                                                         82
    <if test="contains( $field , $field −escape−delimiter)">                               83
      <message>Error in source: The selected node(<value−of select="$selection"/>) contains  84
       an escape delimiter.</message>                                                      85
    </if>                                                                                  86
     ...                                                                                   87
  </ctrl:function>                                                                         88
</transform>                                                                               89
```

Code 8.8: Ausschnitt der Implementation der Referenztransformationsdefinition in XSLT unter der Verwendung der `control`-Sprachkomponente.

Implementierung

Aus der Referenztransformationsdefinition und der Referenzimplementierung wird der Ebenentransformator abgeleitet. Die Umsetzung erfolgt nach dem gleichen systematischen Vorgehen wie die Ebenentransformation bei den universellen Operatoren (siehe Abschnitt 8.1.1).

Gleiches gilt für die Generierung der Validierungskomponenten und der Dokumentation sowie für die Integration. Deshalb wird auf die Beschreibung des jeweiligen Codes an dieser Stelle verzichtet.

Im Folgenden soll exemplarisch die Anpassung des integrierten XML-Editors im Eclipse WTP gezeigt werden. Das WTP-Plugin stellt vordefinierte Erweiterungspunkte bereit, mit denen u. a. auch neue Schemadefinitionen in XML Schema registriert werden können. Im Verzeichnis `.metadata\.plugins\org.eclipse.wst.xml.core` befindet sich die Konfigurationsdatei `user_catalog.xml` (siehe Code 8.9). In dieser Datei muss der gewählte Namensraum einem Schema zugeordnet werden, indem auf eine URI oder eine lokale Speicheradresse des Schemas verwiesen wird (Zeilen 8–9).

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <catalog xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog">
3    <uri name="http://xml.ascc.net/schematron"
4          uri="http://xml.ascc.net/schematron/schematron1-5.xsd"/>
5    <uri name="http://www.informatik.uni-kiel.de/Control"
6          uri="platform:/resource/XTC/level-library/XSLT/Control/control.xsd"/>
7     ...
8    <uri name="http://www.informatik.uni-kiel.de/DSV"
9          uri="platform:/resource/XTC/level-library/XSLT/DSV/dsv.xsd"/>
10 </catalog>
```

Code 8.9: Registrierung des Schemas von XML2DSV beim Editor.

Nachdem diese Registrierung durchgeführt wurde, zeigt der Editor sowohl Fehlermeldungen als auch Hinweise für den Entwickler an. Wie die Abbildung 8.7 verdeutlicht, werden dabei aber noch nicht alle Einschränkungen des Schemas unterstützt. Beispielsweise wurde im Schema von XML2DSV definiert, dass es nur einen `header`-Operator im `data`-Operator geben darf. Der Hinweis gibt trotz eines bereits vorhandenen `header`-Operators diesen als Option an.

Gerade deswegen ist eine gesonderte Validierung der XML2DSV-Programme vor der eigentlichen Ebenentransformation so essentiell. Neben der fehlenden Unterstützung einiger XML Schema-Elemente werden ebenso die zusätzlichen Einschränkungen in Schematron-Notation nicht beachtet bzw. unterstützt.

8.2.2 Weitere Fallbeispiele

Neben der Entwicklung einer spezifischen Transformationssprache zur Erzeugung von Zieldokumenten im DSV-Format, wurde eine Reihe weiterer domänenspezifischer Sprachkomponenten mit dem vorgeschlagenen Framework entwickelt. Folgende Liste zählt einige davon auf:

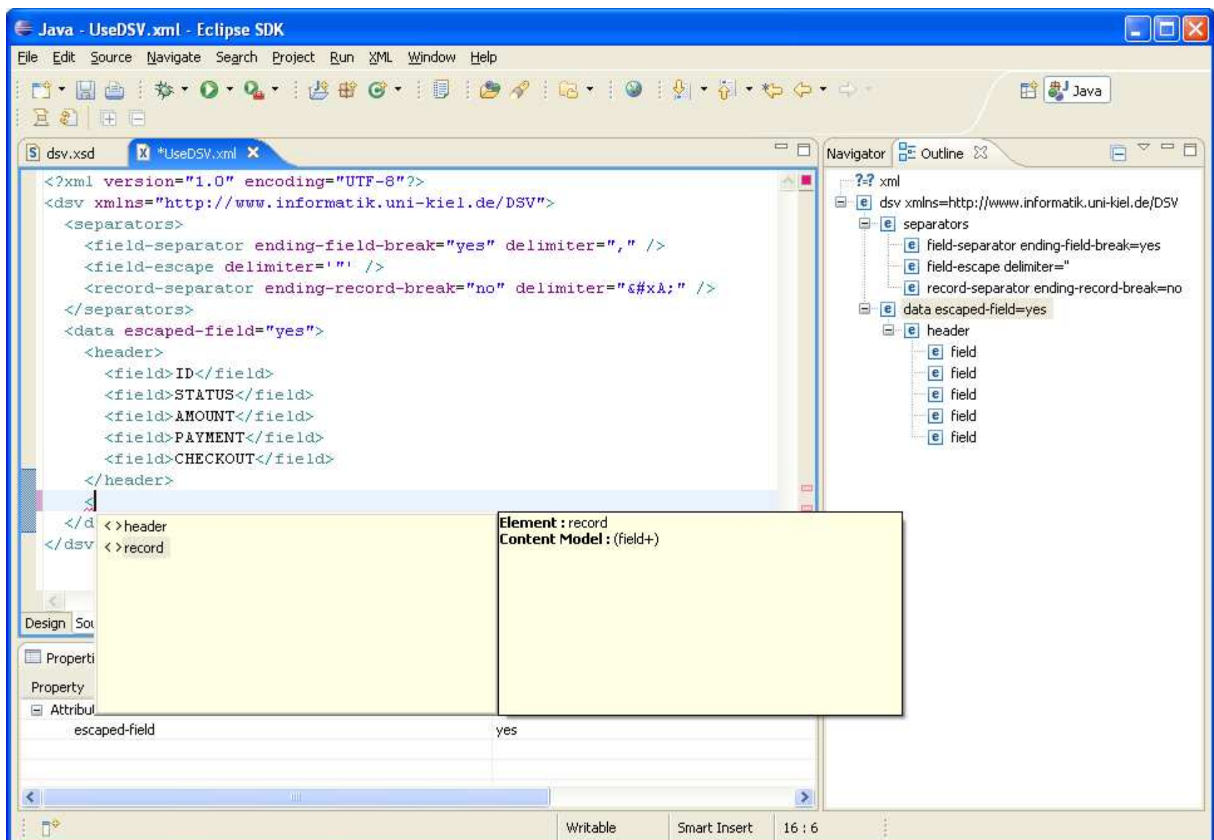


ABBILDUNG 8.7: Angepasster Editor für XML2DSV.

- XSLT-spezifische Sprachkomponente: Diese Sprachkomponente wurde entwickelt, um die Operatoren, die zur Erzeugung von XSLT genutzt werden, zu überprüfen und die syntaktischen Fehler bereits im Ursprung zu finden. Die Sprachkomponente wurde extensiv bei den Implementierungen der Ebenentransformationen erfolgreich genutzt.

- XHTML-spezifische und ISML-spezifische Sprachkomponenten: In Zusammenarbeit mit der Firma Intershop Communications AG wurden diese Sprachkomponenten entwickelt, um Daten mit grafischen Darstellungsinformationen anzureichern (ISML - Intershop Markup Language). In [FSH05] wird das grundlegende Vorgehen der Transformation beschrieben. Eine konkrete Transformationsaufgabe kann [FSRK05] entnommen werden.

- BPEL-spezifische Sprachkomponente: Im Rahmen des BMBF-Projektes IE (Integration Engineering) [IE206] wurde diese Sprachkomponente entwickelt, um eine leichtere Überführung von fachlichen EPKs (Ereignisgesteuerte Prozessketten) nach technischen BPEL-Prozessen (*Business Process Execution Language*) zu erzielen. Die dafür verwendete Transformationsarchitektur wird in [Föt05] vorgestellt. [Föt07b] beschreibt die Implementierungsaspekte und [FFS06, FF07] zwei Anwendungen dieser Überführung.

- SMV-spezifische Sprachkomponente: Im Rahmen des BMBF-Projektes OrViA (Orchestrierung und Validierung integrierter Anwendungssysteme) [OrV08] wurde diese Sprachkomponente entwickelt, um die Implementierung der Überführung von EPKs sowie anderer Prozesssprachen in ein Format des Modellprüfers SMV (*Symbolic Model Verifier*) zu vereinfachen. Dadurch können diese Prozessmodelle validiert werden. In [FPR06] wird hierfür eine grundlegende Transformationsarchitektur vorgestellt. In [FF08] wird u. a. auch die konkrete Transformation diskutiert.

- CTL-spezifische Sprachkomponente: Im Rahmen des BMBF-Projektes OrViA wurde diese Sprachkomponente entwickelt, um die G-CTL (*Graphical Computational Temporal Logic*) nach CTL (*Computational Temporal Logic*) zu überführen. Einen Überblick über CTL gibt bspw. [SPFF06]. G-CTL und die Transformation zu G-CTL werden in [FF08] und [FFS08] beschrieben.

8.3 Schlussbemerkungen

Die gezeigten Anwendungsbeispiele demonstrieren exemplarisch die Erweiterung von XSLT (Version 1.0 [Cla99a]) durch zusätzliche Sprachkomponenten. Um das Potential am Beispiel XSLT herauszustreichen, wird im ersten Teilabschnitt 8.3.1 die ursprüngliche Transformationssprache mit der erweiterten verglichen. Im zweiten Teilabschnitt 8.3.2 werden die Möglichkeiten, die mit unseren vorgeschlagenen Frameworks basierend auf XSLT 1.0 realisiert werden können, den Neuerungen von XSLT 2.0 gegenübergestellt.

8.3.1 Vergleich mit XSLT 1.0

Wird XSLT mit der Operatorhierarchie erweitert, können sämtliche ursprüngliche Basisoperatoren weiter verwendet werden; müssen es aber nicht. Es können auch Transformationssprachen definiert werden, die nur eine Teilmenge einbeziehen oder komplett eigenständig sind (z. B. XML2DSV). Tabelle B.1 zeigt, welche im Abschnitt 3.2.2 unterschiedenen Sprachmerkmale basierend auf XSLT 1.0 umgesetzt wurden bzw. werden können. Auf einige wird im Folgenden näher eingegangen.

XSLT 1.0 gleicht aufgrund seiner Historie semantisch stark DSSSL (*Document Style Semantics and Specification Language* [Int96a]) und besitzt daher Eigenschaften einer funktionalen Programmiersprache. Tatsächlich können benannte Templates, als Funktionen verstanden werden, die für den aktuell betrachteten Knoten des Quelldokumentes einen Teil des Ergebnisbaums berechnen. Diese Berechnung ist frei von Seiteneffekten. Allerdings wurde XSLT nicht als vollständige funktionale Sprache entworfen. So erlaubt der Funktionsmechanismus keine gleichbenannten Funktionen mit unterschiedlicher Signatur. Ebenso fehlen XSLT explizite Funktionen höherer Ordnung. Wenn gewünscht, kann aber auch das funktionale Paradigma aufgeweicht werden, indem typische, aus imperativen Sprachen bekannte Schleifenkonstrukte (z. B. `while`- und `for`-Schleifen) eingeführt werden. Beide Entwicklungsrichtungen können, wie zum Teil gezeigt, mit dem Operatorhierarchiekonzept realisiert werden.

XSLT 1.0 ist nur schwach typisiert. Es existieren fünf Datentypen: Boolescher Wert, Zahl, Zeichenkette, Knotenmenge und Ergebnisbaumfragment. Während der Transformation werden Werte immer automatisch in den jeweils verlangten Typ umgewandelt. Typfehler treten in XSLT 1.0 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. Mit zusätzlichen Sprachkomponenten könnten explizite, statische Typprüfungen (z. B. von Variablen) vorgenommen werden. Es könnten bei der Ebenentransformation Kontrollblöcke für Vorbedingungen, Nachbedingungen und Invarianten generiert werden, die sichern, dass der Variablenwert in einem bestimmten Wertebereich liegt.

Die von XSLT 1.0 zu erzeugende Zielstruktur kann direkt in den Transformationscode eingebettet werden. Es werden somit keine speziellen Ausgabeanweisungen benötigt. Die Zielstruktur wird dabei mit Hilfe von Literalen (Aneinanderreihung von Zeichenketten) beschrieben, so dass keine zusätzliche Abstraktionsebene notwendig ist. Soll die Zielstruktur XML-basiert sein, kann diese mit Hilfe von XML-Literalen (XML-Zeichenketten) notiert werden. Der Sprachprozessor von XSLT 1.0 erkennt XSLT-Anweisungen anhand ihres Namensraums und kann sie so von den XML-Literalen unterscheiden. Durch die Einbettung der XML-Zielstruktur werden Verletzungen der Wohlgeformtheit (z. B. ein fehlendes literales End-Tag) bereits statisch vom Sprachprozessor erkannt. Sprachkonstrukte für spezielle Markup-Sprachen oder nicht-XML-basierte Zielsprachen werden nicht von XSLT 1.0 angeboten. Mit dem Operatorhierarchiekonzept können solche domänenspezifischen Sprachkonstrukte entworfen werden.

8.3.2 Vergleich mit XSLT 2.0

Mit der zweiten Version von XSLT wurden einige Änderungen an der Sprachdefinition vorgenommen, wobei die Abwärtskompatibilität zur Vorgängerversion gegeben ist. In diesem Abschnitt werden einige der Änderungen hinsichtlich ihrer Umsetzbarkeit mit der Operatorhierarchie untersucht. Eine umfassende Liste der Änderungen kann der Spezifikation entnommen werden [Kay07].

XSLT 2.0 bietet in Verbindung mit XPath 2.0 u. a. die Möglichkeit, benutzerspezifische Funktionen zu definieren. Dazu wurde eigens ein neuer Sprachkonstrukt `function` eingeführt. Die benutzerdefinierten Funktionen können innerhalb eines XPath-Ausdrucks aufgerufen werden. Dadurch können die vordefinierten XPath-Funktionen nach Bedarf ergänzt werden. Unser vorgeschlagenes Framework erlaubt demgegenüber, die Sprache XSLT selbst um weitere Operatoren zu erweitern. Ein Operator wird durch die Strukturierung von womöglich mehreren Elementen und ihren Attributen reflektiert. Ein solcher Operator kann einfacher als die kompakte Repräsentation in XPath validiert sowie generiert werden, z. B. von höheren Operatoren. Zudem sind komplexe Operatoren möglich, die wiederum andere, sowohl ursprüngliche als auch neue Operatoren enthalten können, z. B. `while`-Schleifen. Da das vorgeschlagene Framework ebenso für XSLT 2.0 anwendbar ist, können entsprechende Operatoren auch für XSLT 2.0 konstruiert werden. Der neue Funktionsmechanismus und die Operatorerweiterungen lassen sich somit sehr gut miteinander kombinieren. Theoretisch könnte auch der neue Funktionsmechanismus von XSLT 2.0 simuliert werden. Da aber dafür XPath-Ausdrücke analysiert und transformiert werden müssten, sind XML-Transformationssprachen für diese Aufgabe weniger geeignet. Gleichermaßen unpraktikabel ist die Simulation der neu eingeführten, temporären Speicherung von Bäumen, auf die mit XPath-Ausdrücken zugegriffen werden kann.

Die neuen Sprachkonstrukte für das Gruppieren (z. B. `for-each-group`) können hingegen gut durch höhere Operatoren basierend auf XSLT 1.0 konstruiert werden. Beispielsweise bildet die „Muench'sche Methode“ (nach deren Entwickler Steve Muench [Sil01]) eine Implementierungsmöglichkeit.

Betrachtet man den direkten Vergleich anhand der aufgestellten abstrakteren Sprachmerkmale (siehe Tabelle B.1), so wird deutlich, dass XSLT 1.0 mit zusätzlichen Sprachkomponenten nicht XSLT 2.0 ersetzen kann. Der Hauptgrund dafür ist, dass die Operatoren in den neu entworfenen Sprachkomponenten letztlich auf XSLT 1.0 abgebildet werden. Unterstützt XSLT 1.0 eine elementare Funktionalität nicht, kann diese auch nicht mit höheren Operatoren gewonnen werden. Ein anschauliches Beispiel ist das Erzeugen von Outputs. Mit XSLT 2.0 ist es nun möglich mehrere Zieldokumente zu erzeugen. Diese elementare Funktionalität kann bei XSLT 1.0 nicht durch höhere Operatoren aufgebaut werden.

Wiederum auf der anderen Seite können mit dem Operatorhierarchiekonzept höhere Operatoren entwickelt werden, mit denen bspw. eine andere Traversierung als die übliche preorder-Reihenfolge durch das Quelldokument festgelegt werden kann. Wie die Fallbeispiele von Abschnitt 8.2.1 verdeutlicht haben, können mit dem Framework ebenfalls domänenspezifische Operatoren für eine ganz bestimmte Transformationsaufgabe definiert werden. Beides ist mit

XSLT 2.0 so nicht möglich. Insgesamt zeigen die angeführten Beispiele, dass das Operatorhierarchiekonzept und die daraus entwickelten Sprachkomponenten als Ergänzung zu einer Transformationssprache gesehen werden können, nicht als vollständiger Ersatz.

Kapitel 9

Diskussion

In diesem Kapitel werden bisherige Erfahrungen, die bei der Umsetzung der XSLT-Operatorhierarchie sowie bei der Erstellung spezifischer Sprachen für die Fallbeispiele aus Abschnitt 8 gesammelt wurden, diskutiert. Zunächst werden die Vorteile von Transformationsdefinitionen mit höheren Operatoren aufgeführt, bevor auf die notwendigen Investitionen eingegangen wird.

9.1 Nutzen

Durch Abstraktion vom Ausdrucksniveau heutiger Transformationssprachen durch höhere Operatoren können Transformationen auf einer abstrakteren Ebene beschrieben werden. Höhere Operatoren können dabei universell für die ganze XML-Transformationsdomäne oder domänenspezifisch für einen Ausschnitt, einen bestimmten Anwendungsbereich, entwickelt werden. Aus der Möglichkeit, bisherige XML-Transformationssprachen anreichern zu können, ergeben sich verschiedene Potentiale, die sich positiv auf die Anwendungsentwicklung auswirken. Einige werden im Folgenden näher diskutiert.

9.1.1 Verständlichkeit und Kompaktheit

Bei heutigen Transformationssprachen sind viele Konzepte und fachliche Zusammenhänge aufgrund der feingranularen Operatoren, die zur Verfügung gestellt werden, schnell nicht mehr erkennbar. Höhere Operatoren können so gewählt werden, dass das fachliche Wissen direkt verkörpert wird. Immer wiederkehrende Codefragmente und Details können herausfaktorisiert werden. Die technischen Details, die für eine maschinelle Verwertbarkeit durch den Sprachprozessor einer Basistransformationssprache notwendig sind, werden aus den Ebenentransformationsdefinitionen erzeugt. Hierdurch kann eine klare und formale Entkoppelung von fachlichen und technischen Code vollzogen werden. Dadurch wird insbesondere bei problemangepassten Transformationssprachen in einer kompakteren und für Domänenexperten verständlicheren Weise die Transformationsstruktur und -logik wiedergegeben. Transformationsdefinitionen mit höheren Operatoren sind daher ein weiterer Schritt in Richtung selbstdokumentierter Transformationsbeschreibungen.

Universelle Operatoren

Ob universelle Operatoren die Verständlichkeit und Kompaktheit steigern, hängt stark von ihrer Intention ab. Eine Intention ist es, häufig verwendete elementare und u. U. andere universelle Operatoren zu kapseln. Entsprechend werden Redundanzen vermieden und der Code kompakter, wie die Beispieloperatoren `move-after` und `swap` aus dem Abschnitt 4.1.1 anschaulich belegen (Reduzierung der Codemenge um 69% bzw. 84%). Durch die erzielte Kompaktheit wird der Code i. d. R. auch verständlicher. Der Beispielcode 9.1 zeigt diesen Effekt an dem `for`-Operator der `control`-Sprachkomponente und dessen Implementierung in XSLT.

```
<xsl:template match="/">
  ...
 <ctrl:for name="columns"
   from="1" to="10" step="1">
   <!-- Content -->
 </ctrl:for>
  ...
</xsl:template>
```

```
<xsl:template match="/">
  ...
 <xsl:call-template name="for-columns-d6e183">
  <xsl:with-param name="columns" select="1" />
  <xsl:with-param name="to" select="10" />
  <xsl:with-param name="step" select="1" />
 </xsl:call-template>
  ...
</xsl:template>

<xsl:template name="for-columns-d6e183">
 <xsl:param name="columns" />
 <xsl:param name="to" />
 <xsl:param name="step" />
 <xsl:if test="$columns &lt;= $to">
   <!-- Content -->
 </xsl:if>
 <xsl:if test="$columns + $step &lt;= $to">
  <xsl:call-template name="for-columns-d6e183">
   <xsl:with-param
     name="columns" select="$columns + $step" />
   <xsl:with-param name="to" select="$to" />
   <xsl:with-param name="step" select="$step" />
  </xsl:call-template>
 </xsl:if>
</xsl:template>
```

Code 9.1: Vergleich des `for`-Operators der `control`-Sprachkomponente und dessen Implementierung in XSLT.

Eine andere Motivation liegt in der Verbesserung von bestimmten Eigenschaften bisheriger Operatoren. Dies können funktionale, nichtfunktionale oder strukturelle Eigenschaften von Operatoren sein. Die neuen `function`- und `call-function`-Operatoren der `control`-Sprachkomponente bspw. führen einen sichereren und flexibleren Funktionsmechanismus in XSLT ein. Hierdurch soll das potentielle Fehlerrisiko verringert werden. Eine signifikante Verbesserung der Verständlichkeit oder der Kompaktheit wird durch diese Operatoren nicht erzielt.

Domänenspezifische Operatoren

Domänenspezifische Operatoren nehmen eine besondere Rolle ein. Es können bspw. spezielle Operatoren entwickelt werden, die nur Sprachkonstrukte von einer bestimmten Zielsprache erzeugen können. Es ist aber ebenso denkbar, quellsprachenspezifische Operatoren zu erstellen.

Dadurch kann die Verständlichkeit einer Transformationsdefinition erheblich verbessert werden. Wie die Abbildung 9.1 zeigt, kann darüber hinaus eine kompaktere Beschreibung erreicht werden.

Ist die Zielsprache XML-basiert, kann die Notation bspw. so gewählt werden, dass sie den Sprachkonstrukten der Zielsprache entspricht. Dadurch sind die neuen Transformationsoperatoren leicht zu erlernen. Neben der angepassten Notationsform der Operatoren trägt ebenso die Trennung zwischen spezifischen Operatoren, die die Sprachkonstrukte der Zielsprache erzeugen, und generischen Kontrolloperatoren, die bspw. für Fallunterscheidungen notwendig sind, für eine erhöhte Lesbarkeit der Transformationsdefinitionen bei. Unter Umständen kann diese Trennung allerdings dazu führen, dass der Code für die Erzeugung eines bestimmten Sprachkonstruktes umfangreicher wird, da dieser nicht mehr in seine Einzelbestandteile zerlegt werden kann. Dies trat aber in unseren bisherigen Beispielimplementationen selten auf. Eine Reduzierung der Menge des Quellcodes kann mit dieser Notation der domänenspezifischen Operatoren gegenüber den generischen Operatoren zur Erzeugung von XML-Elementen, Attributen, etc. erreicht werden. Gegenüber literalen XML-Ergebniselementen kann keine Verringerung erzielt werden. Wie bspw. im Abschnitt 9.1.2 erläutert wird, können jedoch zusätzliche Vorteile verwirklicht werden.

Weitergehende Analysen zeigen, dass die Notation theoretisch auch gegenüber den literalen XML-Ergebniselementen verkürzt werden kann. Sollen bspw. einfache XML-Elemente im Zieldokument erzeugt werden, die keine Attribute und nicht weiter verschachtelt sind, so müssen diese nicht als eigene Operatoren umgesetzt werden. Sie könnten im Vateroperator als Attribut einfließen. Da damit von der Reihenfolge abstrahiert wird, muss diese Information aus der Ebenentransformationsdefinition gewonnen werden. Ebenso ist ein Mechanismus einzuführen, der ursprüngliche Attribute und durch einfache Kindelemente zusätzlich hinzugekommene Attribute voneinander unterscheiden kann, da diese mglw. die gleichen Namen haben könnten. Eine andere Möglichkeit ist es, Standardwerte oder redundante Werte für Attribute und Elemente automatisch zu generieren. Beide beispielhaften Vorschläge erschweren das Entwickeln der Ebenentransformation wie [Sch07] zeigt.

Bei der Transformation von XML-Dokumenten in nicht-XML-basierte Dokumente kann die Syntax der Sprachkonstrukte der Zielsprache nicht übernommen werden, wenn für die Ebenentransformation XML-Transformationssprachen sinnvoll wiederverwendet werden sollen. Dies setzt voraus, dass die Operatoren XML-basiert formuliert werden. Deshalb müssen hier eigene XML-Notationen für Operatoren entwickelt werden. Die Namensgebung und der Aufbau der Operatoren können aber wiederum den Konzepten und Terminologien der Zielsprache, wie z. B. die Transformationssprache XML2DSV im Abschnitt 8.2.1 zeigt, entnommen werden, so dass auch hier die neuen Transformationsoperatoren leicht erlernt werden können.

Wird eine separate spezifische Transformationssprache entwickelt, kann u. U. eine enorme Kompaktheit erzielt werden. Abbildung 9.1 veranschaulicht exemplarisch die reine Reduzierung der Menge des Quellcodes am Beispiel der vorgestellten Transformationssprache XML2DSV. Dabei wird die Referenzimplementierung von reinem XSLT (im Code 8.2.1 muss der `function`-Operator und `call-function`-Operator noch aufgelöst werden) mit der Referenzimplementie-

Abbildung 9.1: Verhältnis von XSLT-Code zu XML2DSV-Code.

rung von XML2DSV (siehe Code 8.7) verglichen. Betrachtet man lediglich diese Codequellen, dann kann eine Reduzierung um 75% der domänenspezifischen Sprache gegenüber XSLT festgestellt werden.

Generierte Operatoren

Die Bedenken, dass die generierten, letztendlich elementaren Operatoren, wie aus der Ebenentransformation, schlechter (z. B. weniger lesbar oder performant) seien als von Hand geschriebene, sind i. d. R. unbegründet. Dies hängt vollständig von der Qualität der Ebenentransformationsdefinition ab. Wird der Code bei umfassenderen Realisierungen aus der Referenzimplementierung abgeleitet (vgl. mit dem Top-down-Entwicklungsvorgehen im Abschnitt 7.1), so ist er qualitativ so schlecht oder so gut, wie die Referenzimplementierung selbst.

Für generierten Code spricht generell, dass gleiche Aspekte vom Ebenentransformator immer wieder gleich aussehend bzw. gleich funktionierend und damit konsistent erstellt werden. Vor allem in größeren Projekten ist es ein massives Problem, dass die üblicherweise unterschiedlich gut qualifizierten Entwickler gleiche Stilrichtlinien und Qualitätsmaßstäbe, was bspw. die Lesbarkeit oder die Effizienz des Codes betrifft, umsetzen. Höhere Sprachkomponenten können helfen, eine immer gleiche Realisierung zu erzwingen. Der Ebenentransformator verankert dabei das Expertenwissen in maschinenverarbeitbarer Form. Je spezifischer die Sprachkomponenten sind, umso mehr Wissen kann in die Ebenentransformation aufgenommen werden, z. B. im Rahmen einer betreffenden Domäne.

9.1.2 Prüfbarkeit

Eine Vielzahl von Aspekten, die höhere Operatoren mit sich bringen, wirken sich positiv auf den Aufwand für das Überprüfen und Testen aus. Fehler können im Allgemeinen durch unerwarteten oder inkorrekten Input oder fehlerbehaftete Transformationsdefinitionen entstehen. Zwar wird

durch höhere Operatoren eine dritte mögliche Fehlerquelle hinzugefügt – die Ebenentransformation – jedoch werden die einzelnen Fehlerquellen deutlich entlastet.

Durch das Zusammenfassen von repetitiven Operatorkombinationen zu höheren Operatoren werden Redundanzen in der Transformationsdefinition reduziert und dadurch das Fehlerrisiko (z. B. Fehler durch *Copy & Paste*) gesenkt. Die aus dem Schema mit Hilfe von XOpGen generierten Ebenenvalidierer können die Syntax der verwendeten höheren Operatoren überprüfen. Dadurch werden indirekt die repetitiven Operatorkombinationen validiert, was ohne höhere Operatoren nicht ohne Weiteres automatisch möglich gewesen wäre. Das für die Schemadefinition genutzte XML Schema wurde dabei so erweitert, dass damit auch gegenseitige Abhängigkeiten (z. B. zwischen Attributen oder Attributwerten eines Operators) spezifiziert werden können. Werden bei der Schemadefinition Wertebereichsfestlegungen durch vor- oder benutzerdefinierten Datentypen formuliert, so können bei der Verwendung von XOpGen entsprechende Validierungsblöcke generiert werden, die wiederum die Inputdaten prüfen. Hierdurch können falsche Inputdaten frühzeitig erkannt und, wenn nötig, Verbesserungen an der Implementierung, die von der Ebenentransformation generiert wird, vorgenommen werden. Durch den Einsatz beider Validierungskomponenten können bei auftretenden Fehlern detailliertere Angaben über deren Ursache gemacht werden. Bei der Anwendungsentwicklung, bei der Nutzung der Operatoren, ist dafür kein zusätzlicher Aufwand notwendig (z. B. explizite Kennzeichnung von Datentypen usw.).

Insbesondere bei domänenspezifischen Operatoren, die i. d. R. weitreichendere Einschränkungen aufgrund ihres speziellen Anwendungsbereiches haben, kann eine umfassendere Fehlererkennung erfolgen. Als Beispiel soll die `XHTML`-Sprachkomponente dienen. Die darin definierten XHTML-spezifischen Operatoren können exakt genauso, wie die literalen Ergebniselemente gebraucht werden. So wäre der Code 4.4 des zweiten Anwendungsszenarios im Abschnitt 4.1.2 bis auf den Namensraum, der nun die Ebenentransformationsdefinition identifiziert, identisch. Die XHTML-spezifischen Operatoren leisten aber deutlich mehr als die literalen Ergebniselemente. Zwar können mit literalen Ergebniselementen und elementaren Erzeugungsanweisungen ebenso Dokumente der Zielsprache erzeugt werden, dass sie jedoch korrekt sind, kann in keiner Weise garantiert werden (vgl. mit Abschnitt 4.1.2). Dies führt zu einem typischen Probierprozess: Eine Transformationsdefinition wird geändert und anschließend wird geprüft, z. B. mit einem Schema wie DTD oder XML Schema, ob das Ergebnis den Vorstellungen des Entwicklers entspricht. Erfahrungen mit XSLT und anderen Transformationssprachen haben gezeigt, dass dies ein aufwendiger und zugleich fehleranfälliger Prozess ist. Dies hat mehrere Ursachen:

- Üblicherweise wird nicht nach jeder Änderung einer Transformationsdefinition diese sofort getestet. Unter der Annahme, ein solcher Test würde jedesmal durchgeführt, muss immer noch gewährleistet sein, dass das Beispielquelldokument alle Testfälle abdeckt, um eine Korrektheit zu beweisen. Dies ist in den allermeisten Fällen nicht möglich.

- Wird nun doch ein Fehler im Zieldokument durch ein entsprechendes Schema der Zielsprache entdeckt, kann nur gesagt werden, wo sich der Fehler im Zieldokument befindet. Es kann aber nicht gesagt werden, wo er bei der Transformation entstanden ist und was die

Ursache dafür war. Dies könnten bspw. eine fehlerhafte Änderung innerhalb der Transformationsdefinition oder aber falsch angenommene Eingabedaten des Quelldokumentes zu verantworten haben.

Kurzum es muss viel Zeit investiert werden, um einerseits zu testen, ob ein Fehler durch die Transformationsdefinition auftreten kann, und um andererseits die Ursache von aufgetretenden Fehlern zu lokalisieren.

Erfahrungen bei bisherigen Implementierungen zeigen, dass durch den Einsatz von zielsprachenspezifischen Operatoren dieser Probierprozess deutlich verkürzt wird. Dies wird dadurch ermöglicht, dass bei der Ebenentransformation überprüft wird, ob korrekte Operatoren verwendet wurden. In Abhängigkeit davon wie intensiv die Operatoren eingeschränkt wurden, werden an der Stelle Fehler gemeldet, an der sie auftreten. Die Fehlermeldungen können mit konkreten Hinweisen versehen werden. So können eine Vielzahl von vor allem syntaktischen Fehlern beim Ausführen der Ebenentransformation gefunden werden.

Betrachtet man erneut den Beispielcode 4.4, so würde der mit der Standardeinstellung von XOpGen generierte Ebenenvalidierer folgende Fehler melden:

```
------------------------------------------------
Starting the validation of the higher-level transformation definitions.
------------------------------------------------
Error: The attribute 'height' is not allowed on the operator 'table'.
Path: /xsl:stylesheet[1]/xsl:template[1]/html[1]/body[1]/table[1]/@height

Error: The value of the attribute 'align' is not allowed on the element 'tr'. It must be 'left' or 'center'
  or 'right' or 'justify' or 'char'.
Path: /xsl:stylesheet[1]/xsl:template[1]/html[1]/body[1]/table[1]/tr[1]

Error: The value of the attribute 'valign' is not allowed on the element 'tr'. It must be 'top' or 'middle'
  or 'bottom' or 'baseline'.
Path: /xsl:stylesheet[1]/xsl:template[1]/html[1]/body[1]/table[1]/tr[1]

3 error(s) detected.
```

In diesem speziellen Fall, indem die spezifischen Operatoren exakt gleich den Sprachkonstrukten der XML-basierten Zielsprache sind, kann ein generischer Ebenentransformator (siehe Code 6.7) eingesetzt werden, so dass aus einem gegebenen Schema voll automatisch sämtliche benötigten Komponenten mit XOpGen erzeugt werden können.

Zusätzlich können neben den elementaren Operatoren, die die eigentliche Überführung auf die niedrigsten Ebene übernehmen, auch Überprüfungen auf unterster Ebene generiert werden. Eine solche Validierung kann bspw. testen, ob die Daten des Quelldokumentes den korrekten Datentyp für eine Weiterverarbeitung enthalten (vgl. mit Codeausschnitt 6.4). Der Entwickler, der z. B. den `tr`-Operator für eine XHTML-Tabelle einsetzt, braucht sich nicht mehr darum zu kümmern. Grundsätzlich kann allerdings lediglich gesichert werden, dass Sprachkonstrukte richtig erzeugt werden. Um eine vollständige Korrektheit zu garantieren, müsste eine komplett eigenständige domänenspezifische Transformationssprache entwickelt werden (wie z. B. XML2DSV).

Ungeachtet dessen kann neben der obigen Standardausgabe auch eine andere Ausgabe erzeugt werden, z. B. ein HTML-Report oder ein anderes Format, das eine Entwicklungsumgebung für deren Fehleranzeige verarbeiten kann. Die offenen Variationspunkte beim Generatorsystem XOpGen erlauben eine entsprechende Anpassung des Generates.

9.1.3 Wiederverwendbarkeit

Mit Wiederverwendung wird eine Qualitätsverbesserung, Verkürzung der Entwicklungszeit und Kostensenkung von Anwendungen angestrebt. Höhere Operatoren fassen immer wiederkehrende Codefragmente unabhängig oder abhängig von einer Domäne zusammen. Sie haben deshalb entsprechend generell oder in einer abgegrenzten Domäne ein bestimmtes Wiederverwendbarkeitspotential. Der Wiederverwendbarkeitsgrad kann dabei unterschiedlich groß sein. Dieser hängt sowohl von den funktionalen Eigenschaften (Funktion des Operators) als auch von den nichtfunktionalen Eigenschaften (der Größe des Anwendungsbereiches, der Breite des Wirkungsspektrums und womöglich von den Ausführungkosten) ab. Strukturelle Eigenschaften spielen eher eine untergeordnete Rolle.

Im Operatorhierarchiekonzept kann ein Operator auf zwei Ebenen wiederverwendet werden: auf Anwendungsebene und auf Sprachentwicklungsebene. Die Operatoren der `control`-Sprachkomponente können wie im Beispielcode 8.6 genutzt werden, um eine konkrete Transformationsaufgabe zu lösen. Sie können aber auch Grundlage für höhere Operatoren sein, wie bei der Implementierung der XML2DSV-Tranformationssprache. Bei der Implementierung der Fallbeispiele wurde die `control`-Sprachkomponente am häufigsten wiederverwendet. Sie kam bei allen Transformationsdefinitionen zum Einsatz.

Die einfache Nutzung des Operators, wie bei der Wiederverwendung der `control`-Sprachkomponente, ist die häufigste Form der Wiederverwendung. Sie wird auch als Black-box-Wiederverwendung bezeichnet, da der Operator nicht verändert bzw. so eingesetzt wird, wie er für bspw. eine betreffende Basistransformationssprache erstellt wurde. Eine abgeschwächte Form ergibt sich, wenn er für eine andere Plattform, z. B. eine andere Implementierung oder gar eine andere Basistransformationssprache eingesetzt werden soll. In diesem Fall sind lediglich die Schemadefinition und die Generatoren von XOpGen für die automatische Erstellung der Validierungskomponenten ohne Anpassung nutzbar. Die Generatoren von XOpGen müssen für die neue oder geänderte Plattform nur konfiguriert werden. Die Ebenentransformationsdefinition muss jedoch modifiziert oder ganz neu spezifiziert werden. Man spricht dann auch von einer White-box-Wiederverwendung.

Insgesamt gilt, dass ein Operator sinnvoller ist, je mehr Anwendungen oder Anwendungsteile mit einem Operator entstehen. Desto schneller haben sich die Kosten für seine Erstellung amortisiert (siehe Abschnitt 9.2). In diesem Zusammenhang ist die Dokumentation zum Finden und Verstehen eines Operators oder der gesamten Sprachkomponente, in der er definiert wurde, von besonderer Bedeutung. Gerade das Finden und Verstehen eines Operators, wofür und wie er genutzt wird, sind Voraussetzungen für dessen Wiederverwendung.

9.1.4 Adaptivität

Durch die modulare Entwicklung der Sprachkomponenten können diese entsprechend der konkreten Problemstellung unterschiedlich zusammengesetzt werden. Durch die Möglichkeit, Sprachkomponenten miteinander in verschiedenen Konfigurationen zu einer Sprache zu kombinieren, wird die Wiederverwendbarkeit der einzelnen Sprachkomponenten erhöht. Dies ist unabhängig davon, ob es sich um eine universelle oder eine domänenspezifische Sprachkomponente handelt. Zudem vermeidet die Modularisierung monolithische, unüberschaubare Sprachen. Einfache Problemstellungen können mit einer womöglich kleinen Anzahl von Sprachkomponenten gelöst werden. Wenn das Problem komplexer wird, können entsprechend weitere Sprachkomponenten genutzt werden. Hierdurch kann ein hohe Flexibilität bzgl. des Sprachumfangs einer Transformationssprache bewirkt werden. Beispielsweise wurden zum Lösen des Transformationsproblems der Firma Intershop Communications AG die Sprachkomponenten `control`, `string`, `log`, `ISML` und `XHTML` eingesetzt. Dagegen genügen für die Überführung von AML (ARIS Markup Language) in das Format des SMVs (Symbolic Model Verifier) die Sprachkomponenten `control` und `SMV`.

Ein weiterer Vorteil dieser Flexibilität besteht in der Möglichkeit, schneller Transformationssprachen aus vorhandenen Sprachkomponenten zu entwickeln bzw. anzupassen, als konventionelle, monolithische Transformationssprachen. Fragwürdige Sprachkonstrukte können bspw. bei diesen festen Sprachen nicht so leicht entfernt werden, da sie u. U. noch verwendet werden. Zudem besteht generell für die Implementierung der Sprachkomponente (Ebenentransformator, Ebenenvalidierer, etc.) jegliche Freiheit bei der Wahl der Implementierungstechnik.

In der Gesamtheit kann flexibler, schneller und kostengünstiger auf Anpassungen des Problemraums (z. B. durch eine detailliertere Spezifikation) reagiert werden.

9.1.5 Agilität

Durch die Tatsache, dass aus der Schemadefinition der höheren Operatoren eine ganze Reihe von Artefakten generiert werden können (Validierungskomponenten, Dokumentation, Editor-Anpassungen, etc.), können diese gegenseitig konsistent gehalten werden. Messungen an zwei konkreten Beispielen sollen zeigen, wie sich die Verteilung zwischen generischen, manuell erstellten und generierten Artefakten verhält. Als Kenngröße soll die Dateigröße in Byte, die für eine vollständige Implementierung einer Sprachkomponente benötigt werden, dienen.

Die Infrastrukturkomponente bildet den generischen Code. Der manuelle Codeanteil ergibt sich aus der Definition der Sprachkomponente (erweitertes XML Schema), Referenzimplementierung (mit und ohne neuer Sprachkomponente), Implementierung des Ebenentransformators und Konfiguration von XOpGen. Der vorwiegend aus dem Schema generierte Codeanteil setzt sich aus den Validierungskomponenten (Ebenen- und Inputvalidierer) sowie der Dokumentation zusammen. Die Zwischenprodukte, die während der Generation der Validierungskomponenten mit XOpGen anfallen, sowie andere Generate (z. B. Editorkonfigurationen) werden nicht mit in die Analyse einbezogen.

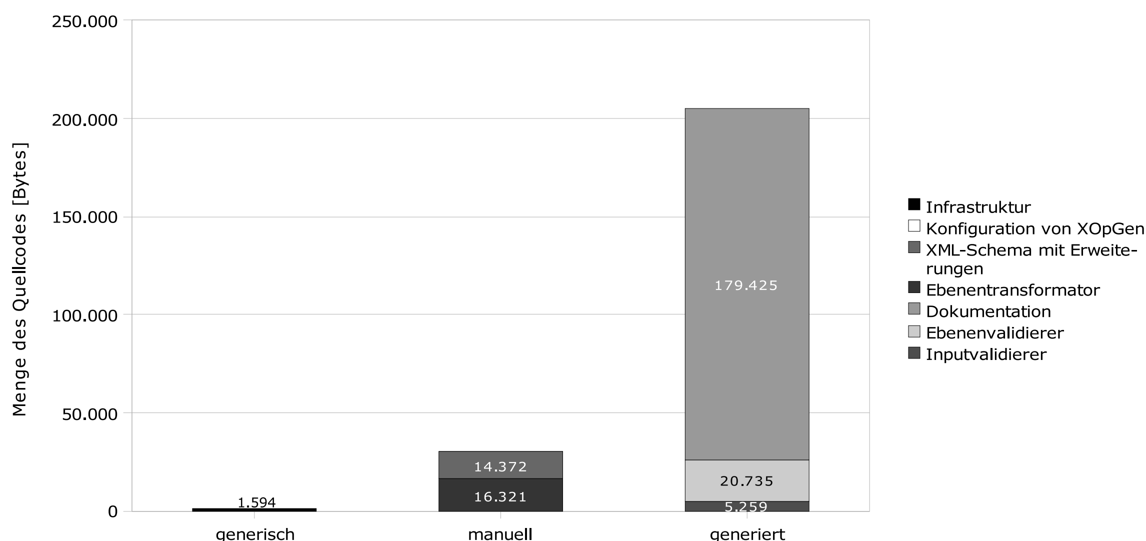


Abbildung 9.2: Verhältnis von generischen zu manuell erstellten und generierten Artefakten am Beispiel der universellen `control`-Sprachkomponente.

Abbildung 9.2 zeigt die Verteilung am konkreten Beispiel der im Abschnitt 8.1.1 vorgestellten `control`-Sprachkomponente für XSLT. Bei dieser universellen Sprachkomponente fällt kein Quellcode für eine Konfiguration des Generators an, da die Standardeinstellung von XOpGen genutzt werden kann. Insgesamt ergibt sich hier ein Verhältnis von generischem Code, manuellem Code und generiertem Code von 0,66 zu 12,74 zu 86,6. Da in der aktuellen Version von XOpGen noch nicht alle potentiellen generierbaren Überprüfungen implementiert sind, wird sich das Verhältnis für diese Sprachkomponente weiter in Richtung generiertem Code verschieben. Beim Beispiel der im Abschnitt 8.2.1 entworfenen domänenspezifischen Transformationssprache XML2DSV ist das Verhältnis zwischen generischem, manuellem und generiertem Code 0,78 zu 17,91 zu 81,31. Abbildung 9.3 verdeutlicht die Verteilung im Einzelnen.

Die Zahlen dieser beiden Beispiele können nicht einfach generalisiert werden, da vieles von der konkreten, zu implementierenden Sprachkomponente (Abhängigkeiten und Integritäten zwischen den Operatoren, Anzahl und Struktur der Operatoren, Zielplattform, etc.) abhängt. Nichtsdestotrotz wird das Generierungspotential angedeutet. Die Trennung zwischen manuellem, universellem und generierbarem Code schafft die Grundlage für eine hohe Portabilität und Agilität. Eine Änderung an *einer* Spezifikation (in XML Schema) wirkt sich auf die verschiedenen Artefakte aus, die aus ihr automatisch erstellt werden. Darüber hinaus kann durch Konfiguration oder Erweiterungen der Generatoren verhältnismäßig einfach und schnell bspw. auf eine andere Version der Basistransformationssprache migriert werden. Lediglich die Ebenentransformationsdefinition muss, wie im Abschnitt 9.1.3 beschrieben, i. d. R. manuell angepasst werden. Nur in Ausnahmefällen kann ein generischer Ebenentransformator eingesetzt werden.

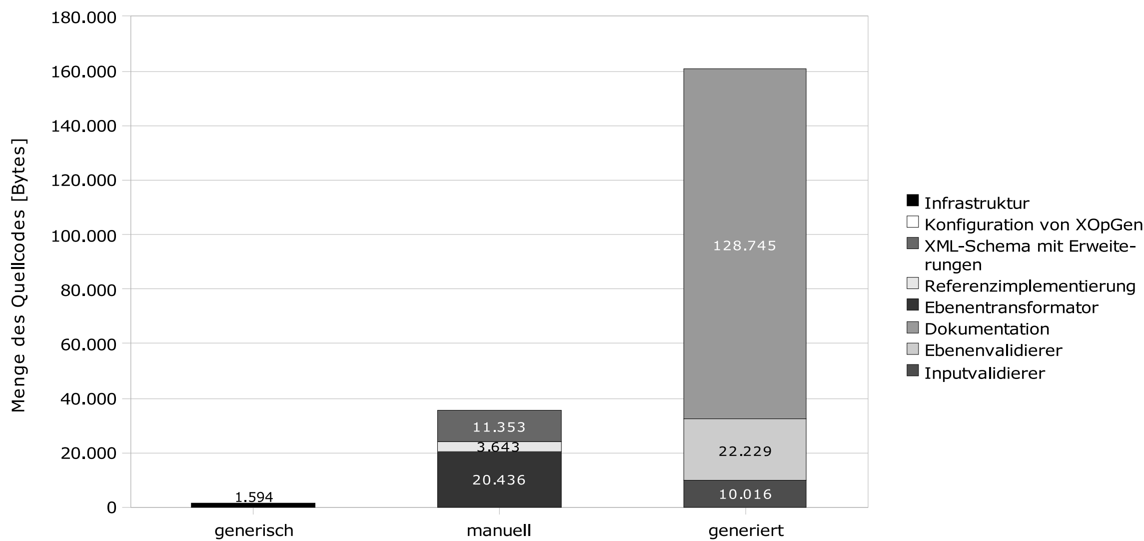


Abbildung 9.3: Verhältnis von generischen zu manuell erstellten und generierten Artefakten am Beispiel der domänenspezifischen Transformationssprache XML2DSV.

9.2 Kosten

Dem Nutzen stehen Kosten für den Aufwand der Erstellung der höheren Operatoren, Sprachkomponenten oder Transformationssprachen gegenüber. Zu dem Initialaufwand kommen Investitionen in Schulung und Infrastruktur sowohl technischer als auch organisatorischer Art hinzu. In diesem Abschnitt wird eine Gesamtsicht über mögliche anfallende Kosten und das daraus hergeleitete Kosten/Nutzen-Verhältnis gegeben. Zunächst wird eine Analyse bzgl. der Codemenge durchgeführt, bevor der Gesamtaufwand diskutiert wird.

9.2.1 Auswirkungen auf die Codemenge

Wird die Kostenanalyse auf Codeebene durchgeführt, dann müssen alle manuell erstellten Artefakte, die für die Entwicklung einer Sprachkomponente notwendig sind, als Kosten betrachtet werden. Die notwendigen manuell erstellten Artefakte für eine Sprachkomponente sind:

- Schema der Sprachkomponente (C_{Schema}), die den syntaktischen und zum Teil semantischen Aufbau der Operatoren in der Sprachkomponente festlegt,
- Konfiguration von XOpgen ($C_{Konfiguration}$), falls bspw. anwendungsspezifische Fehlermeldungen erzeugt werden sollen,
- Ebenentransformator der Sprachkomponente ($C_{Ebenentransformator}$), der die eigentliche Ebenentransformation definiert.

Im Folgenden soll sich auf die Codegröße in Byte bezogen werden. Eine Alternative wäre die Anzahl der Codezeilen (*lines of code*, kurz LOC).

Eine Investition in die Entwicklung einer Sprachkomponente ist grundsätzlich betriebswirtschaftlich kosteneffektiv, wenn die Kosten K für die Investition geringer sind als der daraus erzielte Nutzen N.

$$K < N \tag{9.1}$$

Wenn man eine reine Reduzierung auf die Masse des Quellcodes als Kenngröße annimmt, dann amortisiert sich die Entwicklung einer Sprachkomponente (vgl. mit Formel 9.2), wenn die Codemenge aus manuell erstelltem Code für die Implementierung der neuen Sprachkomponente ($C_{Schema} + C_{Konfiguration} + C_{Ebenentransformator}$) kleiner ist als die Ersparnis, die sich aus dem Code u. U. mehrerer Anwendungsentwicklungen mit dieser Sprachkomponente (C_i^{SK} mit $i \in \mathbb{N}$) gegenüber dem Code von Anwendungsentwicklungen ohne diese Sprachkomponente (C_i mit $i \in \mathbb{N}$) erreichen lässt. Diese Ersparnis stellt gerade den Nutzen auf Codeebene dar.

$$C_{Schema} + C_{Konfiguration} + C_{Ebenentransformator} < \sum_{i=1}^{n}(C_i - C_i^{SK}) \tag{9.2}$$

Die Kosten/Nutzen-Relation, auch *Return-On-Investment* (ROI) genannt, ergibt sich dann zu:

$$ROI = \frac{\sum_{i=1}^{n}(C_i - C_i^{SK})}{C_{Schema} + C_{Konfiguration} + C_{Ebenentransformator}} \tag{9.3}$$

Wenn ROI kleiner als 1 ist, dann verzeichnet die Investition einen Verlust. Ist ROI größer als 1, dann bringt sie einen Gewinn. Um eine Sprachkomponente möglichst kosteneffektiv zu gestalten, bieten sich demnach drei möglich Strategien an:

- hochgradig wiederverwendbare Sprachkomponenten entwickeln, um die Häufigkeit der Wiederverwendung zu erhöhen,
- Investitionskosten für die Entwicklung von Sprachkomponenten reduzieren (z. B. Generierungsgrad weiter erhöhen, um manuell zu erstellenden Code zu minimieren) und
- qualitativ hochwertige Sprachkomponenten erstellen, die viel Einsparpotential gegenüber der ursprünglichen Sprachunterstützung aufweisen.

Betrachtet man nun den `move-after`-Operator und den `swap`-Operator aus dem Anwendungsszenario des Abschnittes 4.1.1, dann amortisiert sich deren Entwicklung bei einer 13-maligen bzw. 6-maligen Anwendung (siehe Tabelle 9.1). Bei dieser Berechnung wurden jeweils die ungünstigsten Eingaben für die höheren Operatoren angenommen, die im Verhältnis am wenigsten Code erzeugen. Die Entwicklung der beiden Operatoren kann sich daher schon für ein kleineres n rentieren.

Eine alleinige Analyse der Codemenge ist für eine Betrachtung des Arbeitsaufwandes allerdings nur begrenzt geeignet, da sich sowohl die (Weiter-)Entwicklung einer Sprache als auch die Anwendungsentwicklung nicht nur auf das Codieren reduzieren lässt. Beispielsweise würde sich

Operator	C_{Schema}	$C_{Konfiguration}$	$C_{Ebenentransformator}$	C_i	C_i^{SK}	n (*break even*)
`move-after`	728	0	1344	202	35	12,41
`swap`	699	0	1713	458	34	5,69

TABELLE 9.1: Analyse des `move-after`-Operators und des `swap`-Operators auf Codeebene.

bei einer solchen eingeschränkten Analyse die Erstellung des `function-` und `call-function-`Operators (siehe Abschnitt 8.1.1) nicht lohnen, da dessen Intention nicht die Verringerung der Codemenge, sondern vielmehr die Verringerung des Fehlerpotentials ist.

9.2.2 Auswirkungen auf den Gesamtaufwand

Für eine umfassende Betrachtung des Gesamtaufwandes müssen jegliche Kosten berücksichtigt werden. Die folgende Liste führt die wesentlichen Kostengruppen aus unserer Sicht auf:

- Entwicklungskosten der Sprachkomponente (K_E),
- Start- und Overheadkosten (K_{SO}) und
- sonstige Kosten (K_S).

Im Folgenden werden die einzelnen Kostengruppen weiter aufgeschlüsselt.

9.2.2.1 Entwicklungskosten der Sprachkomponente

Für die Aufwandsabschätzung der Sprachentwicklung müssen jegliche Kosten für die Analyse und Dokumentation der Anforderungen, für die Definition und den Entwurf bis hin zur Testerstellung und Umsetzung der Sprachkomponente beachtet werden. Die notwendigen Aktivitäten für die Entwicklung von universellen und domänenspezifischen Sprachkomponenten unterscheiden sich dabei beträchtlich. Aus diesem Grund werden diese getrennt untersucht.

Universelle Sprachkomponenten

Diese Sprachkomponenten enthalten universelle Operatoren, welche keine fachlich motivierte Domänenanalyse voraussetzen und sich im Wesentlichen auf die Generierung von wiederholenden Codefragmenten beschränken. Durch ein mögliches Bottom-up-Vorgehen (siehe Abschnitt 7.2) ist der Investitionsaufwand insgesamt vergleichsweise gering. Die Identifizierung von Operatoren kann während der Anwendungsprogrammierung oder in einem separaten Schritt im Anschluss durchgeführt werden. Dadurch entstehen die Hauptkosten für die Erstellung der Operatoren in erster Linie bei deren Definition sowie der Umsetzung des Ebenentransformators basierend auf den identifizierten Codefragmenten. Die tatsächlichen Kosten sind deshalb vorrangig von der Breite des Wirkungsspektrums und der Größe des Anwendungsbereiches der zu erstellenden Operatoren abhängig (siehe Abschnitt 5.1). Je breiter das Wirkungsspektrum eines Operators

auf seine Umwelt und je größer dessen Anwendungsbereich sein soll, desto komplexer und aufwendiger ist die Erstellung des universellen Operators.

Domänenspezifische Sprachkomponenten

Eine erfolgreiche Entwicklung solcher fachlich motivierten Sprachkomponenten setzt erhebliche Erfahrungen in der betreffenden Transformationsdomäne voraus und verlangt daher eine gebührende Domänenanalyse. Es müssen Anwenderzielgruppen festgelegt, Terminologien und Konzepte der Domäne erfasst, Quell- und Zielsprachen eingegrenzt sowie das gesammelte Wissen gebündelt und aufbereitet werden (siehe Top-down-Vorgehensmodell im Abschnitt 7.1). Erst dann ist eine vernünftige Definition der domänenspezifischen Operatoren möglich. Ebenso kann i. d. R. nicht, wie bei universellen Operatoren, auf eine bestehende Implementierung der Operatoren in einer Zielplattform aufgesetzt werden. Entsprechend muss ein umfassendes Referenztransformationsszenario erstellt werden. Insgesamt sollte deshalb der Gesamtaufwand nicht unterschätzt werden, der beträchtlich höher sein kann als bei den universellen Sprachkomponenten. Diese „Denkarbeit“ wird bei der Analyse der Codemenge in keiner Weise gewürdigt.

9.2.2.2 Startup- und Overhead-Kosten

Unter diese Kategorie fallen die Kosten, die vor der Nutzung von Sprachkomponenten (Startup-Kosten) oder während der Benutzung auftreten und sich u. U. wiederholen können (Overhead-Kosten). Da viele Kosten nicht einer Kostenposition explizit zugeordnet werden können, werden beide zusammen untersucht. Typische Kosten dieser Kategorie sind

- Kosten für das Finden oder auch Nichtfinden der Sprachkomponente (K_F),
- Kosten für das Verstehen der Sprachkomponente (K_V),
- Kosten für die mögliche Anpassung oder Erweiterung der Sprachkomponente (K_A).

Es zählen aber auch organisatorische Kosten dazu wie:

- Kosten für den Aufbau der Infrastruktur (K_I): z. B. Aufbau und Wartung des Archivs, Katalogisierung und Installation, Versionskontrolle,
- Kosten für das Training der Mitarbeiter (K_T),
- Managementkosten (K_M): z. B. Anreizzahlungen für die Verwendung von Sprachkomponenten.

9.2.2.3 Sonstige Kosten

Dieser Kategorie werden all diese Kosten zugeordnet, die projektspezifisch und deshalb nicht zu generalisieren sind. Solche Kosten sind bspw. Migrationskosten von Sprachkomponenten auf eine andere Plattform oder die Entwicklung einer eigenen, speziellen Editorumgebung für eine neue spezifische Transformationssprache.

9.2.3 Schlussbemerkungen

An Metriken für die Schätzung des Gesamtaufwandes von Software im Allgemeinen fehlt es nicht. In [Zus91, FP98, Sne98] und anderen werden auf mehr als 150 einzelne Metriken hingewiesen. Allein für die ökonomische Analyse von Software-Wiederverwendung werden in [Wil99] 25 verschiedene Modelle vorgestellt. Allerdings können diese nicht ohne weiteres auf die Weiterentwicklung von XML-Transformationssprachen übertragen werden, da nicht alle spezifischen Kosten einbezogen werden.

Wie die Kostenpositionen im Einzelnen für die Schätzung des Gesamtaufwandes quantifiziert werden und wie sie im Zusammenhang stehen, muss empirisch an umfangreichen praktischen Anwendungen untersucht werden. Das folgende Kostenmodell für die Gesamtkosten, in Anlehnung an das vorgeschlagene Framework von [Wil99],

$$K = K_E + K_{SO} + K_S \tag{9.4}$$

$$K_{SO} = K_F + K_V + K_A + K_I + K_T + K_M \tag{9.5}$$

müsste entsprechend kalibriert werden. Dies kann bspw. mit Hilfe von empirischen Konstanten, wie beim COCOMO von [Boe81], verwirklicht werden.

Diesen Kosten muss dem Nutzen, der sich durch die Verwendung der Sprachkomponente bei der Anwendungsentwicklung ergibt, gegenübergestellt werden. Durch Wiederverwendung können bestimmte Aufgaben schneller und mit weniger Aufwand als bisher gelöst werden. Die bessere Verständlichkeit und Prüfbarkeit erhöhen zudem die Qualität und verringern die Kosten für die Wartung. Die Angepasstheit der Transformationssprache bspw. bzgl. einer bestimmten Transformationsdomäne wirkt sich ebenso positiv auf die Produktivität aus.

Der Nutzen ergibt sich insofern aus den Kostenvorteilen, die sich aus der Differenz der Kosten des gesamten Lebenszyklus einer Anwendungsentwicklung ohne die Sprachkomponente (K_{AE_i}) und der Kosten des gesamten Lebenszyklus einer Anwendungsentwicklung mit der Sprachkomponente ($K_{AE_i^{SK}}$) berechnen lassen. Diese direkten Kostenvorteile verteilen sich auf viele Anwendungsentwicklungen.

$$N = \sum_{i=1}^{n} (K_{AE_i} - K_{AE_i^{SK}} + N_{Z_i}) \tag{9.6}$$

Neben den direkten Kostenvorteilen sollten auch indirekte Kostenvorteile nicht vernachlässigt werden, die aus den produktiveren Anwendungsentwicklungen resultieren. In [MW93] werden zusätzliche Nutzen (N_{Z_i}) genannt, die u. a. durch kürzere Fertigstellungszeiten (*time to market*) und alternative Nutzung der freiwerdenden Ressourcen (z. B. um den Kundenanforderungen gerechter zu werden) erzielt werden können. [Lim92] schließt an dieser Stelle auch Kostenersparnisse ein, die durch eine höhere Produktivität vermieden werden können (z. B. Kosten für das Verpassen einer Deadline oder das Anwerben von weiteren Mitarbeitern).

Kapitel 10

Zusammenfassung und Ausblick

Ziel dieser Arbeit ist es, die notwendigen technischen Voraussetzungen zu schaffen, um die existierenden starren XML-Transformationssprachen flexibel anpassen zu können. Bisherige XML-Transformationssprachen bieten keine Mittel, um neue Sprachkonstrukte hinzuzufügen. Das in der vorliegenden Arbeit vorgestellte Konzept der Operatorhierarchie füllt diese Lücke. Es ermöglicht, basierend auf existierenden Operatoren (Sprachkonstrukten) einer Basistransformationssprache, neue Operatoren zu definieren, die eine bestimmte Aufgabe besser erfüllen, da sie bspw. kompakter, für ein spezielles Problem angepasster oder einfach nur weniger fehleranfällig und somit sicherer sind. Die neuen Operatoren können wiederum Ausgangspunkt für eine weitere Operatoradaption bzw. -komposition sein. Durch das feingranulare Konzept der Operatorhierarchie können so einzelne Sprachkonstrukte, größere Sprachkomponenten, aber auch gänzlich eigenständige Transformationssprachen aufbauend auf einer gegebenen Basistransformationssprache konstruiert werden. Da das Konzept grundsätzlich unabhängig von einer konkreten Transformationssprache ist, können eine Vielzahl von heutigen und zukünftigen Basistransformationssprachen angepasst werden.

Mit XTC wird in dieser Arbeit eine Infrastruktur vorgestellt, die das Fundament für die Umsetzung des vorgeschlagenen Konzeptes bildet. XTC stellt Mechanismen zur Verfügung, um in Transformationsbeschreibungen höhere Operatoren zu erkennen und die notwendige Ebenentransformationsdefinitionen auszuführen. Ebenentransformationen übersetzen die neu definierten, höheren Operatoren einer Sprachkomponente in niedrigere Operatoren. Der Ebenentransformationsprozess wird solange wiederholt, bis alle höheren Operatoren in elementare Operatoren einer Basistransformationssprache transformiert wurden und somit ausführbar sind. XTC wird dabei durch entsprechende standardisierte Schnittstellen so konzipiert, dass die Möglichkeit besteht, unterschiedlichste Sprachprozessoren für die verschiedenen Transformationssprachen einzubinden. Dadurch können die eingebundenen Sprachen selbst erweitert werden aber auch zur Definition notwendiger Ebenentransformationen für die Erweiterung anderer Transformationssprachen genutzt werden. Hierdurch kann eine hohe Flexibilität bei der Implementierung der Ebenentransformationen erreicht werden.

Um den Aufwand der Definition einer Ebenentransformation weiter zu reduzieren, wird in dieser Arbeit darüber hinaus das Generatorsystem XOpGen entworfen. XOpGen erstellt automatisiert aus einer Schemadefinition in XML Schema in mehreren Generationsschritten Validierungskomponenten, die die syntaktische und zum Teil semantische Analyse der neu definierten Operatoren

vornehmen. Durch diese vorherige Validierung kann sich bei der Ebenentransformationsdefinition auf das Wesentliche konzentriert werden. Die grammatikbasierte Schemasprache wurde für diesen Zweck mit einem regelbasierten Ansatz standardkonform angereichert. Dadurch können vorher nicht formulierbare Abhängigkeiten (z. B. gegenseitiger Ausschluss) spezifiziert werden und in den Generierungsprozess der Validierungskomponenten einfließen. Aus einem solchen erweiterten Schema kann XOpGen ebenso Kontrollblöcke für die Ebenentransformationsdefinition selbst generieren, die den erwarteten Wertebereich der Inputdaten bzgl. der neuen höheren Operatoren später überprüfen. XOpGen ist dabei bspw. nicht an eine bestimmte Fehlermeldung oder festgelegte Basistransformationssprache gebunden. Durch Anpassung definierter Variationspunkte kann anwendungsspezifischer Code hinzugefügt werden, der während der Ausführung von XOpGen verarbeitet wird. XOpGen kann dadurch auch vollkommen unabhängig von XTC und dem Konzept der Operatorhierarchie eingesetzt werden.

Die beiden, in dieser Arbeit vorgeschlagenen Vorgehensmodelle schaffen einen organisatorischen Rahmen für die Nutzung der zuvor entwickelten Konzepte und Werkzeuge. Der Rahmen legt fest, welche Aktivitäten in welcher Reihenfolge von welchen Rollen erledigt werden, welche Werkzeuge genutzt werden und welche Ergebnisse dabei entstehen. Die Vorgehensmodelle berücksichtigen dabei die jeweiligen Motivationen für eine (Weiter-)Entwicklung einer Transformationssprache. Das Top-down-Vorgehensmodell deckt fachlich motivierte Gründe ab. Dagegen sollte das Bottom-up-Vorgehensmodell zur Beseitigung von Defiziten einer konkreten Transformationssprache bevorzugt werden.

Der praktische Nutzen aus Konzept, Infrastruktur und Generatorsystem wird an der Erweiterung von XSLT exemplarisch gezeigt. Es werden sowohl universelle als auch domänenspezifische Sprachkomponenten für diese Transformationssprache entwickelt. Für eine Reihe von Fallbeispielen konnten diese Sprachkomponenten erfolgreich eingesetzt werden. In dieser Arbeit wird die Entwicklung eines sicheren und flexiblen Funktionsmechanismus unter Anwendung des vorgeschlagenen Vorgehens vorgestellt. Ebenso wird die Entwicklung einer eigenständigen, problemspezifischen Transformationssprache demonstriert. Diese Beispiele zeigen die Potentiale, die das entwickelte Framework bietet. Basierend auf den Erfahrungen, die bei der Entwicklung der Operatorhierarchie für XSLT, bei der Erweiterung von XUpdate sowie bei der Anwendung der entwickelten Sprachkomponenten in den verschiedenen Fallbeispielen gemacht wurden, werden Nutzen und Kosten des Operatorhierarchiekonzeptes diskutiert.

Ausblick

Die Analyse von Nutzen und Kosten wird bisher lediglich vollständig auf Codeebene durchgeführt. Um die exakten Gesamtkosten und den Gesamtnutzen schätzen zu können und Entscheidung vor der Entwicklung einer Sprachkomponente treffen zu können, müssen aussagekräftige Metriken erschlossen oder existierende Metriken entsprechend adaptiert werden. Diese können aber erst nach weitergehenden praktischen Erfahrungen sinnvoll eingeführt und definiert werden. Allerdings wird auch nach Erschließung von aussagekräftigen Metriken eine exakte objektive Messung bspw. des Nutzens, der aus einer besseren Verständlichkeit einer höheren Transformationssprache resultiert, nicht möglich sein.

Der bisherige Prototyp von XTC hat gezeigt, das einige Konzepte weiter verbessert werden können. Bei einer bestimmten Gruppe von höheren Operatoren – Operatoren, aus denen verteilte Operatoren erzeugt werden – können derartige wechselseitige Abhängigkeiten zwischen den zu erzeugenden Operatoren entstehen, dass die Ausführungsreihenfolge der Ebenentransformationsdefinitionen signifikant ist. Ähnlich wie bei Aspekten in aspektorientierten Ansätzen sind zusätzliche Informationen notwendig, die festlegen, wie die beabsichtigte Ausführungsreihenfolge ist. Die bisherige Umsetzung von expliziter Priorisierung einer Sprachkomponente soll durch deskriptive Beschreibungen, die bspw. Prioritäten zwischen ein oder mehreren Sprachkomponenten erlauben, ergänzt werden. Um den alltäglichen Einsatz weiter zu erleichtern, wäre darüber hinaus ein graphisches Frontend für die Erstellung von Operatoren, Sprachkomponenten und Transformationssprachen sicherlich nützlich. Bisher müssen die Ebenentransformationensdefinitionen in generischen XML-Editoren entwickelt werden. In XTC wurde das Bibliotheksarchiv bisher auf Dateiebene verwaltet. Um bereits vorhandene Sprachkomponenten schnell, sicher und bequem zu finden, ist ein projektübergreifendes, z. B. über ein Netz leicht zugängliches, Archiv förderlich. Es sollte zudem umfangreichere Klassifikations- und Suchtechniken unterstützen. Dahingehend müssen weitere Anstrengungen unternommen werden.

Um das Potential der neuen Operatoren in der Praxis nutzen zu können, muss ein wiederverwendungsorientiertes Vorgehen bei der Anwendungsentwicklung etabliert werden. Es ist wesentlich, dass dieses Vorgehen Aktivitäten vorsieht, in denen neue oder anzupassende Operatoren für Sprachkomponenten identifiziert werden. Es ist aber mindestens genauso wichtig, dass Aktivitäten eingeführt werden, in denen nach geeigneten Sprachkomponenten gesucht und ausgewählt werden kann. Gerade in der Möglichkeit, Sprachkomponenten modular und anwendungsspezifisch auszuwählen, besteht der große Vorteil, Transformationssprachen angemessen für ein Transformationsproblem zusammenzustellen.

Das in dieser Arbeit vorgestellte Framework ermöglicht die Definition von höheren Operatoren auf einer Basistransformationssprache. An dieser Stelle stellen sich folgende interessante Fragen: Wie sieht eine minimale XML-Transformationssprache aus? Kann diese minimale Transformationssprache ferner die in der Arbeit vorgestellten Kategorien von Transformationsmethoden mit Hilfe einer aufsetzenden Operatorhierarchie abdecken? Eine genaue Untersuchung dieser Fragen ist geplant.

Anhang A

Quellcode der Anwendungsbeispiele

A.1 Definition des `function`- und `call-function`-Operators

Der folgende Code zeigt den Ausschnitt des Schemas der `control`-Sprachkomponenten, der den `function`-Operator und den `call-function`-Operator definiert. Das Schema wird in der Schemasprache XML Schema definiert, die mit Schematronregeln angereichert wird.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema
  targetNamespace="http://www.informatik.uni-kiel.de/Control"
  elementFormDefault="qualified"
  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:ctrl="http://www.informatik.uni-kiel.de/Control">

  <element name="declaration" type="ctrl:otherAttributes" abstract="true">
    <annotation>
      <documentation>
        The abstract declaration element encompasses operators who have to be defined on the
        top level. They cannot be locally defined.
      </documentation>
    </annotation>
  </element>

  <element name="instruction" type="ctrl:otherAttributes" abstract="true">
    <annotation>
      <documentation>
        The abstract instruction element encompasses operators who have to be defined in a
        local context. They cannot be defined on the top level.
      </documentation>
    </annotation>
  </element>

  <element name="function" substitutionGroup="ctrl:declaration">
    <annotation>
      <documentation>
        A function definition is specified with the function operator. The content of the
        function operator contains the code, which performs a specific task, when the function
        is invoked. The function operator must have a name attribute. It may have further
        parameters specified by param child elements. The function operator can be invoked by
        a call-function operator by the name and the corresponding parameters.
      </documentation>
    </annotation>
    <complexType>
      <complexContent mixed="true">
        <extension base="ctrl:otherAttributes">
          <sequence>
            <element ref="ctrl:param" minOccurs="0" maxOccurs="unbounded"/>
```

```
          <group ref="ctrl:constructorGroup" minOccurs="0" maxOccurs="unbounded"/>
        </sequence>
        <attribute name="name" type="QName" use="required">
          <annotation>
            <documentation>
              The value of the name attribute is a qualified name.
            </documentation>
          </annotation>
        </attribute>
      </extension>
    </complexContent>
  </complexType>
</element>

<element name="param">
  <annotation>
    <documentation>
      A parameter has a required name attribute, which specifies the name of the parameter.
    </documentation>
  </annotation>
  <complexType>
    <attribute name="name" type="QName" use="required">
      <annotation>
        <documentation>
          The value of the name attribute is a qualified name.
        </documentation>
      </annotation>
    </attribute>
  </complexType>
</element>

<element name="call-function" substitutionGroup="ctrl:instruction">
  <annotation>
    <documentation>
      The call-function operator invokes the function definition . The corresponding function
      definition is identified by the required name, specified in the required name
      attribute , and the names of the parameters, specified in the parameter child elements.
      The call-function operator does not change the current node or the current node list .
      It is an error if a call -function invokes an undefined function definition .
    </documentation>
  </annotation>
  <complexType>
    <complexContent>
      <extension base="ctrl:otherAttributes">
        <sequence minOccurs="0" maxOccurs="unbounded">
          <element ref="ctrl:with-param"/>
        </sequence>
        <attribute name="name" type="QName" use="required">
          <annotation>
            <documentation>
              The value of the name attribute is a qualified name.
            </documentation>
          </annotation>
        </attribute>
      </extension>
    </complexContent>
  </complexType>
</element>

<element name="with-param">
  <annotation>
    <documentation>
      Parameters are passed to functions using the with-param element. The required name
      attribute specifies the name of the parameter. The parameter name is specified in the
```

```
        same way as the corresponding parameter name in the function definition. The value of
        the passed parameter is defined by either a select attribute or the content of the element.
      </documentation>
      <appinfo>
        <sch:assert
          xmlns:sch="http://www.ascc.net/xml/schematron" test="not(@select and *)">
          Element 'with-param' may have either a 'select' attribute or a content.</sch:assert>
      </appinfo>
    </annotation>
    <complexType>
      <complexContent mixed="true">
        <extension base="ctrl:constructorType">
          <attribute name="name" type="string" use="required">
            <annotation>
              <documentation>
                The value of the name attribute is a qualified name.
              </documentation>
            </annotation>
          </attribute>
          <attribute name="select" type="string" use="optional">
            <annotation>
              <documentation>
                The value of the select attribute is an XPath expression.
              </documentation>
            </annotation>
          </attribute>
        </extension>
      </complexContent>
    </complexType>
  </element>

  ...

  <complexType name="otherAttributes">
    <annotation>
      <documentation>
        The complex type enables attributes not specified by the schema. The namespace must
        be another than "http://www.informatik.uni-kiel.de/Control".
      </documentation>
    </annotation>
    <anyAttribute namespace="##other" processContents="skip"/>
  </complexType>

  <complexType name="constructorType">
    <annotation>
      <documentation>
        The complex type enables the mixed application of text and operators of the
        constructor group.
      </documentation>
    </annotation>
    <complexContent mixed="true">
      <extension base="ctrl:otherAttributes">
        <group ref="ctrl:constructorGroup" minOccurs="0" maxOccurs="unbounded"/>
      </extension>
    </complexContent>
  </complexType>

  <group name="constructorGroup">
    <annotation>
      <documentation>
        The group provides either an operator of the abstract instruction element or an
        element of another namespace.
      </documentation>
    </annotation>
```

```
    <choice>
      <element ref="ctrl: instruction"/>
      <group ref="ctrl:otherElements"/>
    </choice>
  </group>

  <group name="otherElements">
    <annotation>
      <documentation>
        The group provides either an element of another namespace or an element without a
        namespace binding.
      </documentation>
    </annotation>
    <choice>
      <any namespace="##other" processContents="skip"/>
      <any namespace="##local" processContents="skip"/>
    </choice>
  </group>

</schema>
```

CODE A.1: Ausschnitt aus dem Schema (`control.xsd`) – der `function`-Operators und der `call-function`-Operators.

A.2 Definition der Transformationssprache XML2DSV

Der Code A.2 legt die Sprachdefinition von XML2DSV fest. Die Definition erfolgt in einem um Schematronregeln erweitertes XML Schema.

```
<?xml version="1.0" encoding="UTF−8"?>
<schema
  targetNamespace="http://www.informatik.uni−kiel.de/DSV"
  elementFormDefault="qualified"
  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:dsv="http://www.informatik.uni−kiel.de/DSV"
  xmlns:sch="http://www.ascc.net/xml/schematron">

  <element name="dsv">
    <annotation>
      <documentation>
        An XML2DSV program is represented by a dsv element in an XML document. The dsv
        element may contain a separator description and must contain a description for the
        data fields and records.

        The mode attribute configures the implementation mode. It can be static or dynamic.
      </documentation>
    </annotation>
    <complexType>
      <sequence>
        <element ref="dsv:separators" minOccurs="0" maxOccurs="1"/>
        <element ref="dsv:data" minOccurs="1" maxOccurs="1"/>
      </sequence>
      <attribute name="mode" use="optional">
        <simpleType>
          <restriction base="string">
            <enumeration value="static"/>
            <enumeration value="dynamic"/>
          </restriction>
        </simpleType>
```

```
    </attribute>
  </complexType>
</element>

<element name="separators">
  <annotation>
    <documentation>
      The optional separator description, specified with the separator element, allows
      to define the specific delimiters.

      If the separator description is omitted, the default setting corresponds to the
      CSV standard values: ending-field-break is "yes" and delimiter for fields is a
      comma, delimiter for field escape is ", and ending-record-break is "no" and
      delimiter for records is a newline.

      Note: The values of the separator characters must be different and it has at
      least one child operator.
    </documentation>
    <appinfo>
      <sch:assert test="dsv:*">The 'separators' operator has at least one child
        operator.</sch:assert>
    </appinfo>
  </annotation>
  <complexType>
    <sequence>
      <element ref="dsv:field-separator" minOccurs="0" maxOccurs="1"/>
      <element ref="dsv:field-escape" minOccurs="0" maxOccurs="1"/>
      <element ref="dsv:record-separator" minOccurs="0" maxOccurs="1"/>
    </sequence>
  </complexType>
</element>

<element name="field-separator">
  <annotation>
    <documentation>
      The field-separator operator states the field delimiter, and whether the last field may
      or may not have an ending delimiter.

      In the default setting the ending-field-break is "yes" and the delimiter is a
      comma.
    </documentation>
    <appinfo>
      <sch:report test="@delimiter and ((../field-escape and @delimiter=../field-escape/
        @delimiter) or @delimiter='&#x22;')">The field separator delimiter has the same
        value as the (default) field escape delimiter. This is not allowed.</sch:report>
      <sch:report test="@delimiter and ((../record-separator and @delimiter=
        ../record-separator/@delimiter) or @delimiter='&#xa;')">The field separator
        delimiter has the same value as the (default) record separator delimiter.
        This is not allowed.</sch:report>
      <sch:assert test="@delimiter or @ending-field-break">The 'field-separator' must
        have an 'ending-field-break' attribute, a 'delimiter' attribute, or both.
      </sch:assert>
    </appinfo>
  </annotation>
  <complexType>
    <attribute name="ending-field-break" type="dsv:YesNoType" use="optional"/>
    <attribute name="delimiter" type="string" use="optional"/>
  </complexType>
</element>

<element name="field-escape">
  <annotation>
    <documentation>
      The field-escape operator states the escape character(s) which may be used to
```

```
        enclose a field .

        In the default setting the field escape delimiter is ".
      </documentation>
      <appinfo>
        <sch:report test="@delimiter and ((../ field −separator and @delimiter=..
          / field −separator/@delimiter) or @delimiter=',')">The field escape delimiter has
          the same value as the (default) field separator delimiter . This is not allowed.
        </sch:report>
        <sch:report test="@delimiter and ((../record−separator and @delimiter=
          ../ record−separator/@delimiter) or @delimiter='&#xa;')">The field escape delimiter
          has the same value as the (default) record separator delimiter . This is not allowed.
        </sch:report>
      </appinfo>
    </annotation>
    <complexType>
      <attribute name="delimiter" type="string" use="required"/>
    </complexType>
  </element>

  <element name="record−separator">
    <annotation>
      <documentation>
        The record−separator operator states the record delimiter , and whether the last record
        may or may not have an ending delimiter.

        In the default setting the ending−record−break is "no" and the delimiter is a
        newline.
      </documentation>
      <appinfo>
        <sch:report test="@delimiter and ((../ field −separator and @delimiter=
          ../ field −separator/@delimiter) or @delimiter=',')">The record separator delimiter
          has the same value as the (default) field separator delimiter . This is not allowed.
        </sch:report>
        <sch:report test="@delimiter and ((../ field −escape and @delimiter=
          ../ field −escape/@delimiter) or @delimiter='&#x22;')">The record separator delimiter
          has the same value as the (default) field escape delimiter . This is not allowed.
        </sch:report>
        <sch:assert test="@delimiter or @ending−field−break">The 'field−separator' must have
          an 'ending−field−break' attribute, a ' delimiter ' attribute , or both.</sch:assert>
      </appinfo>
    </annotation>
    <complexType>
      <attribute name="ending−record−break" type="dsv:YesNoType" use="optional"/>
      <attribute name="delimiter" type="string" use="optional"/>
    </complexType>
  </element>

  <element name="data">
    <annotation>
      <documentation>
        The data element is used to define the delimited data. The corresponding data
        selection is specified by adding record operators as children of the data element.
        There may be an optional header record (header operator) appearing as the first
        record of the DSV with the same format as ordinary records.

        The data, header as well as record, and field operators support an escaped−field
        attribute . It is set if a character, specified in the field −escape operator, is
        used to enclose fields or not; default setting is the latter case. A setting
        shadows another setting if the setting occurs at a point where the other setting
        is visible .

        Note: The field operators in the record and header operators must be the same
        number.
```

```
    </documentation>
    <appinfo>
      <sch:report test="count(dsv:*[1]/dsv: field )!=count(*[position()&gt;1]/dsv: field )">
        The field operators in the record and header operators must be the same number.
      </sch:report>
    </appinfo>
  </annotation>
  <complexType>
    <complexContent>
      <extension base="dsv:field−escapingType">
        <sequence>
          <element ref="dsv:header" minOccurs="0" maxOccurs="1"/>
          <element ref="dsv:record" minOccurs="1" maxOccurs="unbounded"/>
        </sequence>
      </extension>
    </complexContent>
  </complexType>
</element>

<element name="header" type="dsv:recordType">
  <annotation>
    <documentation>
      The header operator appearing as the first record of the DSV with the same format
      as ordinary records.
    </documentation>
  </annotation>
</element>
<element name="record" type="dsv:recordType">
  <annotation>
    <documentation>
      The record operator contains a description of data fields . These are instantiated
      for each node identified by the XPath expression specified by the match attribute.
      If all data fields bind literal data characters, the match attribute in the record
      operator may be omitted.
    </documentation>
  </annotation>
</element>
<complexType name="recordType">
  <complexContent>
    <extension base="dsv:field−escapingType">
      <sequence>
        <element ref="dsv:field"/>
      </sequence>
      <attribute name="match" type="string" use="optional"/>
    </extension>
  </complexContent>
</complexType>

<element name="field">
  <annotation>
    <documentation>
      With respect to these context nodes defined in the record/header operator, the
      concrete data items are selected using the select attribute in the field operator.
      If all data fields bind literal data characters, the select attribute in the record
      operator may be omitted. The literals are specified in the content of the field
      operator.

      Note: It is forbidden to have a select attribute and a content, or to have no
      select attribute and no content. Furthermore, a select attribute is not allowed if
      the parent operator has no match attribute.
    </documentation>
    <appinfo>
      <sch:assert test="(@select and (string−length(text())=0 and not(*))) or (not(@select)
        and (string−length(text())>0 or *))">The 'field' operator must have either a
```

```
          ' select '  attribute  or  a content.</sch:assert>
        <sch:assert  test="not(@select) or (@select and ../@match)">The 'select' attribute is
          not allowed on the ' field '  operator  if  the parent '<sch:name path=".."/>' operator
          has no 'match' attribute.</sch:assert>
      </appinfo>
    </annotation>
    <complexType>
      <complexContent mixed="true">
        <extension base="dsv:field-escapingType">
          <sequence>
            <any namespace="##other" processContents="skip"/>
          </sequence>
          <attribute name="select" type="string" use="optional"/>
        </extension>
      </complexContent>
    </complexType>
  </element>

  <simpleType name="YesNoType">
    <annotation>
      <documentation>The allowed values of this type are 'yes' and 'no'.</documentation>
    </annotation>
    <restriction  base="string">
      <enumeration value="yes"/>
      <enumeration value="no"/>
    </restriction>
  </simpleType>

  <complexType name="field-escapingType">
    <annotation>
      <documentation>
        This type provides the  setting  whether  fields  are escaped or not.
      </documentation>
    </annotation>
    <attribute name="escaped-field" type="dsv:YesNoType" use="optional"/>
  </complexType>

</schema>
```

CODE A.2: Schema der Transformationssprache XML2DSV.

ANHANG B

Sprachmerkmale von XSLT

In diesem Anhang werden die Transformationssprache XSLT 1.0, die mit Sprachkomponenten erweiterte Transformationssprache XSLT 1.0 und die Transformationssprache XSLT 2.0 anhand der im Abschnitt 3.2.2.1 herausgearbeiteten Sprachmerkmalen verglichen. Die Tabelle B.1 gibt eine entsprechende Übersicht, welche Sprachmerkmale jeweils komplett unterstützt (u), bedingt unterstützt (b), quellseitig unterstützt (q), zielseitig unterstützt (z) oder quell-/zielseitig unterstützt werden (qz).

Merkmal / **Sprache**	XSLT 1.0	XSLT 1.0 mit Erweiterungen	XSLT 2.0
Transformationsregeln			
Domain			
DSL		u	
Modus			
Art			
in	u	u	u
in/out			
out	u	u	u
Bindungszeitpunkt			
statisch	u	u	u
dynamisch			
Muster			
Struktur			
Ausdruck	u	u	u
String			
Syntax			
abstrakt	q	qz	q
konkret	z	z	z

Fortsetzung auf der folgenden Seite

Merkmal / Sprache	XSLT 1.0	XSLT 1.0 mit Erweiterungen	XSLT 2.0
Typisierung			
untypisiert			
typisiert			
stark		b	u
schwach	u	u	u
dynamisch	u	u	u
statisch		b	u
implizit	u	u	u
explizit		b	u
Variablen			
Wertspezifikation			
Wertbindung	u	u	u
imperative Zuweisung		b	
Constraints			
Typisierung			
untypisiert	u	u	u
typisiert			
stark		u	u
schwach		b	
dynamisch			
statisch		u	u
implizit			
explizit		u	u
Logik			
Paradigma			
deklarativ	u	u	u
imperativ			
Ausführbarkeit	u	u	u
Elementerzeugung			
implizit		b	
explizit	u	u	u
syntaktische Trennung	b	b	b
explizite Bedingung	u	u	u
intermediäre Datenhaltung	b	b	u

Fortsetzung auf der folgenden Seite

Merkmal / Sprache	XSLT 1.0	XSLT 1.0 mit Erweiterungen	XSLT 2.0
Multidirektionalität			
Parametrisierung			
Art			
Typparameter			
Kontrollparameter	u	u	u
Regel höherer Ordnung		u	
variable Parameterzahl	u	u	u
benannte Parameter	u	u	u
Regelausführungskontrolle			
Traversierung			
Festlegung			
explizit			
intern	u	u	u
extern		u	
implizit	u	u	u
Art			
top-down (pre-order)	u	u	u
bottom-up (post-oder)		u	
inner-most			
outer-most			
benutzerdefiniert		u	
Auswahl			
explizite Bedingung	u	u	u
nichtdeterministisch			
deterministisch			
Fehlermeldung	u	u	u
Konfliktlösungsmechanismen	u	u	u
interaktiv	u	u	u
anwendbare Regelmenge	u	u	u
Kontrollstrukturen			
Iteration			
Rekursion	u	u	u
Schleife		u	
Fixpunkt			

Fortsetzung auf der folgenden Seite

Merkmal / Sprache	XSLT 1.0	XSLT 1.0 mit Erweiterungen	XSLT 2.0
Verzweigung			
ein- und zweiseitige	u	u	u
mehrfach	u	u	u
Regelorganisation			
Modularisierung	u	u	u
Wiederverwendung			
Erweiterung	u	u	u
Spezialisierung	u	u	u
Vererbung			
organisatorische Strukturen			
quellorientiert	u	u	u
hybrid	u	u	u
zielorientiert	u	u	u
Ein- und Ausgabe			
Beziehung zwischen Quelle und Ziel			
neues Ziel	u	u	u
existierendes Ziel	u	u	u
Bereichsfestlegung			
Quelle		u	
Ziel			
mehrere Ziele			u
mehrere Quellen	u	u	u
Abstrahierung	qz	qz	qz
Transformationsrichtung			
unidirektional	u	u	u
bidirektional			
Mechanismen zur Transformationsverfolgung			
manuell	u	u	u
automatisiert		u	
Programmparadigma			
deklarativ	u	u	u
imperativ		b	
Format			
XML-basiert	qz	qz	qz

Fortsetzung auf der folgenden Seite

Merkmal / Sprache	XSLT 1.0	XSLT 1.0 mit Erweiterungen	XSLT 2.0
nicht-XML-basiert	z	z	z

Legende:
u = unterstützt, b = bedingt unterstützt,
q = quellseitig unterstützt, z = zielseitig unterstützt,
qz = quell- und zielseitig unterstützt

TABELLE B.1: Vergleich von XSLT 1.0, XSLT 1.0 mit Erweiterungen und XSLT 2.0 anhand der aufgestellten Sprachmerkmale.

Literaturverzeichnis

[AK03] Colin Atkinson und Thomas Kühne. Model-Driven Development: A Metamodeling Foundation. *Software, IEEE Computer Society*, 20(5):36–41, 2003.

[Alt07] Altova GmbH. *Altova® MapForce 2007 Benutzer- und Referenzhandbuch*, 2007.

[Amo02] Daniel Amor. *Dynamic Commerce – Online-Auktionen – Handeln mit Waren und Dienstleistungen in der Neuen Wirtschaft*. Galileo Press, Oktober 2002.

[Apa01] Apache. Crimson 1.1.3, Release Date 01.11.2001. http://xml.apache.org/crimson/, 2001.

[Apa05] Apache. Xerces, Version 2.9.1, Release Date 15.09.2007. http://xerces.apache.org/, 2005.

[Apa07] Apache Xalan Project. Xalan-Java 2.7.1, November 2007. http://xalan.apache.org/, 2007.

[Arn93] Dennis S. Arnon. Scrimshaw: A Language for Document Queries and Transformations. *Electronic Publishing*, 6(4):385–396, 1993.

[ASF07] ASF+SDF Meta-Environment, Version 1.5, Release Date 22.02.2007. http://www.cwi.nl/htbin/sen1/twiki/bin/view/Meta-Environment, 2007.

[Asp08] AspectJ 1.6.1, Release Date 03.07.2008. http://www.eclipse.org/aspectj, 2008.

[ASU86] Alfred V. Aho, Ravi Sethi und Jeffrey D. Ullman. *Compilers – Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.

[Bad00] Greg J. Badros. JavaML: A Markup Language for Java Source Code. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 33(1-6):159–177, 2000.

[BCF+02] Michael Benedikt, Chee Yong Chan, Wenfei Fan, Rajeev Rastogi, Shihui Zheng und Aoying Zhou. DTD-Directed Publishing with Attribute Translation Grammars. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB'02), August 20-23, 2002, Hong Kong, China*, Seiten 838–849. Morgan Kaufmann, 2002.

[BCF03] Véronique Benzaken, Guiseppe Castagna und Alain Frisch. CDuce: An XML-Centric General-Purpose Language. In *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming (ICFP'03)*, Seiten 51–63, New York, NY, USA, 2003. ACM Press.

[BDJ+03] Jean Bézivin, Grégoire Dupé, Frédéric Jouault, Gilles Pitette und Jamal Eddine Rougui. First Experiments with the ATL Model Transformation Language: Transforming XSLT into XQuery. In *Proceeding of the OOPSLA Workshop on Generative Techniques in the Context of MDA (OOPSLA'03)*, 2003.

[Bec03] Oliver Becker. Transforming XML on the Fly – How STX Enables the Processing of Large Documents. In *Proceedings of the XML Europe 2003 Conference*, May 2003.

[Bec04] Oliver Becker. *Serielle Transformationen von XML – Probleme, Methoden, Lösungen.* Dissertation, Humboldt-Universität zu Berlin, November 2004.

[Ben86] Jon Bentley. Programming Pearls: Little Languages. *Communications of the ACM*, 29(8):711–721, August 1986.

[Ben03] Ron Benson (Hrsg.). *Streaming API for XML (JSR-173) for Java ® Specification, Version 1.0.* Java Community Process, October 2003.

[Büh34] Karl Bühler. *Sprachtheorie. Die Darstellungsfunktion der Sprache.* Fischer Verlag, Jena, 1934.

[BHLT06] Tim Bray, Dave Hollander, Andrew Layman und Richard Tobin (Hrsg.). *Namespaces in XML (Second Edition), W3C Recommendation 16 August 2006.* W3C, 2006.

[BHW97] James M. Boyle, Terence J. Harmer und Victor L. Winter. The TAMPR Program Transformation System: Simplifying the Development of Numerical Software. In *Modern Software Tools for Scientific Computing*, Seiten 353–372, Cambridge, MA, USA, 1997. Birkhauser Boston Inc.

[BKVV05] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas und Eelco Visser. *Stratego/XT Reference Manual*, 2005.

[BM04] Paul V. Biron und Ashok Malhotra (Hrsg.). *XML Schema Part 2: Datatypes Second Edition, W3C Recommendation 28 October 2004.* W3C, 2004.

[Boe81] Barry W. Boehm. *Software Engineering Economics.* Prentice-Hall, 1981.

[Bol03] Erwin Bolwidt. Jaxup 1.01, Release Date 06.08.2003. http://www.klomp.org/jaxup/, 2003.

[BPS97] Tim Bray, Jean Paoli und C. M. Sperberg-McQueen (Hrsg.). *Extensible Markup Language (XML), W3C Proposed, 8 December 1997.* 1997.

[BPS+06a] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler und Françoir Yergeau (Hrsg.). *Extensible Markup Language (XML) Version 1.0 (Fourth Edition), W3C Recommendation, 16 August 2006, edited in place 29 September 2006.* 2006.

[BPS+06b] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, Françoir Yergeau und John Cowan (Hrsg.). *Extensible Markup Language (XML) Version 1.1 (Second Edition), W3C Recommendation, 16 August 2006, edited in place 29 September 2006.* 2006.

[Bra98] Tim Bray. An Introduction to XML Processing with Lark and Larval. http://www.textuality.com/Lark/, January 1998.

[Bro00] *Der Brockhaus von A-Z in drei Bänden.* Weltbild Verlag, F. A. Brockhaus GmbH, Mannheim, 2000.

[BS86] Gèrard Berry und Ravi Sethi. From Regular Expressions to Deterministic Automata. *Theoretical Computer Science*, 48(1):117–126, 1986.

[BS96] Tim Bray und C. M. Sperberg-McQueen (Hrsg.). *Extensible Markup Language (XML), W3C Working Draft, 14 November 1996.* 1996.

[BS02a] Alexandru Berlea und Helmut Seidl. Transforming XML Documents Using fxt. *Journal of Computing and Information Technology*, 10(1):19–35, 2002.

[BS02b] Claus Brabrand und Michael I. Schwartzbach. Growing Languages with Metamorphic Syntax Macros. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '02)*, Seiten 31–40, New York, NY, USA, 2002. ACM Press.

[bul07] buldocs Ltd. xnsdoc 1.3, Release Date 27.06.2007. http://www.buldocs.com/xnsdoc/, 2007.

[BV04] Martin Bravenboer und Eelco Visser. Concrete Syntax for Objects: Domain-Specific Language Embedding and Assimilation without Restrictions. In *Proceedings of the 19th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '04)*, Seiten 365–383, New York, NY, USA, 2004. ACM Press.

[Béz05] Jean Bézivin. On The Unification Power of Models. *Software and Systems Modeling*, 4(2):171–188, May 2005.

[Cas08] Castor 1.2, Release Date 04.02.2008. http://www.castor.org, 2008.

[CBN+07] Petr Cimprich, Oliver Becker, Christian Nentwich, Honza Jiroušek, Manos Batsis, Paul Brown und Michael Kay. *Streaming Transformations for XML (STX), Version 1.0, Working Draft 27 April 2007*, 2007.

[CBNW98] Mike Champion, Steve Byrne, Gavin Nicol und Lauren Woods (Hrsg.). *Document Object Model (DOM) Level 1, W3C Recommendation 01 October 1998.* W3C, 1998.

[CCH07] James R. Cordy, Ian H. Carmichael und Russell Halliday. *The TXL Programming Language, Version 10.5, November 2007.* Software Technology Laboratory School of Computing, Queen's University at Kingston, Kingston, Ontario K7L 3N6, Canada, 2007.

[CD99] James Clark und Steve DeRosa (Hrsg.). *XML Path Language (XPath), Version 1.0, W3C Recommendation 16 November 1999*. W3C, 1999.

[CDu08] CDuce 0.5.2, Release Date 25.02.2008. http://www.cduce.org/, 2008.

[CE00] Krzysztof Czarnecki und Ulrich W. Eisenecker. *Generative Programming – Methods, Tools, and Applications*. Addison-Wesley, Pearson Education, 2000.

[CEP99] Francisco Cubera, David A. Epstein und Terrence Poon. Efficient Encoding of XML Updates. In *Proceedings of the XML Developers Conference, August 19-20, 1999, Montréal, Canada*, 1999.

[CH03] Krzysztof Czarnecki und Simon Helsen. Classification of Model Transformation Approaches. In *Proceedings of the Workshop on Generative Techniques in the Context of Model-Driven Architecture (OOPSLA'03)*, October 2003.

[CH06] Krysztof Czarnecki und Simon Helsen. Feature-Based Survey of Model Transformation Approaches. *IBM System Journal*, 45(3):621 – 645, 2006.

[Chi95] Shigeru Chiba. A Metaobject Protocol for C++. *ACM SIGPLAN Notice*, 30(10):285–299, 1995.

[CKM04] Aske Simon Christensen, Christian Kirkegaard und Anders Møller. A Runtime System for XML Transformations in Java. In *Proceedings of the Second International XML Database Symposium on Database and XML Technologies (XSym'04), Toronto, Canada, August 29-30, 2004*, Band 3186 aus *Lecture Notes in Computer Science*, Seiten 143–157. Springer Verlag, 2004.

[CKR04] Ben Chang, Joe Kesselman und Rezaur Rahman (Hrsg.). *Document Object Model (DOM) Level 3 Validation Specification, Version 1.0, W3C Recommendation 27 January 2004*. W3C, 2004.

[Cla98] James Clark. XP – an XML Parser in Java, Version 0.5. http://www.jclark.com/xml/xp/index.html, 1998.

[Cla99a] James Clark (Hrsg.). *XSL Transformations (XSLT), Version 1.0, W3C Recommendation 16 November 1999*. W3C, 1999.

[Cla99b] James Clark. XT, Release Date 05.11.1999. http://www.jclark.com/xml/xt-old.html, 1999.

[Cle88] J. Craig Cleaveland. Building Application Generators. *IEEE Computer Society*, 5(4):25–33, Juli 1988.

[Com07] Computer Woche. CW-Ranking: Die Top-IT-Begriffe 2006, 30.01.2007. http://www.computerwoche.de/treffpunkt/cw-rankings/587188/index.html, Januar 2007.

[Cor06] James R. Cordy. The TXL Source Transformation Language. *Science of Computer Programming*, 61(3):190–210, 2006.

[CRF01] Donald D. Chamberlin, Jonathan Robie und Daniela Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. In *Proceeding of the Third International Workshop on World Wide Web and Databases (WebDB '00), Dallas, TX, USA, May 2000*, Band 1997, Seiten 1–25. Springer Verlag, 2001.

[CT04] John Cowan und Richard Tobin (Hrsg.). *XML Information Set (Second Edition), W3C Recommendation, 24 February 2004*. W3C, 2004.

[Dat08] DataDirect Technologies. Stylus Studio® 2008 XML Enterprise Suite, Release Date 05.01.2008. http://www.stylusstudio.com, 2008.

[Dec02] Decisionsoft. XML Script 2.0, X-Trac v2 (XPath) Release Date 10.10.2002. http://www.xmlscript.org, 2002.

[DFH+99] Andrew Davidson, Matthew Fuchs, Mette Hedin, Mudita Jain, Jari Koistinen, Chris Lloyd, Murray Maloney und Kelly Schwarzhof. *Schema for Object-Oriented XML 2.0, W3C Note 30 July 1999*. W3C, 1999.

[DFS04] Rémi Douence, Pascal Fradet und Mario Südholt. Composition, Reuse and Interaction Analysis of Stateful Aspects. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD'04)*, Seiten 141–150, New York, NY, USA, 2004. ACM Press.

[DHK96] Arie van Deursen, Jan Heering und Paul Klint. Language Prototyping: An Algebraic Specification Approach. Band 5 aus *Amast Series in Computing*. World Scientific Publishing Co, 1996.

[Dij02] Edsger W. Dijkstra. Go to Statement Considered Harmful. In *Software Pioneers: Contributions to Software Engineering*, Seiten 351–355, New York, NY, USA, 2002. Springer Verlag.

[DK97] Arie van Deursen und Paul Klint. Little Languages: Little Maintenance? In Samuel N. Kamin (Hrsg.), *Proceeding of the first ACM-SIGPLAN Workshop on Domain-Specific Languages (DSL'97)*, Seiten 109–127, January 1997.

[Dod01] Leigh Dodds. Schematron: Validating XML Using XSLT. http://www.ldodds.com/papers/schematron_xsltuk.html, April 2001.

[Dör94] Dietrich Dörner. *Problemlösen als Informationsverarbeitung*. Kohlhammer, 3. Auflage, 1994.

[DS84] L. Peter Deutsch und Allan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '84)*, Seiten 297–302, New York, NY, USA, 1984. ACM Press.

[DST02] DSTC Pty Ltd. xs3p Version 1.1.3. http://xml.fiforms.org/xs3p/, 2002.

[Dud07] *Duden*. Dudenverlag, Mannheim, Leipzig, Wien, Zürich, 6. Auflage, 2007.

[Dvo05] Jana Dvorakova. On Classification of XML Document Transformations. In Karel Richta, Václav Snásel und Jaroslav Pokorný (Hrsg.), *Proceedings of the Annual International Workshop on Databases, Texts, Specifications and Objects (DaTeSO'05), Desna, Czech Republic, April 13-15, 2005*, Seiten 69–83. CEUR-WS.org, 2005.

[eba08] ebay Inc. ebay Trading Web Service API Guide, Version 551, Release Date 06.02.2008. http://developer.ebay.com/developercenter/xml/, 2008.

[ecl08a] eclipse.org. Eclipse Modeling Framework Project (EMF), Version 2.4, Release Date 09.06.2008. http://www.eclipse.org/modeling/emf/, June 2008.

[ecl08b] eclipse.org. Eclipse Web Tools Platform (WTP) Project, Version 3.0, Release Date 16.06.2008. http://download.eclipse.org/webtools/, June 2008.

[Ehr79] Hartmut Ehrig. Introduction to the Algebraic Theory of Graph Grammars (A Survey). In *Proceedings of the International Workshop on Graph-Grammars and their Application to Computer Science and Biology*, Seiten 1–69, London, UK, 1979. Springer Verlag.

[Eng99] Dawson R. Engler. Interface Compilation: Steps Toward Compiling Program Interfaces as Languages. *IEEE Transactions on Software Engineering*, 25(3):387–400, 1999.

[eXi08] eXist, Open Source Native XML Database, Version 1.2, Release Date 14.05.2008. http://exist.sourceforge.net/, 2008.

[EXS08] EXSLT Initiative. http://www.exslt.org, 2008.

[FC04] Alain Frisch und Luca Cardelli. Greedy Regular Expression Matching. In Josep Díaz, Juhani Karhumäki, Arto Lepistö und Donald Sannella (Hrsg.), *Proceedings of the 31st International Colloquium on Automata, Languages and Programming (ICALP'04), Turku, Finland, July 12-16, 2004.*, Seiten 618–629. Springer Verlag, 2004.

[FF07] Sven Feja und Daniel Fötsch. Systemintegration durch Service-orientierte Architektur im elektronischen Handel und der Beschaffung. In Klaus-Peter Fähnrich, Maik Thränert und Peter Wetzel (Hrsg.), *Integration Engineering: Motivation – Begriffe – Methoden – Anwendungsfälle*, Band VI aus *Leipziger Beiträge zur Informatik*, Seiten 191–203. Eigenverlag Leipziger Informatik-Verbund (LIV), September 2007.

[FF08] Sven Feja und Daniel Fötsch. Model Checking with Graphical Validation Rules. In *Proceedings of the 15th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS'08), 31 March - 4 April 2008, Belfast, North Ireland*, Seiten 117–125. IEEE Computer Society, 2008.

[FFS06] Sven Feja, Daniel Fötsch und Jens Schenderlein. E-Procurement-Integration am Beispiel des ERP-Systems GODYO P/4 . In Klaus-Peter Fähnrich, Stefan Kühne,

Andreas Speck und Julia Wagner (Hrsg.), *Integration betrieblicher Informationssysteme: Problemanalysen und Lösungsansätze des Model-Driven Integration Engineering*, Band IV aus *Leipziger Beiträge zur Informatik*, Seiten 42–48. Eigenverlag Leipziger Informatik-Verbund (LIV), Leipzig, Germany, September 2006.

[FFS08] Sven Feja, Daniel Fötsch und Sebastian Stein. Grafische Validierungsregeln am Beispiel von EPKs. In *Beiträge zu den Workshops der Software Engineering 2008, Fachtagung des GI-Fachbereichs Softwaretechnik, 18.-22. Februar 2008 in München*, Band 122 aus *LNI*, 198–204, 2008. GI.

[FFSD06] Daniel Fötsch, Sven Feja, Sascha Sauer und Andreas David. Problemstellungen agiler Schnittstellen am Beispiel des Commerce Management Systems von Truition/AGETO. In Klaus-Peter Fähnrich, Stefan Kühne, Andreas Speck und Julia Wagner (Hrsg.), *Integration betrieblicher Informationssysteme: Problemanalysen und Lösungsansätze des Model-Driven Integration Engineering*, Band IV aus *Leipziger Beiträge zur Informatik*, Seiten 49–56. Eigenverlag Leipziger Informatik-Verbund (LIV), Leipzig, Deutschland, September 2006.

[FN05] Jean-Marie Favre und Tam Nguyen. Towards a Megamodel to Model Software Evolution Through Transformations. *Electronic Notes in Theoretical Computer Science*, 127(3):59–74, 2005.

[FNP97] Rickard E. Faith, Lars S. Nyland und Jan F. Prins. KHEPERA: A System for Rapid Implementation of Domain Specific Languages. In *Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL'97)*, Seite 19, Berkeley, CA, USA, 1997. USENIX Association.

[FP98] Norman E. Fenton und Shari Lawrence Pfleeger. *Software Metrics – A Rigorous and Practical Approach (Revised Printing)*. PWS Publishing Co., Boston, MA, USA, 2. Auflage, 1998.

[FP07a] Daniel Fötsch und Elke Pulvermüller. Applying the Operator Hierarchy Concept to XSLT. In Robert Tolksdorf und Johann-Christoph Freytag (Hrsg.), *XML-Tage 2007 in Berlin, 24.-26. September 2007, Berlin*, Seiten 51–62. XML-Clearinghouse, 2007.

[FP07b] Daniel Fötsch und Elke Pulvermüller. Constructing Higher-Level Transformation Languages Based on XML. In Hamido Fujita und Domenico Pisanelli (Hrsg.), *New Trends in Software Methodologies, Tools and Techniques – Proceeding of the 6th International Conference on Software Methodologies, Tools and Techniques (SoMET'07), 7-9 November 2007, Rome, Italy*, Band 161 aus *Frontiers in Artificial Intelligence and Applications*, Seiten 269–284, Rome, Italy, November 2007. IOS Press.

[FP08] Daniel Fötsch und Elke Pulvermüller. A Concept and Implementation of Higher-Level XML Transformation Languages. *Knowledge-Based Systems*, 2008. To appear.

[FPR06] Daniel Fötsch, Elke Pulvermüller und Wilhelm R. Rossak. Modeling and Verifying Workflow-Based Regulation. In *Proceedings of the International Workshop on Regulations Modelling and their Validation and Verification (REMO2V'06) in Conjunction with CAiSE'06*, Seiten 825–830, Luxembourg, Juni 2006.

[Fre95] Free Software Foundation. Flex, Version 2.5 – A fast scanner generator, Edition 2.5, March 1995. http://www.gnu.org/software/flex/manual/, March 1995.

[Fre06] Free Software Foundation. Bison Manual Formats. http://www.gnu.org/software/bison/manual/, August 2006.

[FS06] Daniel Fötsch und Andreas Speck. XTC – The XML Transformation Coordinator for XML Document Transformation Technologies. In *Proceedings of the 17th International Workshop on Database and Expert Systems Applications (DEXA'06), 4-8 September 2006, Krakow, Poland*, Seiten 507–511, Washington, DC, USA, 2006. IEEE Computer Society.

[FSH05] Daniel Fötsch, Andreas Speck und Peter Hänsgen. The Operator Hierarchy Concept for XML Document Transformation Technologies. In Rainer Eckstein und Robert Tolksdorf (Hrsg.), *Berliner XML-Tage 2005 (BXML'05), 12.-14. September 2005, Berlin*, Seiten 59–70. XML-Clearinghouse, September 2005.

[FSRK05] Daniel Fötsch, Andreas Speck, Wilhelm Rossak und Jörg Krumbiegel. A Concept for Modelling and Validation of Web Based Presentation Templates. In Otto K. Ferstl, Elmar J. Sinz, Sven Eckert und Tilman Isselhorst (Hrsg.), *Wirtschaftsinformatik 2005: eEconomy, eGovernment, eSociety, 7. Internationale Tagung Wirtschaftsinformatik 2005, Bamberg, 23.2.2005 - 25.2.2005 (WI'05)*, Seiten 391–406. Physica Verlag, 2005.

[Föt05] Daniel Fötsch. Eine XML-basierte Transformationsmethode für modellgetriebene Web-Service-Architekturen: Vom ARIS-IE-Geschäftsprozessmodell zum BPEL4WS-Prozess. In Klaus-Peter Fähnrich, Maik Trähnert und Peter Wetzel (Hrsg.), *Umsetzung von kooperativen Geschäftsprozessen auf eine internetbasierte IT-Struktur: Arbeiten aus dem Forschungsvorhaben Integration Engineering, Leipziger Beiträge zur Informatik*, Band III aus *Leipziger Beiträge zur Informatik*, Seiten 107–124. Eigenverlag Leipziger Informatik-Verbund (LIV), Leipzig, Deutschland, September 2005.

[Föt07a] Daniel Fötsch. A Generic Framework for Target-Specific Transformation Operators in XML Transformation Languages. In *Proceeding of the 18th International Workshop on Database and Expert Systems Applications (DEXA'07), 3-7 September 2007, Regensburg, Germany*, Washington, DC, USA, 2007. IEEE Computer Society.

[Föt07b] Daniel Fötsch. Anwendung der Operatorhierarchie zur Vereinfachung der Entwicklung von XML-Transformationen am Beispiel vom ARIS-IE-Geschäftsprozess zum BPEL-Prozess. In Klaus-Peter Fähnrich, Maik Thränert und Peter Wetzel

(Hrsg.), *Integration Engineering: Motivation – Begriffe – Methoden – Anwendungsfälle*, Band VI aus *Leipziger Beiträge zur Informatik*, Seiten 105–120. Eigenverlag Leipziger Informatik-Verbund (LIV), Leipzig, Deutschland, September 2007.

[Fun08] FunctX XSLT Function. http://www.xsltfunctions.com, 2008.

[FW93] An Feng und Toshiro Wakayama. SIMON: A Grammar-Based Transformation System for Structured Documents. *Electronic Publishing*, 6(4):361–372, 1993.

[FW04] David C. Fallside und Priscalla Walmsley (Hrsg.). *XML Schema Part 0: Primer Second Edition, W3C Recommendation 28 October 2004*. W3C, 2004.

[fxt05] fxt 4.6.1, Release Date 05.07.2005. http://www2.informatik.tu-muenchen.de/~berlea/Fxt/, Juli 2005.

[Ger06] German National Research Center for Information Technology, Fraunhofer Institute for Computer Architecture and Software Technology. The Catalog of Compiler Construction Tools. http://catalog.compilertools.net, 2006.

[GGP06] Vladimir Gapeyev, François Garillot und Benjamin C. Pierce. Statically Typed Document Transformation: An Xtatic Experience. In *Proceedings of the Workshop on Programming Language Technologies for XML (PLAN-X'06), Charleston, South Carolina, January 14, 2006*, Seiten 2–13. BRICS, Department of Computer Science, University of Aarhus, 2006.

[GHJV95] Erich Gamma, Richard Helm, Ralph E. Johnson und John Vlissides. *Design Pattern, Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Bosten, MA, 1995.

[GL05] Samuel Z. Guyer und Calvin Lin. Broadway: A Compiler for Exploiting the Domain-Specific Semantics of Software Libraries. *Proceedings of the IEEE, Special Issue on Program Generation, Optimization, and Platform Adaptation*, 93(2):342–357, 2005.

[GMNR05] Martin Gudgin, Noah Mendelsohn, Mark Nottingham und Hervé Ruellan (Hrsg.). *XML-binary Optimized Packaging, W3C Recommendation 25 January 2005*. W3C, 2005.

[Gol90] Charles F. Goldfarb. *The SGML Handbook*. Oxford University Press, Inc., New York, NY, USA, 1990.

[Gro08] Atlas Group. ATL 2.0.0, Release Date 10.06.2008. http://www.eclipse.org/m2m/atl/, 2008.

[Har06] Falk Hartmann. An Architecture for an XML-Template Engine Enabling Safe Authoring. In *Proceedings of the 17th International Workshop on Database and Expert Systems Applications (DEXA'06), 4-8 September 2006, Krakow, Poland*, Seiten 502–507, Washington, DC, USA, 2006. IEEE Computer Society.

[Hau06] Karsten Hauser. Klassifikation und Vergleich von XML-Transformationssprachen und Verbesserungsmöglichkeiten sowie -potentiale. Diplomarbeit, Friedrich-Schiller-Universität Jena, Fakultät für Mathematik und Informatik, Institut für Informatik, Lehrstuhl für Softwaretechnik, Ernst-Abbe-Platz 1-4, 07743 Jena, März 2006.

[HB88] Jr. Robert M. Herndon und Valdis A. Berzins. The Realizable Benefits of a Language Prototyping Language. *IEEE Transactions on Software Engineering*, 14(6):803–809, Juni 1988.

[HHW+04] Arnaud Le Hors, Philippe Le Hégaret, Lauren Wood, Gavin Nicol, Jonathan Robie, Mike Champion und Steve Byrne (Hrsg.). *Document Object Model (DOM) Level 3 Core Specification, Version 1.0, W3C Recommendation 07 April 2004*. W3C, 2004.

[HM04] Elliotte Rusty Harold und W. Scott Means. *XML in a Nutshell*. O'Reilly, 3. Auflage, September 2004.

[HM07] Jason Hunter und Brett McLaughlin. JDOM 1.1, Release Date 13.11.2007. http://jdom.org/, 2007.

[Hol00] G. Ken Holman. What is XSLT. http://www.xml.com/pub/a/2000/08/holman/ index.html, 16. August 2000.

[HP01] Haruo Hosoya und Benjamin Pierce. Regular Expression Pattern Matching for XML. *ACM SIGPLAN Notices*, 36(3):67–80, 2001.

[HP03] Haruo Hosoya und Benjamin C. Pierce. XDuce: A Statically Typed XML Processing Language. *ACM Transaction on Internet Technology (TOIT)*, 3(2):117–148, May 2003.

[HR04] David Harel und Bernhard Rumpe. Meaningful Modeling: What's the Semantics of „Semantics?“. *IEEE Computer*, 37(10):64–72, 2004.

[HRS+05] Matthew Harren, Mukund Raghavachari, Oded Shmueli, Michael G. Burke, Rajesh Bordawekar, Igor Pechtchanski und Vivek Sarkar. XJ: Facilitating XML Processing in Java. In *Proceedings of the 14th International Conference on World Wide Web (WWW'05), Chiba, Japan, May 10-14, 2005*, Seiten 278–287, New York, NY, USA, 2005. ACM Press.

[IE206] IE - Integration Engineering, BMBF-Projekt, 2004–2006. http://ie.informatik.uni-leipzig.de, 2006.

[Inc99] Adobe Systems Incorporated (Hrsg.). *PostScript® Language Reference Third Edition*. Addison-Wesley Publishing Company, 1999.

[Inn07] Innovasys Ltd. Document! X Version 5. http://www.innovasys.com/products/dx5/overview.aspx, 2007.

[Int86] International Organization for Standardization, ISO 8879:1986. *Information Processing – Text and Office Systems – Standard Generalized Markup Language (SGML)*, 1986.

[Int91] International Organization for Standardization, ISO/IEC 646:1991. *Information Technology – Document Schema Definition Languages (DSDL) – Part 2: Grammar-based Validation – RELAX NG*, 1991.

[Int93] International Organization for Standardization, ISO/IEC 10646-1993. *Information Technology – Universal Multiple-Octet Coded Character Set (UCS) – Part 1: Architecture and Basic Multilingual Plan*, 1993.

[Int96a] International Organization for Standardization, ISO/IEC 10179:1996. *Information Technology – Document Description and Processing Languages – Document Style Semantics and Specification Language (DSSSL)*, August 1996.

[Int96b] International Organization for Standardization, ISO/IEC 14977:1996(E). *Information Technology – Syntactic Metalanguage – Extended BNF*, 1996.

[Int00] International Organization for Standardization, ISO/IEC 10646-2000. *Information Technology – Universal Multiple-Octet Coded Character Set (UCS) – Part 1: Architecture and Basic Multilingual Plan*, 2000.

[Int03] International Organization for Standardization, ISO/IEC 19757-2. *Information Technology – Document Schema Definition Languages (DSDL) – Part 2: Grammar-based Validation – RELAX NG*, November 2003.

[Int06] International Organization for Standardization, ISO/IEC 19757-3:2006. *Information Technology – Document Schema Definition Language (DSDL) – Part 3: Rule-based validation – Schematron*, November 2006.

[IT-08] IT-Wissen – Das große Online-Lexikon für Informationstechnologie. Markup. http://www.itwissen.info/definition/lexikon//__markup_Markup.html, 2008.

[Jec00] Mario Jeckle. Konzepte der Metamodellierung – zum Begriff Metamodell. *Softwaretechnik Trends*, 20(2), Mai 2000.

[Jel99] Rick Jelliffe. Using XSL as a Validation Language. http://xml.ascc.net/en/utf-8/XSLvalidation.html, January 1999.

[Jel01] Rick Jelliffe. The Current State of the Art of Schema Languages. In *Proceedings of the XML Asian Pacific, Sydney, Australia*, 2001.

[JF88] Ralph E. Johnson und Brian Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2):22–35, Juli 1988.

[Joh75] Stephen C. Johnson. Yacc - Yet Another Compiler-Compiler. Technischer Bericht, AT&T Bell Laboratories, July 1975.

[Joo08] joost 1.1, Release Date 28.05.2008. http://joost.sourceforge.net/, 2008.

[Kam98] Samuel N. Kamin. Research on Domain-Specific Embedded Languages and Program Generators. Band 14 aus *Electronic Notes in Theoretical Computer Science*. Elsevier Press, 1998.

[Kay07] Michael Kay (Hrsg.). *XSL Transformations (XSLT), Version 2.0, W3C Recommendation 27 January 2007.* W3C, 2007.

[Kay08] Michael Kay. Saxon 9.1.0.1, Release Date 02.07.2008. http://saxon.sourceforge.net, 2008.

[Küh05] Thomas Kühne. What is a Model? In Jean Bezivin und Reiko Heckel (Hrsg.), *Proceedings of the Dagstuhl Seminar, Language Engineering for Model-Driven Software Development, Number 04101.* Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.

[KKF06] Heiko Kern, Stefan Kühne und Daniel Fötsch. Merkmale und Werkzeugunterstützung für Modelltransformationen im Kontext modellgetriebener Softwareentwicklung. In Klaus-Peter Fähnrich, Stefan Kühne, Andreas Speck und Julia Wagner (Hrsg.), *Integration betrieblicher Informationssysteme: Problemanalysen und Lösungsansätze des Model-Driven Integration Engineering*, Band IV aus *Leipziger Beiträge zur Informatik*, Seiten 94–104. Eigenverlag Leipziger Informatik-Verbund (LIV), Leipzig, Deutschland, September 2006.

[KL02] Martin Kempa und Volker Linnemann. On XML Objects. In *Proceedings of the Workshop on Programming Language Technologies for XML (PLAN-X'02)*, Seiten 44–54, Pittsburgh, USA, October 3-8, 2002. ACM Press.

[KMS04] Christian Kirkegaard, Anders Møller und Michael I. Schwartzbach. Static Analysis of XML Transformations in Java. *IEEE Transactions on Software Engineering*, 30(3):181–192, 2004.

[Knu84] Donald E. Knuth. *The TeXbook.* Addison-Wesley, 1984.

[KR77] Brian W. Kernighan und Dennis M. Ritchie. The M4 Macro Processor. Technischer Bericht, Bell Laboratories, Murray Hill, New Jersey 07974, 7 1977.

[KR78] Brian W. Kernighan und Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, Inc., 1978.

[KSLB03] Gabor Karsai, Janos Sztipanovits, Ákos Lédeczi und Ted Bapty. Model-Integrated Development of Embedded Software. *Proceedings of the IEEE*, 91(1):145–164, 2003.

[Kur05] Kurt Riede. xsddoc-1.0, Release Date 15.10.2005. http://xframe.sourceforge.net/xsddoc/, October 2005.

[KW87] Eugene E. Kohlbecker und Mitchell Wand. Macro-by-Example: Deriving Syntactic Transformations from their Specifications. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '87)*, Seiten 77–84, New York, NY, USA, 1987. ACM Press.

[LC00] Dongwon Lee und Wesley W. Chu. Comparative Analysis of Six XML Schema Languages. *ACM SIGMOD Record*, 29(3):76–87, 2000.

[Lea66] Burt M. Leavenworth. Syntax Macros and Extended Translation. *Communications of the ACM*, 9(11):790–793, 1966.

[Lei03] Paula Leinonen. Automating XML Document Structure Transformations. In *Proceedings of the 2003 ACM Symposium on Document Engineering (DocEng'03)*, Seiten 26–28, New York, NY, USA, 2003. ACM Press.

[LHBL06] Roberto Lopez-Herrejon, Don Batory und Christian Lengauer. A Disciplined Approach to Aspect Composition. In *Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'06)*, Seiten 68–77, New York, NY, USA, 2006. ACM Press.

[Lim92] Wayne C. Lim. A Cost Justification Model for Software Reuse. In *Proceedings of the Fifth Workshop on Institutionalizing Reuse*, University of Maine, Orono, 1992.

[LJM+98] Andrew Layman, Edward Jung, Eve Maler, Henry S. Thompson, Jean Paoli, John Tigue, Norbert H. Mikula und Steve De Rose. *XML-Data, W3C Note 05 Jan 1998*. W3C, 1998.

[LM00] Andreas Laux und Lars Martin. XML Update Language, Working Draft, 14 September 2000. http://xmldb-org.sourceforge.net/xupdate/xupdate-wd.html, 2000.

[LS75] Mike E. Lesk und Eric Schmidt. Lex - A Lexical Analyzer Generator. Technischer Bericht, AT&T Bell Laboratories, July 1975.

[Lud03] Jochen Ludewig. Models in Software Engineering- An Introduction. *Software and System Modeling*, 2(1):5–14, 2003.

[Mas02] Ishikawa Masayasu (Hrsg.). *XHTML^{TM} 1.0 in XML Schema, W3C Note 2 September 2002*. W3C, 2002.

[Meg97] David Megginson. AElfred, Version 1.2. http://saxon.sourceforge.net/aelfred.html, 1997.

[Meg98] David Megginson. SAX 1.0, Release Date 11.05.1998. http://www.saxproject.org, 1998.

[Meg02] David Megginson. AElfred, Version 7.0. http://saxon.sourceforge.net/aelfred.html, 2002.

[Meg04] David Megginson. SAX 2.0.2, Release Date 27.04.2004. http://www.saxproject.org, 2004.

[MHS05] Marjan Mernik, Jan Heering und Anthony M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys (CSUR)*, 37(4):316–344, 2005.

[MVG06] Tom Mens und Pieter Van Gorp. A Taxonomy of Model Transformation. In *Proceedings of the International Workshop on Graph and Model Transformation (GraMoT'05)*, Band 152 aus *Electronic Notes in Theoretical Computer Science*, Seiten 125–142, March 2006.

[MW93] Ruth Malan und Kevin Wentzel. Economics of Software Reuse Revisited. Technischer Report, HPL-93-31, Software Technology Laboratory, April 1993.

[MW08] Antoine Marot und Roel Wuyts. A DSL to Declare Aspect Execution Order. In *Proceedings of the 3rd Domain-Specific Aspect Languages Workshop (DSAL'08), AOSD*, April 2008.

[NBA04] István Nagy, Lodewijk Bergmans und Mehmet Aksit. Declarative Aspect Composition. In *Proceedings of the 2nd Software-Engineering Properties of Languages and Aspect Technologies Workshop*, 2004.

[NBA05] István Nagy, Lodewijk Bergmans und Mehmet Aksit. Composing Aspects at Shared Join Points. In *Proceedings of the Net. ObjectDays (NODe'05), Erfurt, Germany, September 20-22*, Band 69 aus *LNI*, Seiten 19–38. GI, 2005.

[Nei86] James M. Neighbors. The Draco Approach to Constructing Software from Reusable Components. In *Readings in Artificial Intelligence and Software Engineering*, Seiten 525–535, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[NJ97] Lloyed H. Nakatani und Mark A. Jones. Jargons and Infocentrism. In *Proceedings of the First ACM SIGPLAN Workshop on Domain-Specific Languages*, Seiten 59–74, 1997.

[Nor99] Francis Norton. Generating XSL for Schema Validation. http://xml.coverpages.org/generatingXslValidators.html, May 1999.

[oAW08] oAW 4.3, Release Date 05.05.2008. http://www.eclipse.org/gmt/oaw, 2008.

[Ogb05] Uche Ogbuji. XUpdate Update. http://www.oreillynet.com/onlamp/blog/2005/04/ xupdate_update.html, April 2005.

[OMG06] OMG. *Meta Object Facility (MOF) Core Specification, OMG Available Specification, Version 2.0, 01.01.06*, 2006.

[Ora06] Oracle. Oracle XML Developer's Kit 10g, Version 10.2.0.2 Production Release . http://www.oracle.com/technology/tech/xml/xdkhome.html, 2006.

[OrV08] OrViA - Orchestrierung und Valididierung integrierter Anwendungssysteme, BMBF-Projekt, 2006–2008. http://www.orvia.de, 2008.

[PAA+02] Steven Pemberton, Murray Altheim, Daniel Austin, Jonny Axelsson, Tantek Çelik, Doug Dominiak, Herman Elenbaas, Beth Epperson, Subramanian Peruvemba, Rob Relyea, Frank Boumphrey, John Burger, Andrew W. Donoho, Sam Dooley, Klaus Hofrichter, Philipp Hoschka, Masayasu Ishikawa, Warner ten Kate, Peter King, Paula Klante, Shin'ichi Matsui, Shane McCarron, Ann Navarro, Zach Nies,

Dave Raggett, Patrick Schmitz, Sebastian Schnitzenbaumer, Peter Stark, Chris Wilson, Ted Wugofski und Dan Zigmond. *XHTML™ 1.0 The Extensible HyperText Markup Language (Second Edition), A Reformulation of HTML 4 in XML 1.0, W3C Recommendation 26 January 2000, revised 1 August 2002*. W3C, 2002.

[Pan04] Tadeusz Pankowski. A High-Level Language for Specifying XML Data Transformations. In *Proceeding of the 8th East European Conference on Advances in Databases and Information Systems (ADBIS'04), Budapest, Hungary, September 22-25, 2004, Proceesing*, Band 3255 aus *Lecture Notes in Computer Science*, Seiten 159–172. Springer Verlag, 2004.

[Pen94] Volker Penner. *Konzepte und Praxis des Compilerbaus – Eine Einführung mit Diskette*. Vieweg Verlag, Braunschweig, Wiesbaden, 1994.

[PS02] Thomas Perst und Helmut Seidl. A Type-Safe Macro System for XML. In *Proceedings of the Extreme Markup Languages 2002 Montréal, Québec, August 6-9, 2002*, 2002.

[PS04] Elke Pulvermüller und Andreas Speck. XOpT - XML-Based Composition Concept. In Volker Gruhn Hamido Fujita (Hrsg.), *Proceedings of the 3rd International Conference on New Software Methodologies, Tools, and Techniques (SoMeT'04)*, Band 111, Seiten 249–262, Leipzig, Germany, September 2004. IOS Press.

[PZB00] Mikaël Peltier, Françoir Ziserman und Jean Bézivin. On Levels of Model Transformation. In *Proceedings of the XML Europe 2000*, Seiten 1–17, Paris, France, 2000.

[RHJ99] Dave Ragget, Arnaud Le Hors und Ian Jacobs (Hrsg.). *HTML 4.01 Specification, W3C Recommendation, 24 December 1999*. 1999.

[RJ96] Don Roberts und Ralph E. Johnson. Evolve Frameworks into Domain-Specific Languages. In *Proceedings of the 3rd International Conference on Pattern Language for Programming (POPL'96), Allerton Park, Illinois, USA, September 4-6*, Allerton Park, Illinois, September 1996.

[Rog06] Roger L. Costello. Best Practice: Use Multiple Schema Languages. http://www.xfront.com/Integrated-schema-approach/Use-Multiple-Schema-Languages.html, November 2006.

[RSB04] Martin Rinard, Alexandru Salcianu und Suhabe Bugrara. A Classification System and Analysis for Aspect-Oriented Programs. *SIGSOFT Software Engineering Notes*, 29(6):147–158, 2004.

[Rud08] Leonid Rudy. DocFlex/XML Version 1.7.0, Release Date 14.04.2008. http://www.filigris.com/downloads/, April 2008.

[SBC+01] Sharon Sadler, Anders Berglund, Jeff Caruso, Stephen Deach, Tony Graham, Eduard Gutentag, Alex Milowski, Scott Parnall, Jeremy Richman und Steve Zilles (Hrsg.). *Extensible Stylesheet Language (XSL), Version 1.0, W3C Recommendation 15 October 2001*. W3C, 2001.

[Sch01] skeleton1-5.xsl - An Implementation of Schematron 1.5 in XSLT, Release Date 12.06.2001. http://xml.ascc.net/schematron/1.5/, 2001.

[Sch07] Hannes Schrödter. Generierung von höheren XSLT-Operatoren. Diplomarbeit, Friedrich-Schiller-Universität Jena, Fakultät für Mathematik und Informatik, Institut für Informatik, Lehrstuhl für Softwaretechnik, Ernst-Abbe-Platz 1-4, 07743 Jena, Februar 2007.

[Sco07] Scott Boag and Don Chamberline and Mary Fernández and Daniela Florescu and Jonathan Robie and Jérôme Siméon (Hrsg.). *XQuery 1.0: An XML Query Language, W3C Recommendation 23 January 2007*. W3C, 2007.

[SG97] James M. Stichnoth und Thomas Gross. Code Composition as an Implementation Language for Compilers. In *Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL'97)*, Seiten 119–132, Berkeley, CA, USA, 1997. USENIX Association.

[SH04] Johnny Stenback und Andy Heninger (Hrsg.). *Document Object Model (DOM) Level 3 Load and Save Specification, Version 1.0, W3C Recommendation 07 April 2004*. W3C, 2004.

[Sha05] Yakov Shafranovich (Hrsg.). *RFC 4180 – Common Format and MIME Type for Comma-Separated Values (CSV) Files*. The International Society, October 2005.

[Sil01] David Silverlight. Shedding a Little Light on XML. http://xml.sys-con.com/read/40292.htm, December 2001.

[Slo02] Anthony Sloane. Post-Design Domain-Specific Language Embedding: A Case Study in the Software Engineering Domain. In *Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02)*, Seite 281, Washington, DC, USA, 2002. IEEE Computer Society.

[Slo05] Aleksander Slominski. XMLPullAPI, Release Date 23.10.2006. http://www.xmlpull.org, September 2005.

[Sne98] Harry M. Sneed. Metriken für die Wiederverwendbarkeit bestehender Software-Systeme. *Softwaretechnik-Trends*, 20(4), November 1998.

[SPFF06] Andreas Speck, Elke Pulvermüller, Daniel Fötsch und Sven Feja. Validierungstechniken für Software. In Klaus-Peter Fähnrich, Stefan Kühne, Andreas Speck und Julia Wagner (Hrsg.), *Integration betrieblicher Informationssysteme: Problemanalysen und Lösungsansätze des Model-Driven Integration Engineering*, Band IV aus *Leipziger Beiträge zur Informatik*, Seiten 105–122. Eigenverlag Leipziger Informatik-Verbund (LIV), Leipzig, Deutschland, September 2006.

[Sta77] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer Verlag, 1977.

[Ste93] Wilhelm Steinmüller. *Informationstechnologie und Gesellschaft: Einführung in die angewandte Informatik*. Wissenschaftliche Buchgesellschaft, Wiesbaden, 1993.

[Str98] Susanne Strahringer. Ein sprachbasierter Metamodellbegriff und seine Verallgemeinerung durch das Konzept des Metaisierungsprinzips. In Klaus Pohl, Andy Schürr und Gottfried Vossen (Hrsg.), *Proceedings des GI-Workshops in Münster (Modellierung'98), 11.-13. März 1998*, Band 9 aus *CEUR Workshop Proceedings*. CEUR-WS.org, 1998.

[Str05] James Strachan. DOM4J, Version 1.6.1, Release Date 16.05.2005. http://www.dom4j.org/, 2005.

[Sun03a] Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, CA 95054, USA. *JavaServer Pages™ Specification, Version 2.0*, November 2003.

[Sun03b] Sun Microsystems, Inc. Java™ 2 SDK, Standard Edition Documentation, Version 1.4.2. http://java.sun.com/j2se/1.4.2/docs/, 2003.

[Sun06] Sun Microsystems, Inc. JDK™ 6 Documentation. http://java.sun.com/javase/6/docs/, 2006.

[Sun07] Sun Microsystems, Inc. Java™ API For XML Processing (JAXP). https://jaxp.dev.java.net/, 2007.

[Swe85] Richard E. Sweet. The Mesa Programming Environment. In *Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments*, Seiten 216–229, New York, NY, USA, 1985. ACM Press.

[TBMM04] Henry S. Thompson, David Beech, Murray Maloney und Noah Mendelsohn (Hrsg.). *XML Schema Part 1: Structures Second Edition, W3C Recommendation 28 October 2004*. W3C, 2004.

[Tha03] Thai Open Source Software Center Ltd. Trang – Multi-Format Schema Converter Based on RELAX NG, Release Date 19.06.2003. http://www.thaiopensource.com/relaxng/trang.html, 2003.

[Tho00] Henry S. Thompson. Internet-Based Application Architectures for the 21st Century: The Role of XML (Keynote Talk). In *Proceedings of the Natural Computing Applications Forum (NCAF'00), January 20-21*, 2000.

[Tru08] Truition GmbH. Homepage. http://www.truition.de, 2008.

[TT02] Xuerxong Tang und Frank W. Tompa. A High-Level Specification Language for Structured Document Transformation. Technischer Bericht, CS-2002-42, UW School of Computer Science, University of Waterloo, 2002.

[Uni96] The Unicode Consortium, Reading, Mass.: Addison Wesley Developers Press. *The Unicode Standard, Version 2.0*, 1996.

[Uni00] The Unicode Consortium, Reading, Mass.: Addison Wesley Developers Press. *The Unicode Standard, Version 3.0*, 2000.

[Vel95] Todd Veldhuizen. Using C++ Template Metaprograms. *C++ Report*, 7(4):36–43, May 1995.

[Vel05] Todd Veldhuizen. Blitz++ User's Guide – A C++ Class Library for Scientific Computing for Version 0.9, 11.10. 2005. http://www.oonumerics.org/blitz/manual/blitz.ps, 2005.

[VF06] Sekhar Vajjhala und Joseph Fialli (Hrsg.). *The Java® Architecture for XML Binding (JAXB) 2.0, Final Release, April 19, 2006*. Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, CA 95054 USA, 2006.

[Vis04] Eelco Visser. Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in StrategoXT-0.9. In Christian Lengauer et al. (Hrsg.), *Domain-Specific Program Generation*, Band 3016 aus *Lecture Notes in Computer Science*, Seiten 216–238. Spinger Verlag, June 2004.

[Wal07] Malcom Wallace. Haskell and XML, HaXml-1.13.3, Release Date 23.11.2007. http://www.cs.york.ac.uk/fp/HaXml/, 2007.

[Wil99] Edward Wiles. Economic model of Software Reuse: A Survey, Comparison and Partial Validation, Version 2.1, Release Date 29.04.1999. Technischer Bericht, UWA-DCS-99-032, Department of Computer Science, University of Wales, Aberystwyth, Ceredigion SY23 3DB, U.K., 1999.

[Wil03] Stephen D. Williams. Efficiency structured XML (esXML): XML Without Most Processing Overhead – Thousands of Messages per Second with a more Straightforward, jet Sophisticated Application Coding. In *Proceedings of the Extreme Markup Languages 2003*, Montreal, Canada, August 2003.

[WM06] Norman Walsh und Leonard Mueller. *DocBook: The Definitive Guide*. O'Reilly & Associates, Inc., 2. Auflage, 2006.

[X-H07] X-Hive Corperation. X-Hive/DB Version 8. http://www.x-hive.com/products/db/index.html, 2007.

[XDu05] XDuce 0.5.0, Release Date 06.04.2005. http://sourceforge.net/projects/xduce/, 2005.

[XIn07] Apache XIndice, Version 1.1. http://xml.apache.org/xindice/, 2007.

[xme08] xmerl Application, Version 1.1.9. http://www.erlang.org/doc/apps/xmerl/, 2008.

[Zus91] Horst Zuse. *Software Complexity – Measures and Methods*. de Gryter Verlag, Berlin, 1991.

[ZWG+03] Aoying Zhou, Qing Wang, Zhimao Guo, Xueqing Gong, Shihui Zheng, Hongwei Wu, Jianchang Xiao, Kun Yue und Wenfei Fan. TREX: DTD-Conforming XML to XML Transformations. In Alon Y. Halevy, Zachary G. Ives und AnHai Doan (Hrsg.), *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, Seite 670, New York, NY, USA, 2003. ACM Press.

Abkürzungsverzeichnis

ACM	*Association for Computing Machinery*
AML	*ARIS Markup Language*
API	*Application Programming Interface*
ARIS	*Architektur integrierter Informationssysteme*
ASCII	*American Standard Code for Information Interchange*
ASF	*Algebraic Specification Formalism*
ASP	*Application Service Provider*
BNF	Backus-Naur-Form
BMBF	Bundesministerium für Bildung und Forschung
BPEL	*Business Process Execution Language*
BPEL4WS	*Business Process Execution Language for Web Services*
bspw.	beispielsweise
bzgl.	bezüglich
bzw.	beziehungsweise
CLI	*Command Line Interface*
CMS	*Commerce Management System*
CSS	*Cascading Stylesheet*
CSV	*Comma-Separated Values*
CTL	*Computational Tree Logic*
d. h.	das heißt
DOM	*Document Object Model*
DOM4J	*DOM for Java*
DSL	*Domain Specific Language*
DSTL	*Domain Specific Transformation Language*
DSSSL	*Document Style Semantics and Specification Language*
DSV	*Delimiter-Separated Values*
DTD	*Document Type Definition*
DTP	*Desktop Publishing*
EBNF	erweiterte Backus-Naur-Form
EDV	elektronische Datenverarbeitung
EMF	*Eclipse Modeling Framework*
EPK	ereignisgesteuerte Prozesskette

et al.	und andere
etc.	et cetera
evtl.	eventuell
fxt	*Functional XML Transformer*
GCTL	*Graphical Computational Temporal Logic*
GI	Gesellschaft für Informatik
GNU	*GNU is not Unix*
GML	*Generalized Markup Language*
Hrsg.	Herausgeber
HTML	*Hyper Text Markup Language*
im Allgemeinen	im Allgemeinen
IDL	*Interface Definition Language*
i. d. R.	in der Regel
IE	*Integration Engineering*
IEC	*International Electrotechnical Commission*
IEEE	*Institut of Electrical and Electronics Engineers*
Inc.	*Incorporated*
Infoset	*XML Information Set*
ISML	*Intershop Markup Language*
ISO	*International Organization for Standardization*
IT	Informationstechnik
J2SE	*Java 2 Plattform, Standard Edition*
JAXB	*Java Architecture for XML Binding*
JAXP	*Java API for XML Processing*
Jaxup	*Java XML Update*
JDK	*Java Developer Kit*
JIT	*Just In Time*
JSP	*Java Server Pages*
JSR	*Java Specification Request*
LNI	*Lecture Notes in Informatics*
Ltd.	*Limited*
LHS	*left hand side*
mglw.	möglicherweise
MIME	*Multipurpose Internet Mail Extensions*
MOF	*Meta Object Facility*
OMG	*Object Management Group*
OrViA	Orchestrierung und Validierung integrierter Anwendungssysteme
PDF	*Portable Document Format*
PSVI	*Post Schema Validation Infoset*
RFC	*Request for Comments*
RHS	*right hand side*
ROI	*return on investment*
RTF	*Rich Text Format*

SAX *Simple API for XML*
SDF *Syntax Definition Formalism*
SDK *Software Developer Kit*
SGML *Standard Generalized Markup Language*
SMIL *Synchronized Multimedia Integration Language*
SMV *Symbolic Model Verifier*
SQL *Structured Query Language*
StAX *Streaming API for XML*
STX *Streaming Transformations for XML*
SVG *Scalable Vector Graphics*
SWUL *Swing User-interface Language*
TrAX *Transformation API for XML*
TREX *Transformation Engine for XML*
TXL *Turing Extender Language*
u. a. unter anderem
UML *Unified Modeling Language*
URI *Uniform Resource Identifiers*
usw. und so weiter
u. U. unter Umständen
vgl. vergleiche
XHTML *Extensible HyperText Markup Language*
XML *Extensible Markup Language*
XOpGen *XML Operator Generator*
XOpT *XML Operator Transformer*
XPath *XML Path Language*
XP *XML Parser*
XPP *XML Pull Parsing*
XSL *Extensible Stylesheet Language*
XSLFO *XSL Formatting Objects*
XSLT *XSL Transformations*
XT *XSLT Transformations*
XTC *XML Transformation Coordinator*
XUL *XML Update Language*
XUpdate *XML Update Language*
W3C *World Wide Web Consortium*
WSDL *Web Service Definition Language*
WTP *Web Tool Platform*
WWW *World Wide Web*
XDK *XML Developer Kit*
XTC *XML Transformation Coordinator*
z. B. zum Beispiel

Index

Änderungssprachen, 52, 55, 70

Abstraktionsgrad, 53
Alphabet, 25
Analyse
  lexikalisch, 41
  syntaktische, 42
Anforderungen, 77
Anwendungsszenario
  Darstellungsformaterzeugung, 73
  Umstrukturierung, 70
Assemblersprachen, 84
Attribut, 15

Bereichsfestlegung, 52
Bibliothek, 60
Black-box-Wiederverwendung, 197
Bottom-up-Vorgehensmodell, 151

CDATA-Abschnitt, 16

datenzentrierte, 17
Document Object Model, *siehe* DOM
dokumentzentriert, 17
Dokumentknoten, 23
DOM, 39
Domänenspezifische Sprache, *siehe* DSL
DSL, 34, 47
  extern, 63
  intern, 59, 63
DSTL
  Anforderungen, 143
  Referenzszenario, 144
  Syntaxdefinition, 143

DTD, 13, 19
  Attributlisten, 19
  Elementtypen, 19
  Entity, 19
  Notation, 19

Ebenentransformation, 110
Ebenentransformationsdefinition, 80
Ebenentransformationsprozess
  Algorithmus, 95
Ebenenvalidierer, 196
Element, 15
elementare Transformation, 109
Elementare Operatoren, 85
Entity, 16
Entity-Referenz, 16
Ersetzungsmuster, 56
Extended Markup Language, *siehe* XML

Fallbeispiele
  Liste, 184
  XML2DSV, *siehe* XML2DSV
Framework, 60

GML, 13

Hook, 131

Infoset, *siehe* XML Information Set
Inputvalidierer, 196

Kommentar, 15
Konfliktlösungsmechanismen, 50
Kontrollstrukturen, 51
Kosten, 200

Kosten/Nutzen-Relation, 201

lexikalische Analyse, 41
lexikalische Verarbeitung, 35
lexikalischer Bereich, 129

Mapping-Sprachen, 58
Markup, 13
  deskriptiv, 13
  generisch, 13
  prozedural, 13
Markup-Sprache, 13
Maschinensprachen, 84
Merkmal, 44
Merkmaldiagramme, 44
Metamodell, 29
  linguistisch, 29
  ontologisch, 29
Metasprache, 26
Modell, 28
  -merkmale, 28
  Abbildungsmerkmal, 28
  pragmatisches Merkmal, 28
  Verkürzungsmerkmal, 28
Module, 51
Multidirektionalität, 49, 58

Namensraum, 17
  -präfix, 18
Nutzen
  Fehlererkennung, 195
  Flexibilität, 198
  Kompaktheit, 193
  Portabilität und Agilität, 199
  Prüfbarkeit, 194
  Verständlichkeit, 191
  Wiederverwendbarkeit, 197, 198

Objektsprache, 26
Operator, 84
  domänenspezifisch, 87, 176
  Semantik, 84, 88
  Syntax, 84
  universell, 86, 164

Parametrisierung, 49
Parser, 37, 43, 60
Pivot, 84
Plattform, 146
Probierprozess, 195
Programmbibliothek, 3
Programmiersprachen
  höhere, 84
Pull-Modell, 37
Pull-Technik, 56, 58
Push-Modell, 37
Push-Technik, 55, 58

qualifizierter Name, 18

Regelausführung
  Auswahl, 50
  Kontrollstrukturen
    Iteration, 51
    Verzweigung, 51
  Traversierung, 49
Regelorganisation
  Module, 51
  organisatorische Struktur, 52
  Wiederverwendung, 52
Return-On-Investment, 201

SAX, 37
Scanner, 42
Schemasprachen, 18, 20
  grammatikbasierte, 125
  regelbasierte, 125
Schematron, 21
semistrukturiert, 17
Serialisierung, 59
SGML, 13
Sprachdefinition
  Formen, 26
Sprache
  formal, 25
  Hierarchie, 27
  Meta, 26
  Objekt, 26
  Semantik, 25

Syntax, 25
abstrakt, 25
konkret, 25
Sprachhierarchie, 84
Sprachkomponente, 90
Sprachkonstrukte
elementar, 3
StAX, 37
Suchmuster, 56
syntaktische Analyse, 42

Templates, 57
Top-down-Vorgehensmodell, 139
Transformation
endogen, 47
exogen, 47
Transformationsdefinition, 43
Regelausführung
Auswahl, 50
Kontrollstrukturen, 51
Traversierung, 49
Regelorganisation
Module, 51
organisatorische Struktur, 52
Wiederverwendung, 52
Transformationsregel
Logik, 47
Modus, 47
Muster, 47
Variable, 47
Transformationsrichtung, 53
Transformationsmethode
quellstrukturgetrieben, 55
zielstrukturgetrieben, 56
Transformationsregel, 43, 45
höherer Ordnung, 49
Multidirektionalität, 49
Parametrisierung, 49
Transformationsrichtung, 53
Traversierung, 49, 188
TrAX, 106
Typisierung, 47

Umstrukturierung, 70
URI, 17

Validierer, 122
Validierung, 122
Verarbeitungsanweisung, 16
Verarbeitungsmodell
baumbasiert, 39
Bewertung, 40
schemaspezifisches Modell, 40
universelles Modell, 39
datenstrombasiert, 37
Bewertung, 38
Datenstruktur, 38
Pull-Modell, 37
Push-Modell, 37
Serialisierung, 38
lexikalisch, 35
Bewertung, 36
Vorgabenamensraum, 18
Vorgehensmodell
bottom-up, 151
Rollenverteilung, 156
top-down, 139

Weiterentwicklung, 4
Wertebereich, 129
White-box-Wiederverwendung, 197
Wiederverwendung, 52
black-box, 197
white-box, 197

XML, 11, 14
-Attribut, 15
-CDATA-Abschnitt, 16
-Daten, 33
-Deklaration, 15
-Dokument, 17
-Element, 15
-Entity-Referenz, 16
-Fragment, 17
-Kommentar, 15
-Namensraum, 17
-Namensraumpräfix, 18
-Text, 33

-Verarbeitungsanweisung, 16
-Vorgabenamensraum, 18
-Wurzelelement, 15
-Zeichenreferenz, 16
gültig, 17
Historie, 12
wohlgeformt, 17
XML Information Set, 23
Eigenschaft, 23
Informationseinheit, 23
XML Schema, 20
Nachteile, 20
Vorteile, 20
XML Transformation Coordinator, *siehe* XTC
XML2DSV
Definition, 181
Implementierung, 184
Motivation, 177
XOpGen, 81, 130
Code-Generatoren, 134
Ebenentransformator, 124, 138
grammatikbasiertes Schema, 126
Infrastrukturkomponente, 136
regelbasiertes Schema, 127
Schema-Generatoren, 131
XSLT, 73
Erweiterung, 163
XTC, 80, 103
Core System, 103
Level Library, 104
XUpdate, 70

Zeichenreferenz, 16