Spezifikation von interoperablen Webservices mit XQuery

Vom Promotionsausschuss der
Technischen Universität Hamburg-Harburg
zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)
genehmigte Dissertation

von

Marcus Venzke

aus Hamburg

2003

1. Gutachter: Prof. Dr. F. H. Vogt

2. Gutachter: Prof. Dr. U. Killat

Tag der mündlichen Prüfung: 1. Dezember 2003

Abstrakt

Die Arbeit löst Interoperabilitätsprobleme von Webservices. Der W3C-Standard zur Spezifikation ihrer Schnittstellen (WSDL) wird präzisiert. Dazu wird ein Spezifikationsverfahren (SXQT) entwickelt, das erlaubt, Schnittstellen mit prädikatenlogischen Ausdrücken (in XQuery) zu spezifizieren. Die Einhaltung solcher Spezifikationen wird mit der automatischen Validation geprüft.

Abstract

The thesis solves interoperability issues regarding Web Services. The W3C standard for the specification of their interfaces (WSDL) is made more precise. This is done by developing a specification technique (SXQT), which allows specifying interfaces using expressions of first order logic (in XQuery). Compliance with such specifications is checked with the automatic validation.

Zusammenfassung

Diese Arbeit löst Interoperabilitätsprobleme zwischen Webkomponenten. Solche Komponenten haben mit der Web Services Description Language (WSDL) spezifizierte Schnittstellen, über die nach dem Protokoll SOAP über das Internet kommuniziert wird. Interoperabilitätsprobleme können durch ungenau spezifizierte Schnittstellen sowie nicht der Spezifikation entsprechende Webkomponenten entstehen.

Anhand von Beispielen wird gezeigt, dass sich mit WSDL wesentliche Anforderungen an Schnittstellen nicht spezifizieren lassen. Nicht spezifizierbar sind z. B. Abhängigkeiten zwischen unterschiedlichen Nachrichten. Sie nicht zu beachten, kann zu Interoperabilitätsproblemen führen. Daher wird in der Arbeit ein Spezifikationsverfahren entwickelt, mit dem sie ausgedrückt werden können.

Zugrunde liegt ein Modell, das die Kommunikation über eine Schnittstelle als Folge von Ereignissen abstrahiert. Die Beobachtung einer SOAP-Nachricht wird als Ereignis angesehen, das ihren Inhalt als Parameter hat. Ein Trace ist eine endliche Folge von Ereignissen, wie sie ein Beobachter in der Reihenfolge der Beobachtung aufzeichnen könnte. Durch das Ausschließen von Gleichzeitigkeit führen auch nebenläufige Prozesse zu sequentiellen Traces.

Die Semantik der Schnittstelle schränkt die Menge möglicher Traces ein. Das in der Arbeit entwickelte Spezifikationsverfahren SXQT (Specifications using XQuery expressions on Traces) beschreibt sie mit prädikatenlogischen Ausdrücken (SXQT-Ausdrücke), notiert in der vom W3C standardisierten Sprache XQuery. Jede Anforderung wird einzeln als SXQT-Ausdruck formuliert, der entscheidet, ob ein beobachteter Trace der Anforderung entspricht. SXQT-Ausdrücke werden WSDL-Dokumenten konform zum WSDL-Standard zugefügt.

Die in der Arbeit entwickelte automatische Validation prüft, ob die Kommunikation über eine Schnittstelle der SXQT-Spezifikation entspricht. Ein Validator realisiert den Beobachter aus dem Modell von SXQT, zeichnet also den Trace auf. Zusätzlich prüft er, ob der Trace der SXQT-Spezifikation entspricht. Das erlaubt die Spezifikationskonformität von Webkomponenten beim Testen oder im produktiven Betrieb zu überwachen.

Inhaltsverzeichnis

1 Einleitung ... 1
1.1 Motivation ... 1
1.2 Zielsetzung ... 2
1.3 Struktur der Arbeit ... 2
2 Webkomponenten ... 5
2.1 Webkomponenten sind Komponenten ... 5
2.1.1 Komponenten ... 5
2.1.2 Schnittstellen ... 6
2.1.3 Komponentenarchitekturen ... 8
2.2 Unterschiede zu traditionellen Komponenten ... 10
2.3 Traditionelle Komponenten als Webkomponenten ... 11
2.3.1 Brücken für Interoperabilität von Komponentenarchitekturen ... 11
2.3.2 Grenzen von Brücken ... 12
2.3.3 Model Driven Architecture ... 15
2.3.4 Schlussfolgerung ... 16
2.4 Architektur für Webkomponenten ... 16
2.4.1 Webanwendungen als Webkomponenten ... 18
2.4.2 XML statt HTML ... 19
2.4.3 Wozu wird SOAP benötigt? ... 21
2.5 Spezifikation von Webkomponenten ... 22
2.6 Beispiel für eine Webkomponente ... 24
2.7 Stabilität von Softwaresystemen aus Webkomponenten ... 26
3 SOAP ... 29
3.1 Grundbegriffe ... 29
3.2 Nachrichtenstruktur ... 30
3.3 SOAP-Processing-Model ... 31
3.4 SOAP-Fehlernachrichten ... 33
3.5 Kodierung anwendungsspezifischer Teile von SOAP-Nachrichten ... 35
3.6 SOAP-Datenmodell ... 37
3.7 SOAP-Encoding ... 37
3.8 Kommunikationsmuster ... 40
3.9 Konventionen für einheitliche SOAP-Nachrichten ... 42
3.10 Protokollbindungen ... 44

4 Spezifikation von Webservices mit WSDL 49
4.1 Überblick 49
4.2 XML-Schema 50
4.2.1 Ziele von XML-Schema 50
4.2.2 Instanz- und Schemadokumente 51
4.2.3 XML-Elemente und Attribute 52
4.2.4 Vordefinierte Typen 53
4.2.5 Ableiten einfacher Typen 55
4.2.6 Definition komplexer Typen 57
4.2.7 Ableiten komplexer Typen 61
4.3 Web Services Description Language (WSDL) 63
4.3.1 Grundbegriffe 63
4.3.2 WSDL-Dokumente 64
4.3.3 Typen 66
4.3.4 Nachrichten 67
4.3.5 Abstrakte Schnittstellen 68
4.3.6 Gebundene Schnittstellen 70
4.3.7 SOAP-Bindung 71
4.3.8 Webservice 75
5 Präzisierungsstandard für Interoperabilität 77
5.1 Grundidee 77
5.1.1 Breite Standards 77
5.1.2 Präzisierungsstandards 79
5.1.3 Präzisierungsstandard für SOAP und WSDL 80
5.2 Präzisierung von SOAP 80
5.2.1 Kommunikationsmuster 80
5.2.2 Encodingstyle-Attribut 81
5.2.3 Verwendung von xsi:type 82
5.2.4 Repräsentation von Operationen 82
5.2.5 Protokollbindungen 84
5.2.6 SOAP-Intermediaries 85
5.3 Präzisierung von WSDL 86
5.3.1 Verwendung von SOAP 86
5.3.2 Typsystem 87
5.3.3 encoded versus literal 87
5.3.4 rpc versus document 90
5.3.5 Type- versus Element-Attribut 92
5.3.6 Verwendung von wsdl:import 93
5.4 Zusammenfassung 95
6 Ausdrucksfähigkeit von WSDL 97
6.1 Was kann nicht in WSDL ausgedrückt werden? 97
6.1.1 Anforderungen an einzelne SOAP-Nachrichten 98
6.1.2 Anforderungen an einzelne Request-/Response-Paare 100
6.1.3 Anforderungen an mehrere Operationsaufrufe 101
6.2 Grenzfälle 104
6.2.1 Optionale Parameter 104

6.2.2 Alternative Struktur ... 105
6.2.3 Dynamische Aufzählungstypen ... 107
6.2.4 Fehlernachrichten ... 109
6.2.5 Headereinträge ... 112

6.3 Fazit ... 116

7 Spezifikation mit XQuery-Ausdrücken über Traces ... 117

7.1 Spezifikation von Entitäten ... 117
7.1.1 Entitäten und Ereignisse ... 118
7.1.2 Prozessterme ... 119
7.1.3 Traces ... 120
7.1.4 Spezifikationen ... 121

7.2 Übertragung auf Webservices ... 123
7.2.1 Entitäten, Ereignisse ... 123
7.2.2 Traces ... 125
7.2.3 Beschreibung von Traces mit XML-Schema ... 128
7.2.4 Sichten des Beobachters ... 129

7.3 Spezifikationen von Webservices ... 130
7.3.1 Logische Aussagen und WSDL ... 131
7.3.2 Übertragen von Gesetzen für Prozessterme ... 132
7.3.3 Erfüllen einer Spezifikation ... 133
7.3.4 Spezifikationen und Sichten ... 134
7.3.5 Spezifikationen und Startzustand ... 134
7.3.6 Notationen für logische Ausdrücke ... 136
7.3.7 Sprachen des W3C für Ausdrücke ... 137

7.4 XQuery ... 138
7.4.1 Typsystem ... 138
7.4.2 Beschreibung von Typen ... 140
7.4.3 Literale ... 141
7.4.4 Beispiele für Operationen und Funktionen ... 142
7.4.5 Eingabefunktionen ... 144
7.4.6 Pfadausdrücke ... 144
7.4.7 Bedingte Ausdrücke ... 146
7.4.8 FLWOR-Ausdrücke ... 146
7.4.9 Quantoren ... 147
7.4.10 Query-Prolog ... 148
7.4.11 Fehlerbehandlung ... 149

7.5 XQuery für logische Aussagen von Spezifikationen ... 150
7.5.1 Grundidee ... 150
7.5.2 Sequenzen in XQuery und bei Hoare ... 150
7.5.3 Traces in SXQT-Ausdrücken ... 152
7.5.4 Beispiel für Anforderung an einzelne SOAP-Nachrichten ... 153
7.5.5 Beispiel für Anforderung an einzelne Request-/Response-Paare ... 155

7.6 Operationen für SXQT-Ausdrücke ... 157
7.6.1 Notwendigkeit von Operationen ... 158
7.6.2 Operationen für Message-Elemente ... 159
7.6.3 Operationen für Traces von Hoare ... 161
7.6.4 Operationen für Traces von XQuery ... 169
7.6.5 Operationen für Traces dieser Arbeit ... 171

7.6.6 Spezifikationsspezifische Filteroperationen für die Initialisierung 173
7.6.7 Anwendung von Operationen in Spezifikationen 175

7.7 Beispiele und Klassifikation von SXQT-Ausdrücken 177
7.7.1 Eine Klassifikation für Anforderungen 177
7.7.2 Relevanz von Sichten und Startzustand 179
7.7.3 Klassen CA und SA 180
7.7.4 Klassen CB und SB 185
7.7.5 Klassen CC und SC 187
7.7.6 Klassen CD und SD 195

8 Ablage von SXQT-Ausdrücken 199

8.1 Mögliche Ablageorte für Anforderungen 199
8.1.1 Eigenen Dateien 199
8.1.2 WSDL-Dokument 200
8.1.3 XML-Schemadokument 201
8.1.4 Vergleich 202

8.2 Zweckmäßige Ablageorte in WSDL und XML-Schema 203
8.2.1 XML-Element oder Attribut 204
8.2.2 Typ 205
8.2.3 Andere Bezüge zu XML-Schema 205
8.2.4 SOAP-Nachricht 206
8.2.5 Request-/Response-Paar oder Schnittstelle 207
8.2.6 Mehrere Schnittstellen oder Instanz des Webservices 208

8.3 Anwendung auf SXQT-Ausdrücke 209
8.3.1 Ablageort für SXQT-Ausdrücke 209
8.3.2 XQuery- versus XQueryX-Syntax 210
8.3.3 Vorschlag für Erweiterung von WSDL für SXQT-Ausdrücke 212

9 Automatische Validation 217

9.1 Programmchecker 217

9.2 Zuverlässigkeit mit Programmcheckern 219
9.2.1 Alternativen 219
9.2.2 Fehlersuche 219
9.2.3 Korrektheit von Programmcheckern 220
9.2.4 Abweichungen vom Paradigma 220

9.3 Automatische Validation von Webkomponenten 221
9.3.1 Programmchecker für Webkomponenten 221
9.3.2 Automatische Validation von Prozessen 222
9.3.3 Automatische Validation von Webkomponenten 222
9.3.4 Korrektheit der Validation 223

9.4 Anwendungen 224
9.4.1 Finden von Fehlern in Implementierungen 224
9.4.2 Prüfen des Verständnisses über Schnittstelle 225
9.4.3 Erkennen relevanter Änderungen der Schnittstelle 226
9.4.4 Schutz der Webkomponenten vor böswilligen Zugriffen 227
9.4.5 Verbergen von Spezifikationsverstößen von Webkomponenten 228

9.5 Architektur mit Validatoren 229
9.5.1 Zuordnung zu Webclient oder Webservice 229

9.5.2 Zuordnung zu Prozess 230
9.5.3 Zeitpunkt der Validation 232
9.6 Vorgang der automatischen Validation 235
9.6.1 XML-Schemadokumente 236
9.6.2 SOAP-Standard und Präzisierungsstandard 237
9.6.3 WSDL-Beschreibung 238
9.6.4 SXQT-Ausdrücke 241
9.7 Optimierungen 242
9.7.1 Notwendigkeit von Optimierungen 242
9.7.2 Zusammenfassen von SXQT-Ausdrücken 243
9.7.3 Validation der letzten SOAP-Nachricht 245
9.7.4 Validation nicht für jede SOAP-Nachricht einzeln 246
10 Spezifikation beliebiger XML-Dokumente 247
10.1 Grenzen von XML-Schema 247
10.2 Spezifikation mit absoluten XQuery-Ausdrücken 247
10.3 Spezifikation mit relativen XQuery-Ausdrücken 249
10.4 Vergleich 251
10.5 Einbettung in XML-Schema 251
10.6 Validation gegen erweitertes XML-Schema 254
11 Verwandte Arbeiten zur Spezifikation von Webservices 257
11.1 Spezifikation von dynamischem Verhalten in Schnittstellen 257
11.1.1 Web Services Conversation Language (WSCL) 257
11.1.2 Web Services Choreography Interface (WSCI) 260
11.2 Spezifikation von Geschäftsprozessen 263
11.2.1 Business Process Modelling Language (BPML) 264
11.2.2 Business Process Execution Language for Web Services (BPEL4WS) 266
11.2.3 ebXML Business Process Specification Schema (BPSS) 269
11.3 Spezifikation mit Invarianten in Implementierungssprache - XL 273
11.4 Spezifikation von Seiteneffekten in der realen Welt - DAML-S 277
11.4.1 Semantisches Web 278
11.4.2 DAML-S 278
11.4.3 Vergleich mit SXQT 279
11.5 Verweise auf Spezifikationen mit tModels - UDDI 280
11.5.1 Verzeichnis von Webservices 280
11.5.2 Beschreibung von Webservices in UDDI 281
11.5.3 Vergleich mit SXQT 282
12 Zusammenfassung und Ausblick 283
12.1 Zusammenfassung 283
12.1.1 Interoperabilitätsprobleme in Standards 283
12.1.2 Spezifikationsverfahren für Schnittstellen von Webkomponenten 283
12.1.3 Automatische Validation 284
12.1.4 Verallgemeinerung auf Spezifikation von XML-Dokumenten 285
12.2 Ausblick 285

12.2.1 Anwendung auf unterschiedliche Webkomponenten ... 285
12.2.2 Anwendung auf Problemklassen ... 286
12.2.3 Erweiterung für komplexere Kommunikationsszenarien ... 286
12.2.4 Anwendung für die Verifikation ... 287

A Formulierungen mit XPath-Ausdrücken ... 289

B Verwendete Präfixe ... 291

C Tabellenverzeichnis ... 293

D Abbildungsverzeichnis ... 295

E Literaturverzeichnis ... 299

1 Einleitung

1.1 Motivation

Durch die wachsende Komplexität von Anwendungen rückt die Architektur der Softwaresysteme in den Mittelpunkt. Softwaresysteme werden in Teilkomponenten aufgeteilt, deren Komplexität von Entwicklern beherrscht werden kann. Gibt es bereits Komponenten, die benötigte Dienste erbringen, können sie wiederverwendet oder zugekauft werden.

Das erfordert eine klare Trennung der Komponenten voneinander. Interaktionen zwischen ihnen finden über Schnittstellen statt, die unabhängig von Implementierungsdetails spezifiziert werden. Traditionelle Komponentenarchitekturen verwenden hierzu objektorientierte Konzepte, so dass zum Verständnis der Spezifikation Details des verwendeten Objektmodells notwendig sind.

Eine neue Komponentenarchitektur ist die der Webkomponenten. Im Gegensatz zu traditionellen Komponenten, die als Software vermarktet und in der Umgebung des Softwaresystems ausgeführt werden, werden die Dienste von Webkomponenten über das Internet angeboten. Ausgeführt werden sie in Rechenzentren von Anbietern, in der Regel von ihren Herstellern.

Damit ergibt sich der Vorteil, dass Nutzer Webkomponenten nicht selbst ausführen und daher benötigte Ressourcen (z. B. leistungsfähige Rechner oder große Datenbanken) nicht bereitstellen müssen. Auch Installations- und Upgradeproblematiken entfallen. Die Abrechnung der Nutzung kann nach Nutzungshäufigkeit geschehen, was bei seltener Nutzung niedrige Kosten verursacht. Für den Anbieter ergeben sich kontinuierliche Einnahmen, die eine kontinuierliche Weiterentwicklung der Webkomponenten ermöglicht. Webkomponenten sind gegen unrechtmäßige Verbreitung geschützt, da ihre Software nicht verbreitet wird.

Interoperabilität ist für Webkomponenten eine entscheidende Anforderung. Nutzer müssen andere Plattformen und Entwicklungswerkzeuge verwenden können als Hersteller. Nur so wird die Verwendung von Webkomponenten durch viele Nutzer ermöglicht. Gegenüber traditionellen Komponentenarchitekturen ergibt sich der Vorteil, dass Interaktionen zwischen Webkomponenten als Nachrichten im Internet explizit werden. Solche Schnittstellen lassen sich ohne komplexe Objektmodelle spezifizieren.

Die Spezifikationen müssen präzise formuliert sein, um Missverständnisse zwischen den Entwicklern zu vermeiden, die zu Interoperabilitätsproblemen und damit zu Stabilitätsproblemen führen können. Aus den gleichen Gründen müssen auch die Standards für Webkomponenten präzise sein. Außerdem müssen Änderungen von Schnittstellen erkannt werden, wenn sie für Webkomponenten relevant sind. Änderungen können für zusätzliche Funktionalitäten notwendig sein. Sie zu erkennen wird durch die auf mehrere Unternehmen verteilte Entwicklung erschwert.

1.2 Zielsetzung

Ziel der Arbeit ist Wege aufzuzeigen, wie Interoperabilitätsprobleme zwischen Webkomponenten ausgeschlossen und damit aus ihnen stabile Softwaresysteme zusammengesetzt werden können. Dazu muss das Folgende untersucht werden.

Es ist zu prüfen, ob die Standards für Webkomponenten selbst die Interoperabilität gefährden. Möglich ist das durch missverständliche Formulierungen, aber auch durch zu viele Optionen in den Standards. Beides ermöglicht unterschiedliche Auslegungen und kann daher zu Inkompatibilitäten zwischen Webkomponenten führen.

Aus dem gleichen Grund gefährden auch ungenau spezifizierte Schnittstellen die Interoperabilität. Für die Spezifikation wird heute die Web Services Definition Language (WSDL) verwendet. Daher ist in der Arbeit zu untersuchen, ob mit ihr so präzise spezifiziert werden kann, dass Webkomponenten stets zueinander interoperabel sind, wenn sie die Spezifikation erfüllen. Ist das nicht der Fall, muss in der Arbeit ein Spezifikationsverfahren entwickelt werden, das diese Eigenschaft hat.

Trotz präziser Spezifikationen wird die Stabilität von Softwaresystemen durch Webkomponenten gefährdet, die sie nicht erfüllen. Abweichungen können durch Fehler in den Implementierungen auftreten oder wenn Entwickler Spezifikationen falsch verstanden haben. In der Arbeit ist daher zu untersuchen, wie in beiden Fällen Abweichungen erkannt werden können.

Erkannt werden muss auch, wenn sich eine Schnittstelle geändert hat. Wird mit einer Webkomponente, die einen Dienst anbietet, stets eine Spezifikation ihrer Schnittstellen veröffentlicht, kann sich diese für neue Funktionalitäten der Webkomponente ändern. Nicht immer müssen dann Webkomponenten angepasst werden, die den Dienst nutzen. Daher ist in der Arbeit zu untersuchen, wie Änderungen erkannt werden können, für die es notwendig ist.

1.3 Struktur der Arbeit

Um das Ziel der Arbeit zu erreichen, werden zunächst in Kapitel 2 die Grundkonzepte von Webkomponenten vorgestellt. Dabei wird vom Begriff der Softwarekomponente ausgegangen und erklärt, inwieweit Webkomponenten Softwarekomponenten gleichen und sich von ihnen unterscheiden. Es wird auch skizziert, wie die in Kapitel 1.2 beschriebenen Untersuchungen angegangen werden sollen.

Kapitel 3 führt danach in das Protokoll SOAP ein, mit dem Webkomponenten kommunizieren und Kapitel 4 in die Web Services Description Language (WSDL), mit der heute Schnittstellen von Webkomponenten spezifiziert werden. Da WSDL auf XML-Schema beruht, wird auch XML-Schema vorgestellt.

Kapitel 5 untersucht Interoperabilitätsprobleme, die durch die Standards von SOAP und WSDL entstehen. Es wird festgestellt, dass sie zu viele Optionen zulassen. Daher wird ein Präzisierungsstandard vorgeschlagen, der sie genauer gefasst.

Danach untersucht Kapitel 6, ob Schnittstellen mit WSDL so präzise spezifiziert werden können, dass Webkomponenten stets zueinander interoperabel sind, wenn sie die Spezifikation erfüllen. Es wird gezeigt, dass das nicht der Fall ist. Hierzu werden Anforderungen vorgestellt, sich nicht in WSDL ausdrücken lassen. Dann wird mit Hilfe von Grenzfällen ausgelotet, wo die Grenzen der Ausdrucksfähigkeit von WSDL liegen.

Um die in Kapitel 6 vorgestellten Anforderungen beschreiben zu können, wird in Kapitel 7 das Spezifikationsverfahren SXQT entwickelt. Ausgegangen wird von einem Spezifikationsverfahren von C.A.R. Hoare für beliebige Entitäten. Dieses wird auf

Webkomponenten übertragen, wozu das Modell angepasst, mit WSDL kombiniert, die Sprache XQuery in das Verfahren eingebracht und eine Vielzahl von Operationen übertragen oder neu vorgeschlagen wird. Außerdem wird eine Reihe von Beispielen für in SXQT formulierte Anforderungen an Schnittstellen gegeben.

Wie eine SXQT-Spezifikation Entwicklern zur Verfügung gestellt werden kann, wird in Kapitel 8 untersucht. Unterschiedliche Möglichkeiten werden diskutiert und eine konkrete Vorgehensweise vorgeschlagen.

Danach betrachtet Kapitel 9, wie geprüft werden kann, ob Implementierungen von Webkomponenten ihrer Spezifikation entsprechen. Dazu wird die automatische Validation von Webkomponenten vorgeschlagen, bei dem beobachtete Kommunikationsprozesse gegen eine Spezifikation geprüft werden. Die automatische Validation kann auch verwendet werden, um relevante Änderungen der Spezifikation zu erkennen.

Ähnlich wie Schnittstellen von Webkomponenten heute mit WSDL spezifiziert werden, werden XML-Dokumente mit XML-Schema spezifiziert. Daher liegt es nahe, das in Kapitel 7 entwickelte Spezifikationsverfahren SXQT, auf die Spezifikation beliebiger XML-Dokumente zu übertragen, was in Kapitel 10 geschieht. Übertragen werden dabei auch Ergebnisse aus den Kapiteln 8 und 9.

Kapitel 11 vergleicht das Spezifikationsverfahren SXQT mit verwandten Arbeiten. Abschließend fasst Kapitel 12 die Arbeit zusammen und gibt einen Ausblick.

2 Webkomponenten

Die Einleitung hat verdeutlicht, dass Webkomponenten neue Möglichkeiten eröffnen, aber auch gleichzeitig neue Herausforderungen stellen. Um die Herausforderungen in dieser Arbeit angehen zu können, werden in diesem Kapitel Grundkonzepte von Webkomponenten eingeführt. Es werden für die Arbeit wichtige Begriffe vorgestellt und ein Überblick gegeben, wie die Herausforderungen angegangen werden sollen.

2.1 Webkomponenten sind Komponenten

Schon das Wort „Webkomponente" legt nahe, dass es sich um eine bestimmte Art von Komponenten handelt. Softwaresysteme werden aus ihnen zusammengesetzt. Jede realisiert einen Teil der für das Softwaresystem benötigten Funktionalität. Daher soll nachfolgend zunächst auf Komponenten eingegangen werden, um danach zu untersuchen, in wieweit Webkomponenten[1] ihnen gleichen.

2.1.1 Komponenten

In dieser Arbeit soll für den Begriff „Komponente" die Definition von Szyperski aus [Szyperski 99] verwendet werden. Dort werden Komponenten als Softwarekomponenten bezeichnet. Damit wird zum Ausdruck gebracht, dass es in den Ingenieurwissenschaften auch Hardwarekomponenten gibt, aus denen z. B. Maschinen aufgebaut werden. Softwarekomponenten sind eine Übertragung von diesen auf den Bereich der Softwaresysteme[2].

Szyperski definiert den Begriff „Softwarekomponente" wie folgt:

> *„Eine Softwarekomponente ist eine Einheit zur Komposition, mit vertraglich spezifizierten Schnittstellen und nur explizit angegebenen Kontextabhängigkeiten. Eine Softwarekomponente kann unabhängig verbreitet werden und ist Gegenstand der Komposition durch Dritte."*[3]

Nach der Definition sind Komponenten Einheiten aus denen Softwaresysteme zusammengesetzt werden. Dazu müssen Komponenten zunächst einmal für sich genommen greifbar werden. Sie müssen klar voneinander und von der Umgebung in der sie ausgeführt werden (die selbst durch Komponenten gebildet wird) getrennt sein. Diese Trennlinien werden als Schnittstellen bezeichnet.

---

[1] Es sei darauf hingewiesen, dass im Kontext von „Java 2 Platform, Enterprise Edition" (J2EE) der Begriff „Web component" anders verwendet wird, als der Begriff „Webkomponente" in dieser Arbeit. Im Glossar zu J2EE [Sun 03] wird er definiert als: *„A component that provides services in response to requests; either a servlet or a JSP page."* Dort handelt es sich also um eine Komponente, mit der ein Teil einer Webanwendung realisiert wird und nicht wie in dieser Arbeit um eine Komponente, die ihren Dienst über das Web anbietet.

[2] In [Szyperski 99] wird jedoch auf Seite 8 auch auf fundamentale Unterschiede zwischen Softwarekomponenten und materiellen Komponenten hingewiesen, die sich aus der Tatsache ergeben, dass Softwarekomponenten im Rechner mit unterschiedlichen Parametern beliebig oft instanziiert werden können.

[3] Übersetzung aus [Szyperski 99], S. 34. Englischer Originaltext: *„A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A Software component can be deployed independently and is subject to composition by third parties."*

Für eine Schnittstelle muss spezifiziert sein, wie Komponenten über sie kommunizieren. In diesem Sinne bildet sie eine Art Vertrag zwischen ihnen. Zu spezifizieren sind nicht nur die Schnittstellen, über die Dienste einer Komponente verwendet werden. Außerdem müssen auch die Schnittstellen spezifiziert werden, die die Komponente für ihr Funktionieren benötigt. Solche Abhängigkeiten von ihrer Umgebung bezeichnet Szyperski als Kontextabhängigkeiten.

Die explizite Spezifikation aller Schnittstellen ist eine Voraussetzung dafür, dass Komponenten unabhängig voneinander verbreitet und später zu einem Softwaresystem zusammengesetzt werden können. Als Einheit zur Verbreitung hält Szyperski eine binäre Form (also übersetzten Programmcode) für zwingend, die zusätzlich eine Beschreibung aller Schnittstellen der Komponente enthält. So können unterschiedliche Lieferanten Komponenten eines Systems unabhängig voneinander entwickeln und verbreiten.

Zum System zusammengesetzt werden die Komponenten nicht von Ihren Entwicklern sondern von Dritten. Diese haben kein Wissen über den inneren Aufbau einer Komponente, sondern kennen nur deren Schnittstelle. Über sie können sie die Dienste der Komponente nutzen. Dazu müssen sie aber sicherstellen, dass die Kontextabhängigkeiten der Komponente erfüllt sind.

Frank Griffel führt in [Griffel 98] eine weitere Eigenschaft von Komponenten an: Ihre grobe Granularität. In seiner Definition beschreibt er sie wie folgt:

> *"Eine Komponente ist ein Stück Software [...] das groß genug ist, um eine sinnvoll einsetzbare Funktionalität zu bieten und eine individuelle Unterstützung zu rechtfertigen [...]"*[4].

Die Granularität von Komponenten ist typischerweise gröber als die Klassen objektorientierter Sprachen. Häufig sind sie aus mehreren Klassen zusammengesetzt, die zusammen die sinnvoll einsetzbare Funktionalität der Komponente bereitstellen.

2.1.2 Schnittstellen

Die zentrale Rolle für Komponenten spielen Schnittstellen. Sie sind die Trennlinien zwischen den Komponenten und beschreiben das Protokoll zwischen ihnen. Don Box geht in [Box 00a] soweit, den Begriff „schnittstellenbasierte Programmierung" („interface-based programming") zu verwenden. Er definiert Schnittstellen als *„abstrakte Typen, die einen Vertrag zwischen zwei Agenten definieren: Die Implementierung (häufig bezeichnet als die Komponente) und der Konsument (häufig bezeichnet als der Client)."*[5]

In der Definition unterscheidet Don Box zwischen Komponenten und ihren Clients und verwendet für beide den Oberbegriff „Agent". Die Unterscheidung in Komponenten und Client betont die beiden Rollen bei der Verwendung der Schnittstelle. Eine Komponente stellt einen Dienst über eine Schnittstelle bereit. Der Client greift über die Schnittstelle auf den Dienst zu. Abweichend von dieser Begriffsbildung, sollen nachfolgend auch Clients als Komponenten bezeichnet werden, weil auch sie Teile des Softwaresystems sind. Der Begriff „Komponente" wird also statt „Agent" als Oberbegriff verwendet.

---

[4] [Griffel 98], S. 31.

[5] Übersetzung aus [Box 00a], S. 245f. Englischer Originaltext: *„Interfaces are abstract types that define a contract between two agents: the implementation (often referred to as the component) and the consumer (often referred to as the client)."*

Die Definition von Don Box stellt in den Vordergrund, dass eine Schnittstelle ein abstrakter Typ ist, der einen Vertrag definiert. Der Vertrag beschreibt, wie Client und Komponente miteinander kommunizieren, also das Protokoll zwischen ihnen. Der Begriff „Vertrag“ drückt aus, dass das Protokoll in jedem Fall einzuhalten ist. Ein Nichteinhalten kann die Stabilität des Softwaresystems gefährden.

Mit dem Begriff „abstrakter Typ“ unterstreicht Don Box, dass Schnittstellen eine ähnliche Bedeutung haben, wie abstrakte Typen von Programmiersprachen. Abstrakte Typen sagen nichts über ihre Implementierung aus. Für sie kann es unterschiedliche Implementierungen geben. Das spielt für den Client jedoch keine Rolle. Er kennt nur den abstrakten Typen und die mit ihm verbundene Semantik.

Noch klarer wird die Bedeutung des Begriffs „abstrakter Typ“, vor dem Hintergrund, dass Komponenten typischerweise ihre Schnittstellen mit Hilfe von Klassen im objektorientierten Sinne bereitstellen. Das liegt nahe, weil Komponenten häufig als Sammlungen von Klassen realisiert sind. Eine Komponente wird dann von einem Client verwendet, indem er in ihr enthaltene Klassen instanziiert. So erhält er Objekte, über die er mit der Komponente interagieren kann. Die Schnittstelle der Komponente wird also durch die Schnittstellen der Objekte gebildet, die aus ihren Klassen instanziiert wurden. Die Klassen implementieren die Schnittstellen, die in Programmiersprachen als abstrakte Typen realisiert werden können.

In [Box 00a] wird ausgeführt, dass Schnittstellen dieser Art auch von vielen Programmiersprachen (wie Java und Visual Basic) und Komponenteninfrastrukturen zur Realisierung von Komponenten oder Klassen unterstützt werden. Die Schnittstellen sind in der Regel typisiert. Sie unterscheiden sich in Details zugrunde liegender Objektmodelle. Das grundlegende Verständnis, was eine Schnittstelle ist, ist aber stets gleich. Eine Schnittstelle ist danach eine typisierte Sammlung von Operationen (bzw. Methoden), die der Client aufruft und die Klasse implementiert. Die Schnittstelle definiert die Syntax und Semantik der einzelnen Operationen sowie die Semantik der Schnittstelle als Ganzes. Für jede Schnittstelle gibt es einen eindeutigen Bezeichner. In ihr wird jede Operation eindeutig gekennzeichnet.

Operationen sind die Grundbausteine der Kommunikation zwischen den Komponenten. Jede hat eine Typensignatur, die aus einer Liste von Namen und den Typen ihrer formalen Parameter besteht. Für jeden Parameter wird angegeben, in welche Richtung Daten ausgetauscht werden. In-Parameter werden vom Client zur Klasse übergeben, Out-Parameter von der Klasse zum Client, Inout-Parameter in beide Richtungen. In-Parameter entsprechen Werteparametern (call by value) von Programmiersprachen, während Inout-Parameter gewisse Ähnlichkeiten mit Referenzparametern (call by reference) haben. Meistens wird für eine Operation auch ein unbenannter Out-Parameter erlaubt, mit dem Ergebnisse von Funktionen repräsentiert werden.

Abbildung 1 veranschaulicht Schnittstellen dieser Art an einem Beispiel. Gezeigt wird eine Komponente, die drei Klassen enthält. Alle drei haben Schnittstellen, die nur innerhalb der Komponente zugänglich sind. Klasse1 und Klasse2 haben zusätzlich Schnittstellen, die von Clients außerhalb der Komponente verwendet werden können. Sie bilden zusammen die Schnittstelle der Komponente. Für Schnittstelle1 ist dargestellt, welche drei Methoden sie hat. Für jede Methode sind Parameter mit ihren Typen angegeben und es wurde zugefügt, ob es sich um einen In-, Out- oder Inout-Parameter handelt. Schließlich ist links noch ein Client dargestellt, der auf die Komponente über die Schnittstelle1 zugreift.

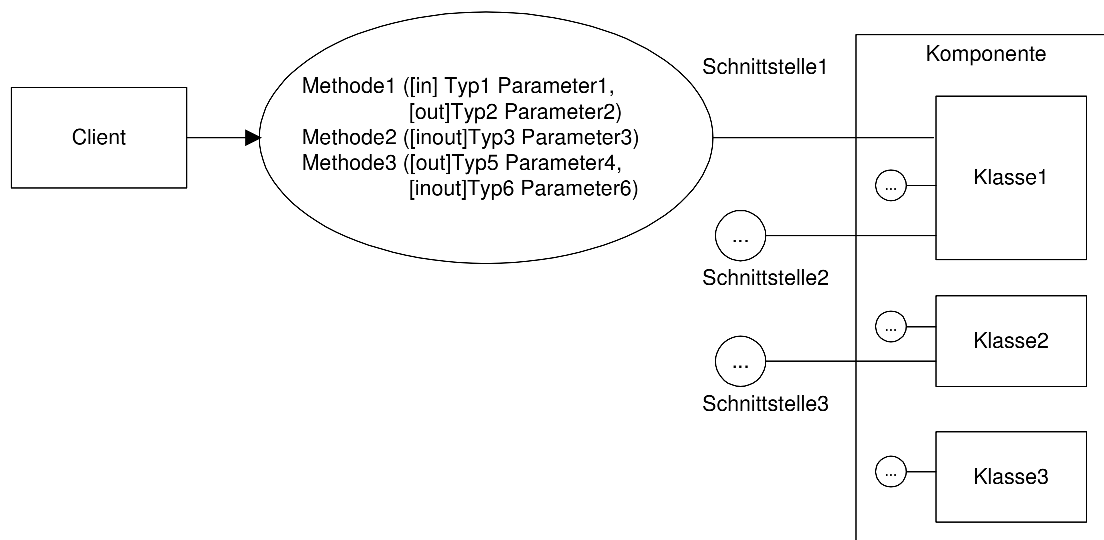

Abbildung 1: Komponente mit Klassen und Schnittstellen

Für den Client sind nur die Schnittstellen der Komponente relevant, die er für seine Implementierung benötigt. Die Komponente kann ausgetauscht werden, wenn nur die vom Client verwendeten Schnittstellen gleich bleiben. Komponenten eines Softwaresystems können so einzeln durch neuere Versionen ersetzt werden, ohne die Funktion des Systems zu beeinträchtigen. Dabei können neue Dienste durch zusätzliche Schnittstellen einer Klasse bereitgestellt werden oder auch durch neue Klassen in der Komponente.

Die Verwendung von Schnittstellen erlaubt, Programmcode für den Client zum Zugriff auf unterschiedliche Komponenten wiederzuverwenden. Hierzu müssen die unterschiedlichen Komponenten dieselben Schnittstellen implementieren. Dann kann Programmcode, der nur unter Berücksichtigung dieser Schnittstelle erstellt wurde, in Clients für die unterschiedlichen Komponenten wiederverwendet werden. Diese Idee legt es nahe, Schnittstellen für bestimmte Anwendungen zu standardisieren. Komponenten sollten nach Möglichkeit solche Standardschnittstellen verwenden, um ihre Dienste bereitzustellen. Neben der Wiederverwertbarkeit von Programmcode erleichtert das auch Entwicklern das Verständnis, wie eine Komponente verwendet wird. Sie müssen dann nur einmal die Verwendung einer Standardschnittstelle erlernen.

2.1.3 Komponentenarchitekturen

Regeln, wie Komponenten und ihre Schnittstellen zu realisieren sind, bilden zusammen eine Komponentenarchitektur. Zu ihnen gehören die vorgestellten Regeln für Schnittstellen ebenso wie ein vorausgesetztes Objektmodell und Regeln wie Komponenten technisch realisiert werden. Unterschiedliche Komponentenarchitekturen wurden entwickelt. Sie unterscheiden jedoch in vielen Details. Damit Komponenten miteinander komponiert werden können, müssen sie nach den Regeln der gleichen Komponentenarchitektur realisiert sein. Sonst könnten z. B. unterschiedliche Objektmodelle dazu führen, dass eine Komponente Objekte einer anderen mit falschen Annahmen verwendet.

Für die unterschiedlichen Komponentenarchitekturen wurden Komponenteninfrastrukturen implementiert, also Software die benötigt wird, um Komponenten der Architektur zu implementieren und zu Systemen zu komponieren. Die vier wichtigsten Komponentenarchitekturen sind CORBA, COM, JavaBeans und .NET. Sie werden im Folgenden vorgestellt.

Die Common Object Request Broker Architecture (CORBA) ist eine Reihe von Standards. Sie wurde von der Object Management Group (OMG) [OMG 02a] entwickelt, einem Konsortium von etwa 800 Partnern aus Industrie und Forschung. CORBA hat das Ziel, in unterschiedlichen Sprachen implementierte Objekte, die verteilt, auf unterschiedlichen Plattformen ausgeführt werden, miteinander interagieren zu lassen. Dazu wurden von unterschiedlichen Unternehmen mit unterschiedlichen Technologien Komponenteninfrastrukturen implementiert. Details zu CORBA finden sich u. a. in [Siegel 00] oder auf der Website der OMG [OMG 02a], die auch die CORBA Standards (z. B. [OMG 02b]) enthält.

Die Komponentenarchitektur COM (Component Object Model) wurde von der Firma Microsoft [Microsoft 02a] entwickelt und wird in der Literatur z. T. auch mit den Begriffen „OLE2“, „ActiveX“ oder „COM+“ bezeichnet, die eigentlich für auf COM basierende Technologien stehen. Komponenteninfrastrukturen sind seit Windows 3.1 in allen Betriebssystemen der Microsoft Windows Familie integriert, was zu einer großen Verbreitung auf diesen Betriebssystemen geführt hat. Zusätzlich gibt es eine Portierung für UNIX von der Firma Software AG [SoftwareAG 02a]. Details zu COM sind z. B. in [Rogerson 97] und [Eddon 99] enthalten.

JavaBeans ist die Komponentenarchitektur zur Sprache Java. Beides wurde von der Firma Sun [Sun 02a] entwickelt. Klassen müssen in der Sprache Java implementiert werden. Auch Schnittstellen werden mit Sprachelementen von Java beschrieben. JavaBeans werden in der Laufzeitumgebung für Java ausgeführt, der Java Virtual Machine (JVM). Deren große Verbreitung auf vielen Plattformen führt auch zur großen Verbreitung von JavaBeans. Details zu JavaBeans einschließlich der Spezifikation finden sich in [Sun 02b].

.NET ist ein Satz von Technologien von Microsoft [Microsoft 02a] auf denen zukünftig die komponentenorientierte Anwendungsentwicklung auf Betriebssystemen der Microsoft Windows Familie beruhen soll. Es steht erst seit Anfang des Jahres 2002 zur Verfügung. Auffallend ist die große Ähnlichkeit zu Java und JavaBeans. Z. B. gibt es eine virtuelle Maschine ähnlich der JVM, die als Common Language Runtime (CLR) bezeichnet wird. Neben Unterschieden in Details ist der Hauptunterschied, dass zur Entwicklung von Komponenten unterschiedliche Sprachen verwendet werden können. Mehr zu .NET findet sich in [Microsoft 02b].

Für die vorgestellten Komponentenarchitekturen wurden Komponenteninfrastrukturen (kurz: Infrastrukturen) implementiert, mit denen Komponenten erstellt und danach zu Systemen komponiert werden können. Entwickler von Komponentensoftware sollen von den Komponenteninfrastrukturen möglichst gut bei allen anfallenden Problemen unterstützt werden. In allen Softwaresystemen wieder auftretende Anforderungen müssen dazu schon von der Infrastruktur gelöst sein.

Die offensichtlichste Anforderung ist, die Komposition von Komponenten zu ermöglichen, ebenso wie eine Kommunikation zwischen ihnen. Hierzu müssen Komponenten und ihre Schnittstellen bezeichnet werden können, so dass Entwickler auf sie verweisen können. Das geschieht häufig über Namen oder andere Bezeichner. Bei COM werden z. B. Klassen (und Schnittstellen) mit Hilfe sogenannter Globally Unique Identifier (GUID) identifiziert. Das sind binäre Bezeichner mit einer Länge von 128 Bit, die verteilt und weltweit eindeutig erzeugt werden können. Wird ein Objekt einer solchen Klasse benötigt, muss die Infrastruktur aus dem Bezeichner die zugehörige Komponente ermitteln und laden, bevor sie das Objekt instanziieren kann.

Ist ein Objekt instanziiert und sollen Methoden aufgerufen werden, ist es wieder die Aufgabe der Infrastruktur das zu ermöglichen. Befinden sich Objekt und Client im selben Prozess oder derselben virtuellen Maschine, ist das einfach zu realisieren. Aufwen-

diger ist es, wenn sich beide in unterschiedlichen Prozessen oder auf unterschiedlichen Rechnern befinden. Für den Client soll der Aufruf der Methode trotzdem auf die gleiche Art möglich sein. CORBA stellt hierzu den Object Request Broker (ORB) zur Verfügung, der Methodenaufrufe entgegennimmt und transparent an das Objekt weiterleitet.

Die Realisierung von großen Systemen, die von vielen Benutzern gleichzeitig verwendet werden, stellt Entwickler vor zusätzliche Anforderungen: Sicherheit, Zuverlässigkeit, Verfügbarkeit und Skalierbarkeit. Auch hier unterstützen Infrastrukturen die Entwickler.

2.2 Unterschiede zu traditionellen Komponenten

Eine Architektur für Webkomponenten ist eine Fortentwicklung der Komponentenarchitekturen. Webkomponenten unterscheiden sich von den in Kapitel 2.1 beschriebenen Komponenten vor allem darin, dass nicht ihre Software verkauft wird, sondern der Dienst den sie erbringen. Auf ihn können Nutzer über das Internet[6] zugreifen und so die Webkomponente in ihr Softwaresystem integrieren.

Anbieter haben ein großes Interesse, dass möglichst viele Nutzer den Dienst ihrer Webkomponenten verwenden, weil das zu größeren Einnahmen führt. Daher ist Interoperabilität für Webkomponenten das entscheidende Kriterium. Webkomponenten müssen für unterschiedliche Rechnertypen und Betriebsysteme erstellt werden können und trotzdem miteinander interoperabel sein. Das erfordert einen herstellerübergreifenden Konsens über verwendete Protokolle, mit denen Webkomponenten miteinander kommunizieren.

Ebenso müssen Schnittstellen von Webkomponenten präzise spezifiziert werden. Eine solche Spezifikation muss möglichst unzweideutig verstanden und über Internet verbreitet werden können. Nur so können Nutzer ein korrektes Verständnis erlangen, wie sie auf eine Webkomponente zugreifen können, was für die Stabilität des Softwaresystems entscheidend ist.

In der Literatur wird meistens statt dem Begriff „Webkomponente“ der Begriff „Webservice“ verwendet, z. B. in den Standards des World Wide Web Consortiums (W3C) [W3C 02a]. Durch den Begriff „Webservice“ wird die Rolle als Diensterbringer betont. Daher soll in dieser Arbeit unter einem Webservice eine Webkomponente in ihrer Rolle als Diensterbringer verstanden werden. Analog dazu ist ein Webclient eine Webkomponente in ihrer Rolle als Dienstnehmer. Es greifen also stets Webclients auf Dienste von Webservices zu, was in Abbildung 2 veranschaulicht wird.

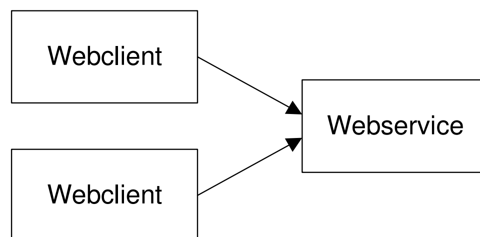

Abbildung 2: Zwei Webclients, die auf einen Webservice zugreifen

[6] In dieser Arbeit wird davon ausgegangen, dass über das Internet auf Webkomponenten zugegriffen wird. Grundsätzlich können Webkomponenten natürlich auch miteinander über andere Netze kommunizieren, die auf den gleichen Technologien wie das Internet beruhen.

2.3 Traditionelle Komponenten als Webkomponenten

Nachfolgend soll untersucht werden, wie Webkomponenten realisiert werden könnten. Dazu wird zunächst betrachtet, ob nicht Komponenten existierender Komponentenarchitekturen als Webkomponenten verwendet werden können. Danach wird festgestellt, warum das nicht sinnvoll ist.

Die Idee eine exsistierende Komponentenarchitektur für Webkomponenten zu verwenden, erscheint zunächst naheliegend. Z. B. könnte CORBA verwendet werden, um über das Internet Dienste von CORBA-Komponenten anzubieten. Zur Kommunikation könnten die Mechanismen der Komponentenarchitektur für entfernte Methodenaufrufe verwendet werden. CORBA stellt hierzu für das Internet das Inter-ORB Protokoll (IIOP) zur Verfügung.

Ein Vorteil diese Ansatzes wäre, dass die zur Erstellung von Webkomponenten benötigten Infrastrukturen zur Verfügung stünden. Mit der Entwicklung von Webkomponenten könnte also direkt begonnen werden. Außerdem könnten für die Infrastruktur vorhandene Komponenten direkt als Webservices verwendet werden.

Es muss jedoch möglich sein, dass unterschiedliche Hersteller miteinander interoperable Infrastrukturen für Webkomponenten implementieren. Würde die Infrastruktur eines Herstellers vorgeschrieben, würden andere Hersteller alternative Infrastrukturen implementieren und diese propagieren. Für die unterschiedlichen Infrastrukturen muss sichergestellt werden, dass mit ihnen realisierte Webkomponenten miteinander interoperieren können. Für CORBA wird das z. B. mit Hilfe des Protokolls IIOP gelöst.

Eine ähnliche Problematik besteht jedoch auch für die verwendete Komponentenarchitektur selbst. Ein Konsens für eine der vier in Kapitel 2.1.3 vorgestellten Komponentenarchitekturen lässt sich nicht finden, da jede wesentliche Befürworter hat. Daher stellt sich die Frage, ob es möglich ist, in unterschiedlichen Komponentenarchitekturen implementierte Webkomponenten miteinander zu integrieren. Ein Webclient müsste auf einen Webservice zugreifen können, der in einer anderen Komponentenarchitektur implementiert wurde. Hierzu könnten Brücken verwendet werden.

2.3.1 Brücken für Interoperabilität von Komponentenarchitekturen

Die Idee der Brücken wurde nicht für Webkomponenten entwickelt. Unabhängig von Webkomponenten haben sie das Ziel, die Komposition von Softwaresystemen aus Komponenten unterschiedlicher Komponentenarchitekturen zu ermöglichen. Hierzu kann es unterschiedliche Gründe geben. So ist es z. B. bei der Fusion von zwei Unternehmen möglich, dass in beiden unterschiedliche Komponentenarchitekturen verwendet wurden. Aber auch innerhalb eines Unternehmens können aufgrund einer gewachsenen „EDV-Landschaft“ oder durch Zukauf von Softwaresystemen Komponenten unterschiedlicher Komponentenarchitekturen vorhanden sein. In beiden Fällen kann es notwendig werden, solche Komponenten zu einem Softwaresystem zu integrieren, ebenso wie bei enger Zusammenarbeit von zwei Unternehmen.

Als Lösung für solche Integrationsprobleme werden Brücken verwendet [Szyperski 99]. Brücken machen Komponenten einer Komponentenarchitektur in einer anderen nutzbar. Abbildung 3 illustriert das an zwei Webkomponenten, die über eine Brücke auf Komponenten einer anderen Komponentenarchitektur zugreifen.

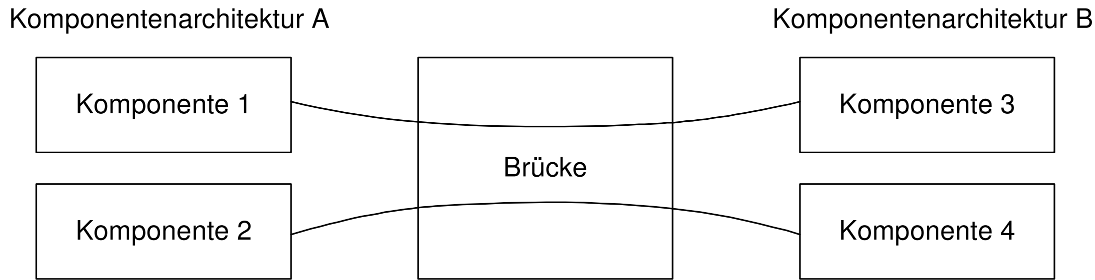

Abbildung 3: Über eine Brücke verbundene Komponenten

In [Foody 97] wird unterschieden zwischen Mapping und Interworking. Beim Mapping werden einige Komponenten einer Komponentenarchitektur als Komponenten einer anderen zur Verfügung gestellt, jedoch nur in dieser einen Richtung. Damit können Komponenten in einfachen Fällen in der anderen Komponentenarchitektur verwendet werden. Es ist aber z. B. nicht möglich Objektreferenzen als Parameter von Methoden zu übergeben, weil diese in der anderen Komponentenarchitektur keine Bedeutung hätten. Das ist beim Interworking anders. Komponenten von zwei Komponentenarchitekturen werden in beiden Richtungen aufeinander abgebildet. Das schließt auch die Abbildung von Objektreferenzen mit ein.

Für verschiedene Paare der vier in Kapitel 2.1.3 vorgestellten Komponentenarchitekturen wurden Brücken entwickelt. Bei der Entwicklung von .NET wurde die Interoperabilität mit COM direkt berücksichtigt. .NET-Komponenten können als COM-Komponenten registriert und dann von COM-Komponenten verwendet werden. Umgekehrt ist es auch möglich, dass .NET Komponenten direkt COM-Komponenten verwenden [Busby 01]. Eine Brücke zwischen COM und Java ist in der JVM von Microsoft enthalten. Java-Klassen, die in ihr ausgeführt werden, können auf COM-Komponenten zugreifen oder so registriert werden, dass sie selbst als COM-Komponenten verwendet werden können. Benötigte Tools gehören zu Microsofts Java-Implementierung J++ [Microsoft 02c]. Eine weitere Brücke zwischen COM und Java wird auch von Sun zur Verfügung gestellt. Sie erlaubt es allerdings nur JavaBeans unter Microsoft Windows so zu registrieren, dass COM-Komponenten sie verwenden können [Sun 02c].

Zwischen CORBA und anderen Komponentenarchitekturen wurden nicht nur proprietäre Brücken entwickelt. Die Interoperabilitätsproblematik wird auch im CORBA Standard [OMG 02b] behandelt. Ein wichtiges Thema ist dabei auch die Interoperabilität zwischen unterschiedlichen Infrastrukturen für CORBA, die von unterschiedlichen Herstellern implementiert wurden. Hierzu wurde im CORBA Standard eine infrastrukturunabhängige Darstellung von Objektreferenzen sowie eine Protokollfamilie zur Kommunikation zwischen den Infrastrukturen festgelegt, zu der auch IIOP gehört. Auch für die Brücken zwischen CORBA und COM befinden sich in im CORBA Standard [OMG 02b] die Grundlagen. [Foody 97] nennt mehrere Produkte, die solche Brücken implementieren.

2.3.2 Grenzen von Brücken

Brücken erlauben es also, Komponenten unterschiedlicher Komponentenarchitekturen zu integrieren. Für Webkomponenten erscheint aber schon problematisch, dass für jedes Paar von zwei Komponentenarchitekturen eine eigene Brücke benötigt wird, was insgesamt eine Vielzahl von Brücken notwendig macht.

Schwerwiegender ist jedoch, dass die Abbildung der Komponentenarchitekturen aufeinander häufig nicht effizient und für den Entwickler nicht vollständig transparent ge-

schieht. Eine Brücke muss alle Aspekte der Komponentenarchitekturen aufeinander abbilden. Solche Abbildungen sind aber nicht trivial und häufig nicht vollständig transparent möglich. Das soll im Folgenden ausgeführt werden.

Neben der Hauptaufgabe, die Zugriffe der Komponenten über Architekturgrenzen hinweg zu ermöglichen, muss eine Brücke zunächst die Spezifikationen der Komponenten sowie ihrer Schnittstellen auf die andere Komponentenarchitektur abbilden. Diese Aufgabe wird dadurch erschwert, dass in den Komponentenarchitekturen unterschiedliche Verfahren zur Spezifikation verwendet werden. COM und CORBA verwenden (ähnliche) Schnittstellenbeschreibungssprachen (Interface Definition Languages, IDLs), COM zusätzlich „Type Libraries". Java dagegen spezifiziert Schnittstellen und Klassen der Komponenten in der Sprache Java. Eine transparente Abbildung würde z. B. erfordern, dass eine in CORBA IDL spezifizierte Schnittstelle, in C++ unter CORBA die gleiche Definition hat, wie in COM, wenn eine Brücke sie auf COM abbildet.

Schon einfache Namenskonventionen können dabei ein Problem sein. So erlaubt COM als erstes Zeichen eines Methodennamens einen Underscore („_"), CORBA jedoch nicht. Auch müssen Typen, die für Parameter von Methodenaufrufen verwendet werden, aufeinander abgebildet werden. Hier liegen die Probleme darin, dass für alle in den Komponentenarchitekturen erlaubten Typen, einfache wie zusammengesetzte, eine geeignete Abbildung gefunden werden muss. Probleme liegen häufig im Detail. In [Foody 97] wird z. B. darauf hingewiesen, dass COM zwei Arten von Typen für Zeichenketten hat, CORBA aber nur einen. Eine Art von Zeichenketten wird mit Null-Zeichen abgeschlossen, was Null-Zeichen innerhalb der Zeichenkette verbietet. In der anderen ist die Länge der Zeichenkette explizit angegeben, so dass Null-Zeichen erlaubt sind, aber die maximale Länge begrenzt ist.

Ein besonderer Datentyp, der häufig als Parameter bei Methodenaufrufen übergeben wird, sind Objektreferenzen. Diese werden immer dann benötigt, wenn eine Methode eines Objektes aufgerufen werden soll. Wird eine Objektreferenz an eine Komponente einer anderen Komponentenarchitektur übergeben, soll diese auch in der Lage sein, sie zu verwenden. Dazu ist Interworking erforderlich (siehe Kapitel 2.3.1). Die Brücke muss also Objekte in beide Richtungen abzubilden. Außerdem muss die Objektreferenz durch die Brücke in eine Form abgebildet werden, die von der Komponente als Objektreferenz verstanden wird.

Um solche Abbildungen zu vermeiden, wurden bei der Standardisierung von CORBA für die Interoperabilität unterschiedlicher CORBA Infrastrukturen eine portable, infrastrukturunabhängige Darstellung von Objektreferenzen festgelegt. Eine solche Festlegung ist zwischen unterschiedlichen Komponentenarchitekturen selbstverständlich nicht möglich. Hinzu kommt, dass in COM keine Objektreferenzen verwendet werden. Statt dessen werden Schnittstellen von Objekten referenziert. Eine Brücke zwischen COM und CORBA müsste das berücksichtigen.

Solche Unterschiede zwischen den Objektmodellen der Komponentenarchitekturen sind es, die die Hauptprobleme bei der Abbildung darstellen. So spielt in COM die Reihenfolge von Methoden in einer Schnittstelle eine Rolle, in CORBA jedoch nicht [Foody 97]. Java kennt die Vererbung von Klassen, COM kennt dagegen nur Schnittstellenvererbung. Eine Vererbungshierarchie von Java kann also auf COM nicht abgebildet werden. Daher ist bei J++ als Brücke eine solche in COM auch nicht sichtbar [Szyperski 99].

Auch Konzepte wie die Speicherverwaltung und Versionierung können ein Problem sein. Die Speicherfreigabe bei Java geschieht durch eine automatische Garbage-

Collection, auch im verteilten Fall. COM verwendet hierzu dagegen Referenzzähler, CORBA hat kein generelles Konzept für eine globale Speicherverwaltung [Szyperski 99]. Greift nun z. B. eine CORBA Komponente auf ein Objekt in einem JavaBean zu, wie soll die Brücke entscheiden, wann das Java-Objekt nicht mehr benötigt wird?

Bei der Versionierung ist das Problem, dass nach Änderung einer Komponente diese möglicherweise nicht mehr korrekt mit anderen zusammenarbeitet. Das kann geschehen, wenn die Syntax oder Semantik einer Schnittstelle geändert wurde. Die Komponentenarchitekturen gehen mit dem Problem unterschiedlich um. In COM darf eine Schnittstelle nach ihrer Veröffentlichung nicht mehr geändert werden. CORBA verwendet Haupt- und Nebenversionsnummern, die jedoch nicht zwingend geprüft werden. Java schließlich befasst sich mit dem Problem nur auf der Ebene der Binärkompatibilität [Szyperski 99]. Eine transparente Abbildung dieser Konzepte erscheint unmöglich.

Interessant ist auch die Beobachtung, dass das Verständnis, was durch ein Objekt repräsentiert wird, in den Komponentenarchitekturen auseinandergeht. So sind in CORBA Objekte typischerweise langlebig. Z. B. könnte ein Drucker als Objekt realisiert werden. Das Objekt wird dann von den Clients unterschiedlicher Benutzer verwendet. Es wird in Verzeichnisdienste eingetragen und behält auch nach einem Neustart des ausführenden Rechners seine Identität. Anders dagegen in COM. Hier sind Objekte typischerweise kurzlebig. Um zu drucken würde ein Client ein neues Objekt einer geeigneten Klasse erzeugen. Das Objekt wird danach, spätestens beim Beenden des Clients wieder vernichtet. Es wird in der Regel nicht von anderen Benutzern verwendet und überdauert auf keinen Fall den Neustart des ausführenden Rechners. Eine bijektive Abbildung (also „eins zu eins") solcher Objekte zwischen COM und CORBA durch eine Brücke würde dem Gedanken der Transparenz widersprechen.

Solche Unterschiede in den Objektmodellen der Komponentenarchitekturen erschweren deren Abbildung erheblich, obwohl sie alle auf Klassen, Objekten und Methoden beruhen und auch viele andere Gemeinsamkeiten aufweisen. Um so schwerer wird die Abbildung, wenn auch diese Annahmen nicht mehr gelten, wenn in einer Komponentenarchitektur Komponenten ihre Dienste nicht in Form von Klassen zur Verfügung stellen. Auf was müssten dann Objekte, Methoden oder Objektreferenzen abgebildet werden?

Unabhängig von Interoperabilitätsproblemen muss auch darauf hingewiesen werden, dass die Abbildung von Methodenaufrufen durch Brücken die Zeit für Methodenaufrufe erhöht und diese somit weniger effizient sind als im Falle nur einer Komponentenarchitektur. So werden z. B. in COM Methodenaufrufe zwischen Komponenten, die innerhalb eines Prozesses ausgeführt werden, auf die gleiche Art realisiert, wie üblicherweise bei C++ Compilern Aufrufe von virtuellen Methoden. Damit sind sie ebenso effizient. Eine Brücke muss dagegen zusätzlichen Programmcode für die Abbildung ausführen. Sollte sich die Brücke nicht im gleichen Prozess befinden, ist zusätzlicher Aufwand für die Kommunikation zwischen den Prozessen notwendig.

Zusammenfassend sei daher festgestellt, dass Brücken in der beschriebenen Form zwar die Komposition von Komponenten unterschiedlicher Komponentenarchitekturen ermöglichen. Die großen Unterschiede zwischen den Objektmodellen und Paradigmen verhindern aber eine für den Entwickler transparente Abbildung. So kommt auch Szyperski zu dem Schluss, dass Brücken nur ein Kompromiss sein können, weil die „Lücken" zwischen den Komponentenarchitekturen zu groß sind, als dass Brücken sie vollständig schließen könnten ([Szyperski 99], Seite 36). Daher sind Brücken nicht der geeignete Weg, um in unterschiedlichen Komponentenarchitekturen erstellte Webkomponenten miteinander zu verbinden.

2.3.3 Model Driven Architecture

Komponenten unterschiedlicher Komponentenarchitekturen zu verbinden, ist auch Anspruch der Model Driven Architecture (MDA). Sie wurde von der Object Management Group entwickelt, die auf ihrer Website [OMG 02a] Informationsmaterial zur MDA bereitstellen. Hierzu gehört [Soley 01], in dem ein Überblick über MDA vermittelt wird und aus dem Informationen für die nachfolgende Beschreibung der MDA übernommen sind.

Basis für das Verbinden unterschiedlicher Komponentenarchitekturen ist ein Modell, das die Anwendung unabhängig von Komponentenarchitektur und Plattform beschreibt. In diesem sogenannten Platform Independent Model (PIM) werden benötigte Datenmodelle und das Verhalten der Anwendung beschrieben. Aus dem PIM wird danach das sogenannte Platform Specific Model (PSM) generiert, das von Komponentenarchitektur und Plattform abhängig ist. Für diese Abbildung enthält die MDA Werkzeuge. Teilweise muss die Abbildung aber auch manuell geschehen. Aus dem PSM können danach wieder mit Werkzeugen der MDA große Teile benötigter Implementierungen von Komponenten generiert werden. Beide Abbildungen sind für die Komponentenarchitekturen aus Kapitel 2.1.3 in Abbildung 4 veranschaulicht.

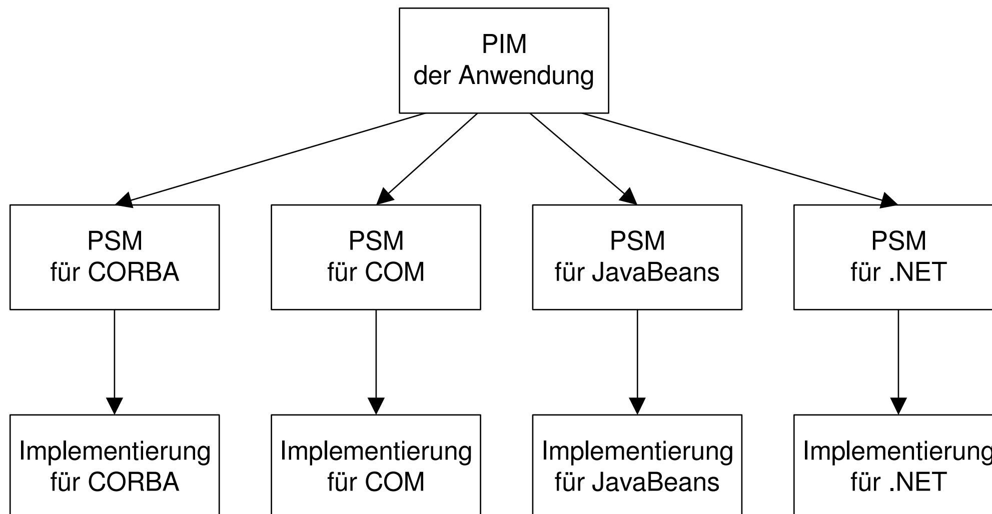

Abbildung 4: Abbildung eines PIMs in PSMs und in Implementierungen

Die MDA erlaubt auch Komponenten zu integrieren, die nicht für die MDA entwickelt wurden. Auch für solche „Legacy-Anwendungen“ wird ein PIM benötigt, das aus ihr mit Hilfe von Reverse Engineering gewonnen wird. Auch um das zu unterstützen, enthält die MDA Werkzeuge.

Für die Interoperabilität zwischen Komponentenarchitekturen werden anwendungsspezifische Brücken generiert. Das wird dadurch vereinfacht, dass mit der PIM ein Modell der Anwendung zur Verfügung steht, für das bekannt ist, wie es auf die unterschiedlichen Komponentenarchitekturen abgebildet wurde (PSM). Mit diesem Wissen sowie dem über die Komponentenarchitekturen selbst, können Brücken die anwendungsspezifischen Modelle aufeinander abbilden.

Es stellt sich die Frage, ob mit der MDA das Interoperabilitätsproblem zwischen verschiedenen Komponentenarchitekturen so gelöst werden kann, dass diese für Webkomponenten verwendet werden können. Das würde bedeuten, dass für Softwaresysteme zunächst das PIM entwickelt werden müsste, unabhängig von der Tatsache, dass Webkomponenten verwendet werden. Danach könnte für unterschiedliche Komponentenarchitekturen das PSM generiert und die Komponenten in unterschiedlichen Komponen-

tenarchitekturen implementiert werden. Die Komponenten können als Webkomponenten im Internet verteilt werden. Um eine Kommunikation zwischen ihnen zu erlauben, müssten aus der PIM für alle Paare kommunizierender Komponentenarchitekturen anwendungsspezifische Brücken generiert werden.

Es fällt auf, dass für den Softwareentwurf bei der MDA grundsätzlich andere Annahmen gemacht werden als bei Webkomponenten. Bei der MDA findet ein Top-Down-Entwurf statt, vom PIM zum PSM zur Implementierung. Nur die Integration von Legacy-Systemen weicht davon ab. Bei Softwaresystemen aus Webkomponenten findet dagegen eher ein Bottom-Up-Entwurf statt. Webservices werden von Anbietern bereitgestellt, die nur eine grobe Vorstellung davon haben können, wie ein Webservice später von anderen Entwicklern in unterschiedlichen Softwaresystemen eingesetzt wird. Diesen Entwicklern soll ein Dienst angeboten werden. Über notwendige Anforderungen an die Schnittstelle des Webservices hinaus, sollen ihnen keine Vorgaben für ihre Softwaresysteme gemacht werden. Sie könnten zu Konflikten führen, wenn mehrere Webservices mit sich widersprechenden Vorgaben verwendet werden sollen.

Durch den Bottom-Up-Entwurf müssen alle Webkomponenten als Legacy-Anwendungen durch Reverse Engineering in das PIM integriert werden. Insbesondere wenn ein großer Teil der Funktionalität des Softwaresystems durch Webkomponenten erbracht wird, muss auch ein großer Teil des PIM auf diese Art erstellt werden.

Es ist außerdem unklar, ob es einen Konsens geben wird, dass alle Webkomponenten mit der MDA entwickelt werden. Ansonsten werden einige Entwickler ihre Webservices anders entwickeln. Für solche Webservices würde aber eine Spezifikation benötigt, die für das Reverse Engineering verwendet werden kann. Komponentenarchitekturen stellen zwar Spezifikationen der Schnittstellen von Komponenten zur Verfügung, aber nicht Spezifikationen oder Modelle der Komponente als ganzes, die dem PIM entspricht.

Bei der Abbildung der Spezifikation einer Schnittstelle in das PIM stellen sich aber vergleichbare Probleme, wie bei deren Abbildung auf eine andere Komponentenarchitektur. Diese Probleme wurden für Brücken in Kapitel 2.3.2 untersucht und führten dort zu dem Schluss, dass Brücken nur ein Kompromiss sein können und daher für Webkomponenten nicht zweckmäßig sind. Somit ist auch die MDA kein geeignetes Mittel, Webkomponenten mit Hilfe exsistierender Komponentenarchitekturen zu realisieren.

2.3.4 Schlussfolgerung

Die Ausführungen zeigen, dass Komponenten exsistierender Komponentenarchitekturen nicht als Webkomponenten geeignet sind. Es wird keinen Konsens auf eine der Komponentenarchitekturen geben. Die Verwendung unterschiedlicher Komponentenarchitekturen würde aber erfordern, diese transparent über Brücken zu verbinden. Ansonsten wäre die Interoperabilität zwischen in unterschiedlichen Komponentenarchitekturen erstellen Webkomponenten in Frage gestellt. Brücken können das aber wegen der unterschiedlichen Objektmodelle und Paradigmen der Komponentenarchitekturen nicht leisten, auch nicht unter Verwendung der MDA. Daher wird für Webkomponenten eine eigene, neue Architektur benötigt.

2.4 Architektur für Webkomponenten

Eine Architektur für Webkomponenten befindet sich beim World Wide Web Consortium (W3C) [W3C 02a] in der Standardisierung. Das W3C hat die meisten der Technologien des World Wide Webs standardisiert. Über sie gibt es ebenso wie für die Architektur generell einen herstellerübergreifenden Konsens.

Auf dieser grundlegenden Architektur aufbauend können speziellere spezifiziert werden, mit der ihr Regeln zugefügt werden. Ein Beispiel ist die Global XML Architecture (GXA) [Box 02] von Microsoft. Sie fügt Regeln hinzu, um die Modularität von infrastrukturspezifischen Protokollen zu erreichen. Außerdem wird eine Menge solcher Protokolle spezifiziert. Da für diese Arbeit die Modularität infrastrukturspezifischer Protokolle nicht relevant ist, wird auf eine genauere Betrachtung der GXA verzichtet. Die Arbeit baut auf der grundlegenden Architektur des W3C auf.

Die Architektur soll nachfolgend jedoch nicht sofort beschrieben werden. Statt dessen wird begründet, warum die Architektur in ihrer Form entwickelt wurde. Das schafft die Voraussetzungen für die Untersuchungen dieser Arbeit. Die beiden wesentlichen Standards des W3C für Webkomponenten werden erst in den beiden nachfolgenden Kapiteln 3 und 4 im Detail beschrieben.

Eine zentrale Forderung ist, dass unterschiedliche Hersteller für die Architektur miteinander interoperable Infrastrukturen entwickeln können. Die Architektur muss das fördern. Dazu ist wichtig, den Schwerpunkt auf Schnittstellen zu legen. Wie über eine Schnittstelle auf einen Webservice zugegriffen und wie sie beschrieben wird, muss von der verwendeten Infrastruktur unabhängig sein.

Schnittstellen von Webkomponenten sollten so modelliert sein, dass sie leicht verständlich und zu realisieren sind. Das erleichtert die Interoperabilität zwischen unterschiedlichen Infrastrukturen. Analog wird dazu im Anforderungskatalog des W3C zur Webservice-Architektur [Austin 02] festgelegt, dass kein Programmiermodell ausgeschlossen werden soll. Das gilt auch für Programmiermodelle, die nicht auf objektorientierten Konzepten beruhen. Solche nicht auszuschließen, wird erleichtert, wenn für die Modellierung der Schnittstellen keine objektorientierten Konzepte verwendet werden.

Die Verwendung objektorientierter Konzepte würde viel Raum für unterschiedliche Interpretationen lassen, die zu inkompatiblen Infrastrukturen führen können. Details des verwendeten Objektmodells müssten in den Schnittstellen befolgt werden. Kapitel 2.3.2 hat aufgezeigt, wie wichtig solche Details für die Interoperabilität sind.

Besser ist es, für Schnittstellen einfache Modelle zu verwenden, die auf Operationen beruhen. Operationen sind als Prozeduren, Funktionen oder auch Methoden in fast allen Programmiermodellen vorhanden. Die Semantik von Operationsaufrufen ist allgemein gut verstanden. Webkomponenten sollten daher Schnittstellen haben, die Mengen von Operationsaufrufen sind.

Die Spezifikation der Schnittstellen sollte sich direkt daran orientieren, wie die Kommunikation zwischen den Webkomponenten stattfindet. Ansonsten wäre eine Abbildung der Spezifikation auf die tatsächliche Kommunikation notwendig, was eine Quelle für Inkompatibilitäten unterschiedlicher Implementierungen ist. Die Kommunikation über das Internet findet über Nachrichten statt. Mit ihnen können Operationsaufrufe realisiert werden. Daher sollten Spezifikationen von Schnittstellen im Sinne ausgetauschter Nachrichten formuliert sein.

Ein Aufruf einer Operation entspricht zwei Nachrichten. Die erste (Request) wird vom Webclient an den Webservice geschickt. Sie enthält alle Parameter der Operation, die in dieser Richtung übertragen werden, um die Operation auszuführen. Nach ihrer Ausführung wird die zweite Nachricht (Response) zurückgesendet, die alle Parameter enthält, die vom Webservice zum Webclient übertragen werden. Das ist in Abbildung 5 veranschaulicht. Protokolle die diesem Kommunikationsmuster folgen, werden als Request-/Response-Protokolle (request-response protocol) bezeichnet. Über solche kommunizieren Webkomponenten.

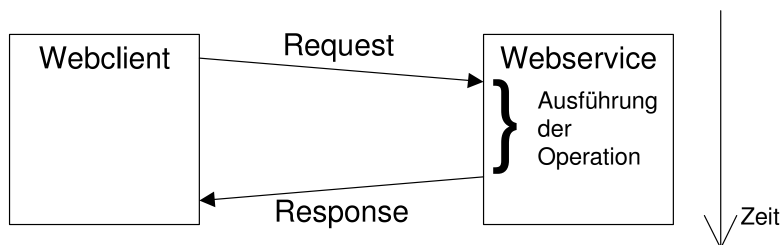

Abbildung 5: Aufrufe einer Operation mit Request-/Response-Protokoll

Damit sind Schnittstellen nicht mehr unteilbare Typen, die Operationen enthalten. In [Box 00a] wird festgestellt, dass Schnittstellen dann Sammlungen von Nachrichtentypen sind, die eine Request-/Response-Semantik (request-response semantics) haben. In diesem Sinne werden Schnittstellen von Webservices spezifiziert, also als anwendungsspezifische Request-/Response-Protokolle, mit denen über Schnittstellen kommuniziert wird.

2.4.1 Webanwendungen als Webkomponenten

Das World Wide Web ist der heute am häufigsten verwendete Weg, Dienste im Internet anzubieten. Es wurde ursprünglich als internetbasierter, verteilter Hypertext entwickelt, der auf Webservern bereitgestellt und auf den mit Webbrowsern zugegriffen wird. Sowohl Webbrowser als auch Webserver können von unterschiedlichen Herstellern erstellt und auf unterschiedlichen Plattformen ausgeführt werden und sind trotzdem miteinander interoperabel.

Der Idee des Hypertextes wurden später Dienste zugefügt, die eine Verarbeitung auf Serverseite zuließen. Solche serverbasierten Webanwendungen verwenden den Webbrowser als eine Art universelles Terminal für ihre Benutzeroberfläche. Zunächst wurden einfach Webanwendungen erstellt, wie Gästebücher oder Suchmaschinen, später immer komplexere. So stellt heute z. B. auch SAP [SAP 02a] mit dem Internet Transaction Server (ITS) eine Möglichkeit zur Verfügung, SAP R/3 per Webbrowser zu bedienen [SAP 99].

Auch wenn es heute möglich ist, Programmteile clientseitig im Webbrowser auszuführen, ist das Kommunikationsprinzip zwischen Webbrowser und Webserver gleich geblieben. Das Hypertext Transport Protokoll (HTTP) [Fielding 99] wird zur Kommunikation verwendet. Nach diesem baut der Webbrowser eine Verbindung zum Webserver auf, fordert eine gewünschte Webseite an, die der Webserver dann an den Webbrowser zurückschickt. HTTP ist also ein Request-/Response-Protokoll. Die Webseite ist in der Hypertext Markup Language (HTML) [Raggett 99] kodiert, die sowohl die informationstragenden Inhalte als auch die zur Darstellung benötigten Formatierungsinformationen enthält.

Viele Webanwendungen bieten heute interessante Dienste an. Hierzu gehörten Dienste wie Babelfish [AltaVista 03], mit dem Texte in verschiedene Sprachen übersetzt werden können oder SMS.DE [Netzquadrat 03], der es erlaubt SMS-Nachrichten an Mobiltelefone („Handys“) zu schicken. Solche Dienste sind nicht nur für die Benutzung mit Webbrowsern als universellem Client hilfreich. Ihre Funktionalität könnte auch in Softwaresystemen wiederverwendet werden. So könnte das Übersetzen mit Babelfish in Textverarbeitungen integriert werden. Das Versenden von SMS-Nachrichten könnte von Administrationswerkzeugen verwendet werden, um einen Administrator bei wichtigen Ereignissen zu informieren. Damit würde eine Webanwendung zu einer Art Webservice, die anderen Webkomponenten ihren Dienst bereitstellt.

Allerdings ist das Integrieren von Webanwendungen in Softwaresysteme mit Schwierigkeiten verbunden. Diese ergeben sich vor allem aus der Tatsache, dass HTML verwendet wird, um Informationen an den Webbrowser zu schicken. Wird die Webanwendung als Webservice verwendet, muss ihr Client die benötigten Daten aus der in HTML kodierten Webseite extrahieren. Das wird dadurch erschwert, dass HTML neben den informationstragenden Inhalten auch die Formatierungsinformationen enthält. Vor allem ist es ein Problem, wenn der Betreiber der Webanwendung die Formatierung ändert. Das geschieht häufig ohne Ankündigung, weil es für den Zugriff mit Webbrowsern keine Rolle spielt. Der Client würde dann jedoch nicht mehr korrekt arbeiten. Er müsste an die neue Formatierung angepasst werden.

Obwohl also Webanwendungen ein guter Weg sind, Dienste im Internet bereitzustellen, so dass Menschen diese mit einem Webbrowser benutzen können, sind sie als Webservices ungeeignet. Würden die informationstragenden Inhalte jedoch in einer anderen Form ohne Formatierungsinformationen kodiert, die für maschinelle Verarbeitung geeignet ist, könnten sie als Webservice verwendet werden.

Implementierungen für das gut verstandene HTTP stehen auf unterschiedlichsten Plattformen zur Verfügung. Durch die Verwendung im World Wide Web gibt es viel Erfahrung mit dem Protokoll, so dass die Implementierungen miteinander interoperabel sind. Aus dem gleichen Grund stehen auch sehr effiziente Implementierungen von HTTP-Servern zur Verfügung, die auch für Webservices nutzbar wären.

Daher wird auch für die Kommunikation zwischen Webclients und Webservices HTTP verwendet. Webservices sind in gewisser Weise Webanwendungen, die Daten statt in HTML in einer für die maschinelle Verarbeitung geeigneten Form kodieren.

2.4.2 XML statt HTML

Für die Interoperabilität der Webkomponenten spielt es eine wichtige Rolle, in welcher für die maschinelle Verarbeitung geeigneten Form sie ausgetauschte Nachrichten kodieren. Anforderungen bei der Wahl sind:

- Die Kodierung sollte über Netzwerke austauschbare Nachrichten liefern.
- Die Kodierung sollte zwischen möglichst vielen Plattformen austauschbar sein.
- Die Kodierung sollte in Industrie und Forschung breit akzeptiert sein.
- Für die Kodierung sollte auf möglichst vielen Plattformen eine Softwareunterstützung zur Verfügung stehen.
- Die Kodierung sollte erlauben, komplexe Datenstrukturen zu modellieren.

Diese Anforderungen erfüllt die unter ihrer Abkürzung XML bekannt gewordenen Extensible Markup Language. In dieser Arbeit werden XML und die dazugehörigen Konzepte und Technologien nicht vorgestellt, sondern lediglich wichtige Themen kurz angesprochen. Ausreichende Kenntnisse werden dem Leser unterstellt. Ansonsten bieten [Skonnard 00], [RefsnesData 02] und [Box 00a] Einführungen.

XML ist eine Familie von anwendungsspezifischen Sprachen, die gemeinsame Grundregeln befolgen. Die Standardisierung von XML und den dazugehörigen Konzepten und Technologien findet weitgehend durch das W3C statt. Es hat hierzu eine Vielzahl von Standards verabschiedet, die jeweils unterschiedliche Aspekte berücksichtigen. Die Standards sind weitgehend zueinander orthogonal, legen also jeweils einen Aspekt von XML fest und ergänzen sich gegenseitig. Diese Form der Standardisierung erleichtert deren Implementierung und auch die Einarbeitung in diese Technologien, weil immer nur relevante Standards berücksichtigt werden müssen. Außerdem können so für neue Konzepte und Technologien neue Standards zugefügt werden, ohne dass die alten ungültig werden. Die Vorgehensweise führt also zu einer langfristigen Stabilität der Standards.

Die Grundregeln der XML-Sprachfamilie bilden vor allem einen Rahmen zur Modellierung von Daten. Sie sind explizit als „XML Information Set“ (XML-Infoset) in [Cowan 01] standardisiert. Die Modellierung erfolgt hierarchisch. Die Hierarchie wird vor allem durch sogenannte Elemente gebildet, die benannt sind und benannte Attribute haben können. Attribute sowie Textknoten, die Blätter der Hierarchie sind, enthalten die zu kodierenden Daten.

Sowohl Elemente als auch Attribute werden über Namen identifiziert. Um die Eindeutigkeit der Namen zu erreichen, werden sie in Namensräumen organisiert, so dass der Bezeichner des Namensraumes zusammen mit dem sogenannten lokalen Namen (local name) eindeutig ist. Als Bezeichner werden für Namensräume URIs verwendet. Eine URI ist nach [Berners-Lee 98] eine kompakte Zeichenkette, die eine abstrakte oder physikalische Ressource identifiziert. Sie ist weltweit eindeutig. Um Namen kürzer zu schreiben, werden die URIs an kurze Präfixe gebunden (z. B. `env`), die zusammen mit den lokalen Namen notiert werden (z. B. `env:Envelope`).

Obwohl es für XML grundsätzlich nicht erforderlich ist, wird für jede URI eines Namensraumes in dieser Arbeit immer das gleiche Präfix verwendet. Außer in Beispielen für XML-Dokumente geschieht das auch im Text, wenn Namen von Elementen oder Attributen angegeben werden. Für eine Liste der verwendeten Präfixe sei auf Anhang B verwiesen. Außerdem werden sie im Text wie eine Abkürzung eingeführt.

XML-Dokumente können in eine Folge von Zeichen serialisiert werden. Die syntaktischen Regeln, die üblicherweise zur Serialisierung verwendet werden, sind in [Bray 00] standardisiert. Da sie vor dem „XML Information Set“ standardisiert wurden, werden sie häufig als der Kern von XML betrachtet. In der serialisierten Form können XML-Dokumente zwischen Webkomponenten über ein Netzwerk ausgetauscht werden.

Alle Daten sind in XML in Zeichenketten kodiert. Der verwendete Zeichensatz ISO/IEC 10646 ist von der ISO [ISO 03] standardisiert und hat große Ähnlichkeiten mit Unicode [Unicode 03]. Neben lateinischen Zeichen enthält er auch arabische und fernöstliche. Somit können auch Texte dieser Sprachen in XML abgelegt werden. Bei der Serialisierung von XML-Dokumenten dürfen die Zeichen auch in anderen Zeichensätzen kodiert sein. Welcher Zeichensatz verwendet wird, kann in der Serialisierung angegeben werden. Für die Interoperabilität müssen zumindest die beiden Zeichensätze UTF-8 [Yergeau 98] und UTF-16 [Hoffman 00] unterstützt werden. Sollte der Zeichensatz einige Zeichen aus ISO/IEC 10646 nicht enthalten, gibt es eine Notation, diese als Zeichencode anzugeben.

Die Tatsache, dass alle Daten in XML als Zeichenketten kodiert sind und für die Serialisierung Standardzeichensätze vorhanden sind, macht es einfach, XML-Dokumente zwischen unterschiedlichen Plattformen auszutauschen. Anders als bei binären Kodierungen, die sich bei unterschiedlichen Plattformen z. B. bei der Kodierung von Zahlen stark unterscheiden, sind textuelle Darstellungen immer gleich. Lediglich Zeichensätze können sich unterscheiden. Mit UTF-8 und UTF-16 existieren für Serialisierungen von XML-Dokumenten jedoch Zeichensätze, die immer vorhanden sind. Somit erlaubt die Verwendung von XML für zwischen Webkomponenten ausgetauschte Nachrichten deren Austauschbarkeit zwischen Plattformen.

XML-Dokumente können nicht nur prinzipiell auf jeder Plattform gelesen werden. Wegen der breiten Akzeptanz von XML in Industrie und Forschung steht für die Verwendung von XML auf den meisten Plattformen eine leistungsfähige Softwareunterstützung zur Verfügung. Einen wichtigen Beitrag liefert dabei Java. Viele Werkzeuge für XML sind in Java implementiert und damit auch auf allen Plattformen verfügbar, auf denen Java verwendet werden kann. Auch Microsoft stellt für seine Komponentenarchitekturen COM und .NET Komponenten für die Verwendung von XML bereit.

Zur Softwareunterstützung gehören XML-Prozessoren. Sie enthalten XML-Parser und XML-Writer, die Serialisierungen von XML-Dokumenten lesen bzw. erzeugen können. XML-Prozessoren haben Programmierschnittstellen, um auf XML-Dokumente zuzugreifen. Neben proprietären Programmierschnittstellen gibt es auch zwei standardisierte, die von vielen XML-Prozessoren zur Verfügung gestellt werden: Das Document Object Model (DOM) [LeHors 02] und die Simple API for XML (SAX) [SAX 02].

XML-Prozessoren können auch weitere Werkzeuge enthalten. Diese können aber auch unabhängig von XML-Prozessoren verbreitet werden. Standardisiert wurden z. B. die XML Path Language (XPath) sowie XSL Transformations (XSLT). Mit Ausdrücken der Sprache XPath [Clark 99a] können Teile von XML-Dokumenten spezifiziert werden. XPath-Ausdrücke werden in dieser Arbeit auch im Text verwendet, um eindeutig anzugeben, welche Elemente oder Attribute in einem XML-Dokument gemeint sind. Das wird genauer in Anhang A beschrieben. XSLT [Clark 99b] ist eine Sprache zur Transformation von XML-Dokumenten in HTML- oder Textdokumente oder in XML-Dokumente anderer Struktur.

Parameter von Operationen, die in Webservices aufgerufen werden, können nicht nur einfache Datentypen haben. Parameter können auch komplex strukturiert sein, also z. B. in einer Sprache wie Pascal aus Arrays und Records bestehen. Da die zwischen Webkomponenten ausgetauschten Nachrichten solche Parameter enthalten, muss es auch möglich sein, dass Nachrichten komplex strukturiert sind. Sind Nachrichten serialisierte XML-Dokumente, kann hierzu die hierarchische Struktur von XML verwendet werden.

Wie ein XML-Dokument strukturiert sein darf, wird mit der Sprache XML-Schema in [Thompson 01] und [Brion 01] festgelegt. In ihr wird ein Vokabular aus Attributen, Elementen sowie deren Typen definiert. Für Elemente kann angegeben werden, welche Kindelemente und Attribute es haben darf. Durch Verwendung von vordefinierten sowie benutzerdefinierten Typen fungiert XML-Schema gleichzeitig als Typsystem für XML. Es wird in Kapitel 4.2 detailliert vorgestellt.

Sind zwischen Webkomponenten ausgetauschte Nachrichten XML-Dokumente, kann XML-Schema auch verwendet werden, um die Struktur der Nachrichten zu spezifizieren. Das ist notwendig, um ihre Schnittstellen zu spezifizieren, da Schnittstellen im Falle von Webkomponenten Sammlungen von Nachrichtentypen sind, wie am Anfang von Kapitel 2.4 festgestellt.

XML-modellierte Datenstrukturen werden nicht nur für Nachrichten zwischen Webkomponenten verwendet. Viele Anwendungen verwenden sie auch für andere Zwecke, z. B. zum persistenten Ablegen von Daten in Dateisystemen oder Datenbanken. Auch hier wird das Vokabular zur Strukturierung in XML-Schema definiert. Interessant ist die Beobachtung, dass so strukturierte Daten einer Anwendung direkt in einer Nachricht zwischen Webkomponenten übertragen werden können. Auch die Spezifikation der Daten der Anwendung in XML-Schema kann dann direkt bei der Spezifikation der Nachricht und damit der Schnittstelle des Webservices wiederverwendet werden.

2.4.3 Wozu wird SOAP benötigt?

Aus den genannten Gründen wird die Kombination von XML und HTTP zur Modellierung, Serialisierung und Übertragung zwischen Webkomponenten ausgetauschter Nachrichten verwendet. Zusätzliche Konventionen sind jedoch zweckmäßig, um einheitlich mit Nachrichten umgehen zu können und eine voneinander unabhängige Fortentwicklung der Webkomponenten zu ermöglichen. Sie sind im Protokoll SOAP enthalten, das sich beim W3C in der Standardisierung befindet und in [Gudgin 02a] und [Gudgin 02b] beschrieben wird. Nachrichten, die diesen Konventionen entsprechen, werden als SOAP-Nachrichten bezeichnet.

Damit einheitlich mit SOAP-Nachrichten umgegangen werden kann, sind Konventionen über ihre Struktur hilfreich. Das führt zu einheitlicheren Schnittstellen und erlaubt zu erkennen, ob es sich überhaupt um eine SOAP-Nachricht handelt. Dabei kann z. B. auch festgelegt werden, wie ein Webservice aus dem Request erkennt, welche Operation er aufrufen soll.

Die Struktur von SOAP-Nachrichten sollte es erlaubten, dass Webclients und Webservices unabhängig voneinander weiterentwickelt werden können. Eine ältere Version eines Webclients soll möglichst mit einer neueren Version eines Webservices kommunizieren können und umgekehrt, auch wenn die neue Version zusätzliche Daten in den XML-Nachrichten austauschen kann. Außerdem müssen Webservices, die den gleichen Dienst erbringen von unterschiedlichen Softwareherstellern implementiert werden können, so dass sie mit den gleichen Webclients verwendet werden können. Solche Webservices können über Standardschnittstellen hinausgehende Eigenschaften haben.

In [Box 00a] wird außerdem festgestellt, dass es häufig wünschenswert ist, in der SOAP-Nachricht Aspekte des Ausführungskontextes des Senders mitzuübertragen. Hierzu gehören z. B. Informationen über den Benutzer oder auch über einen Transaktionskontext. Solche Daten sind von der Ausführungsumgebung der Webkomponenten abhängig. In der Regel sind sie unabhängig von den Parametern der Operation. Daher sollten sie unabhängig von diesen behandelt, aber trotzdem in derselben SOAP-Nachricht übertragen werden.

Für diese Fälle ist es erforderlich, dass ein Sender SOAP-Nachrichten Daten zufügen kann, die nicht in der Spezifikation der Schnittstelle spezifiziert sind. Trotzdem soll der Empfänger eine erweiterte SOAP-Nachricht auch interpretieren können, wenn er zugefügte Daten nicht erwartet. Das ist nur möglich, wenn über das Zufügen von Daten geeignete Konventionen existieren.

Hilfreich sind auch Konventionen, wie Fehler in SOAP-Nachrichten angezeigt werden. Das erlaubt Infrastrukturen für Webkomponenten, stets Fehler erkennen und auf sie einheitlich reagieren zu können. Z. B. ermöglicht es, Fehler auf die Fehlerbehandlungsmechanismen der Infrastruktur abzubilden.

Für die Kommunikation zwischen Webkomponenten werden also nicht nur HTTP und XML benötigt, sondern darüber hinaus unterschiedliche Konventionen. Diese sind im Protokoll SOAP enthalten, das in Kapitel 3 im Detail beschrieben wird.

2.5 Spezifikation von Webkomponenten

Neben Konventionen, wie Webkomponenten miteinander kommunizieren, wird eine Sprache benötigt, mit der ihre Schnittstellen spezifiziert werden können. Bereits in Kapitel 2.2 wurde festgestellt, dass solche Spezifikationen möglichst präzise sein sollten und über das Internet verbreitet werden müssen, damit sie den Entwicklern von Webclients zur Verfügung stehen. Für die Spezifikation von Webservices befindet sich beim W3C die Sprache Web Services Description Language (WSDL) [Chinnici 02] in der Standardisierung.

Zur Spezifikation von Schnittstellen von Webkomponenten wurde bereits in Kapitel 2.4 ausgeführt, dass sie sich möglichst direkt daran orientieren sollte, wie die Kommunikation über das Internet stattfindet. Würden für die Spezifikation andere Paradigmen verwendet, müsste sie erst auf die tatsächliche Kommunikation abgebildet werden. Unterschiedliche Auslegungen der Abbildung wären dann eine Gefahr für die Interoperabilität.

In Kapitel 2.4 bzw. [Box 00a] wurde festgestellt, dass Schnittstellen in diesem Sinne Sammlungen von Nachrichtentypen mit Request-/Response-Semantik sind. Solche Schnittstellen zu spezifizieren, erfordert zumindest die Beschreibung

- der einzelnen Nachrichtentypen,
- der Zuordnung von zwei Nachrichtentypen zu einem Request-/Response-Paar und
- der Zuordnung von Request-/Response-Paaren zu einer Schnittstelle.

Bei SOAP-Nachrichten ist jedoch nicht die Beschreibung der gesamten Nachricht erforderlich, sondern nur von Teilen. Die Grundstruktur von SOAP-Nachrichten ist bereits im SOAP-Standard festgelegt und muss nicht erneut beschrieben werden. Beschrieben werden müssen nur die anwendungsspezifischen Teile. Zusätzlich ist es sinnvoll, Abhängigkeiten zwischen diesen Teilen spezifizieren zu können.

Neben der Beschreibung der einzelnen SOAP-Nachricht, muss festgelegt werden, welche SOAP-Nachrichten zusammen gültige Request-/Response-Paare bilden und somit einen Operationsaufruf repräsentieren. Dabei muss für einen Nachrichtentyp von Requests nicht unbedingt stets der gleiche Nachrichtentyp von Responses verwendet werden. Es können mehrere möglich sein. Insbesondere in Fehlersituationen sendet der Webservice als Response eine Fehlernachricht.

Die einzelnen durch Request-/Response-Paare beschriebenen Operationen bilden zusammen eine Schnittstelle. Die Spezifikation der Schnittstelle muss also enthalten, welche Operationen zur ihr gehören.

Im Kontext von WSDL werden zwei Arten von Schnittstellen unterschieden: abstrakte und gebundene. Bei abstrakten Schnittstellen werden Nachrichten unabhängig davon beschrieben, wie sie kodiert sind und mit welchem Protokoll sie ausgetauscht werden. Dort ist also nicht bekannt, dass SOAP, XML oder HTTP verwendet wird. Das ist erst in einer gebundenen Schnittstelle bekannt, die auf Basis einer abstrakten definiert wird.

Für die Spezifikation eines Webservices ist die Beschreibung seiner Schnittstellen von zentraler Wichtigkeit. Darüber hinaus benötigten Entwickler aber noch weitere Informationen, um auf den Webservice zugreifen zu können. Auch diese müssen in der Spezifikation des Webservices enthalten sein. Zusätzlich zur Schnittstellenbeschreibung muss zumindest beschrieben werden

- welche Netzwerkadressen der Webservice hat und
- über welche Schnittstellen der Webservice seine Dienste bereitstellt.

Der Webclient muss die Netzwerkadresse des Webservices kennen, damit er eine Verbindung zu ihm aufbauen kann. Unterschiedliche gebundene Schnittstellen eines Webservices können über unterschiedliche Netzwerkadressen erreichbar sein. Welche Schnittstellen ein Webservice hat, muss ein Entwickler aus der Beschreibung des Webservices ermitteln können.

Die Spezifikation des Webservices muss in einer Form gespeichert sein, die der Entwickler lesen kann. Ebenso sollte sie auch von Werkzeugen verwendet werden können, um daraus z. B. kommunikationsrelevante Teile des Programmcodes von Webservice und Webclients generieren zu können. Das erfordert, dass die Spezifikation in einer maschinenverständlichen Form vorliegt, was bei Dokumenten der Sprache WSDL (WSDL-Dokumente) der Fall ist.

2.6 Beispiel für eine Webkomponente

Zusammengefasst sind Webkomponenten also Komponenten, bei denen jedoch nicht ihre Software verkauft, sondern ihr Dienst über das Internet angeboten wird. Durch Zugriff über das Internet können Webkomponenten trotzdem in Softwaresysteme integriert werden. Zur Kommunikation wird hierzu das Protokoll SOAP verwendet, das Konventionen enthält, wie in XML kodierte Nachrichten in der Regel mit HTTP ausgetauscht werden. Spezifikationen von Webkomponenten müssen über das Internet verbreitet werden können. Hierzu werden heute WSDL-Dokumente verwendet. In ihnen werden vor allem die Schnittstellen der Webkomponente beschrieben, die als Mengen von Operationen modelliert sind, wobei jede Operation als Request-/Response-Paar beschrieben wird.

Um die Idee der Webkomponenten zu veranschaulichen, soll nachfolgend ein konkretes Beispiel gegeben werden. Es soll in der gesamten Arbeit zur Illustration vorgestellter Konzepte dienen. Aufgabe der vorgestellten Webkomponente ist, Artikel einer Internetzeitung zu verwalten. Hierzu wird sie im Internet bereitgestellt und kann von verschiedenen Webclients von beliebigen Orten über das Internet zugegriffen werden.

Autoren können so an beliebigen Orten ihre Artikel erstellen. Hierzu haben sie ein Autorensystem, das ein Webclient ist. Das Autorensystem unterstützt Autoren bei der Erstellung und Verwaltung der Artikel. Es verwendet die Webkomponente, um die Artikel zu speichern. Über die Webkomponente greift auch die Redaktion auf Artikel zu. Hierzu wird dort ein weiterer Webclient verwendet, das Redaktionssystem. Mit ihm können Artikel weiterbearbeitet und für die Veröffentlichung freigegeben werden.

Auch Leser der Internetzeitung können mit Webclients auf die Webkomponente zugreifen. Sie können von ihr die veröffentlichten Artikel abfragen. Der Webclient könnte z. B. stets die Titel der neuesten Artikel anzeigen. Damit die Leser auch mit einem Webbrowser die Internetzeitung lesen können, wird zusätzlich ein Programm benötigt, dass periodisch als Webclient alle Artikel von der Webkomponente abfragt und hieraus eine Website generiert. Für jeden Artikel könnte sie eine Webseite generieren, die den Artikel anzeigt. Sie könnte aufgebaut sein, wie in Abbildung 6 veranschaulicht.

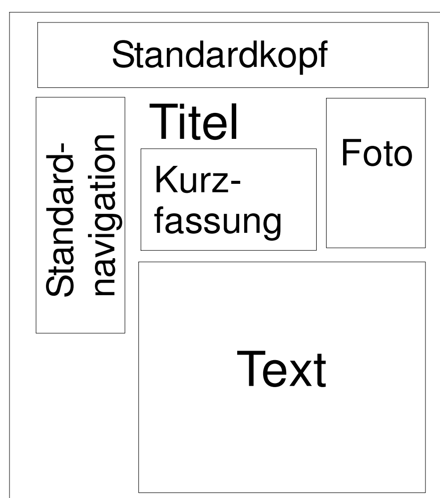

Abbildung 6: Möglicher Aufbau einer Webseite der Internetzeitung

Ein Webclient der Webkomponente für eine Internetzeitung kann auch selbst Dienste als Webservice bereitstellen. Es könnte z. B. eine Suchmaschine sein, die die Suche nach Artikeln erlaubt und diesen Dienst selbst als Webservice zur Verfügung stellt.

Umgekehrt könnte die Webkomponente für eine Internetzeitung auch mit Hilfe anderer Webkomponenten realisiert sein. Diese könnten z. B. eine Benutzerverwaltung bereitstellen oder den Zugriff auf eine Datenbank erlauben.

Die Webkomponente für eine Internetzeitung legt ein Modell fest, wie jeder Artikel aufgebaut ist. Für jeden Artikel wird eine Reihe von Feldern verwaltet:

- `MessageID`: Eine Zeichenkette, die den Artikel eindeutig bezeichnet.
- `Category`: Rubrik der Internetzeitung, zu der der Artikel gehört.
- `Title`: Titel des Artikels.
- `Abstract`: Kurzfassung des Artikels.
- `Date`: Datum, an dem der Artikel veröffentlicht wurde.
- `ImageURL`: URL eines Fotos, dass mit dem Artikel angezeigt werden soll.
- `Text`: Text des Artikels.
- `Public`: Zeigt an, ob der Artikel bereits von der Redaktion veröffentlicht wurde.

Auf die Webkomponente kann über eine Schnittstelle zugegriffen werden, die verschiedene Operationen enthält. Mit der Operation `Login` muss sich ein Benutzer an der Webkomponente anmelden und später mit der Operation `Logout` wieder abmelden. Der Operation `Login` wird der Name des Benutzers sowie ein Passwort als Parameter übergegeben. Beides ist im Request enthalten. Abhängig vom angegebenen Namen des Benutzers sind später unterschiedliche Rechte für den Zugriff auf die Artikel gültig. Wird als Benutzername „Gast“ angegeben, ist kein Passwort erforderlich. Dann ist es jedoch nur möglich veröffentlichte Artikel zu lesen. Webclients der Leser der Internetzeitung melden sich mit diesem Benutzernamen an.

War das angegebene Passwort nicht korrekt, liefert die Operation `Login` als Response eine Fehlernachricht. Ansonsten enthält der Response die sogenannte Session-ID, die nachfolgend bei allen Operationsaufrufen angegeben werden muss, um zu bezeichnen, um welchen Benutzer es sich handelt. Sie wird mit dem Aufrufen der Operation `Logout` ungültig.

Autoren von Artikeln melden sich mit ihrem Namen an. Ihr Webclient kann danach mit der Operation `Create` einen neuen Artikel erstellen oder mit den Operationen `ChangeCategory`, `ChangeTitle`, `ChangeAbstract`, `ChangeText` oder `ChangeImage` jeweils ein Feld eines Artikels verändern. Mit der Operation `Delete` kann ein Artikel gelöscht werden. Der Webclient kann mit der Operation `List` eine Liste von Artikeln abfragen. Dabei kann er auch die Höchstzahl der gelieferten Artikel beschränken oder nur Artikel einer bestimmten Rubrik abfragen. Die Rubriken der Internetzeitung liefert die Operation `Categories`. Schließlich können die Felder von Artikeln mit der Operation `Get` abgefragt werden.

Auch Webclients der Redaktion und der Leser der Internetzeitung verwenden die eben beschriebenen Operationen. Die Leser können jedoch nur mit der Operation `Categories` die Rubriken, mit der Operation `List` veröffentlichte Artikel und mit der Operation `Get` deren Felder abfragen. Die Redaktion hat zusätzlich zu den Operationen der Autoren die Operationen `Publish` und `Unpublish`, mit denen sie einen Artikel veröffentlichen oder die Veröffentlichung zurücknehmen kann.

Um Probleme mit nebenläufigen Zugriffen auf Artikel zu vermeiden, können Artikel gesperrt werden. Hierzu kann im Request der Operation `Get` angegeben werden, dass alle abgefragten Artikel gesperrt werden sollen. Artikel können dann nicht von anderen Benutzern gesperrt oder verändert werden. Das Lesen ist aber weiterhin möglich. Es handelt sich also um Schreibsperren. Für eine Menge von Artikeln können gesetzte Sperren mit der Operation `Unlock` freigegeben werden.

2.7 Stabilität von Softwaresystemen aus Webkomponenten

Diese Arbeit beschäftigt sich mit der Frage, wie die Stabilität von Softwaresystemen aus Webkomponenten, wie der für die Internetzeitung, sichergestellt werden kann. In Kapitel 1.2 wurden dazu Teilziele formuliert, die nachfolgend noch einmal mit dem in diesem Kapitel vermittelten Verständnis über Webkomponenten beleuchtet werden sollen.

Webkomponenten können in unterschiedlichen Unternehmen entwickelt werden, wobei unterschiedliche Werkzeuge zum Einsatz kommen können. Das gilt auch für die Webkomponente für eine Internetzeitung und ihre im Kapitel 2.6 vorgestellten Webclients. Z. B. können unterschiedliche Autorensysteme implementiert werden, die die Schnittstelle der Webkomponente für die Internetzeitung unterstützen. Unterschiedliche Internetzeitungen können von unterschiedlichen Herstellern Webkomponenten implementieren lassen, die die Schnittstelle haben. Die Webkomponenten können auf unterschiedlichen Plattformen ausgeführt werden. Trotzdem sollen die Webkomponenten miteinander interoperabel sein und zusammen ein stabiles Softwaresystem bilden.

Das ist nur möglich, wenn die Standards für Webkomponenten präzise formuliert sind und nicht zu viele Optionen lassen. Ansonsten haben die Hersteller einen Interpretationsspielraum, der zu Interoperabilitätsproblemen führen kann. Die Tatsache, dass die Entwickler in unterschiedlichen Unternehmen sind, erschwert, dass sie sich ein gemeinsames Verständnis über die Standards schaffen. Außerdem erschwert es, die Webkomponenten gegeneinander zu testen. Daher wurde schon in Kapitel 1.2 für diese Arbeit zum Ziel gesetzt zu untersuchen, ob die Standards selbst präzise genug formuliert sind.

Auch Interpretationsspielräume in Spezifikationen von Schnittstellen können zu Interoperabilitätsproblemen führen. Daher müssen Schnittstellen präzise spezifiziert sein. Das verwendete Spezifikationsverfahren muss das erlauben. Ansonsten besteht die Gefahr, dass Webkomponenten nicht zueinander interoperabel sind, obwohl sie die Spezifikation erfüllen. Z. B. könnten Webkomponenten für Internetzeitungen und für diese benötigte Autorensysteme mit derselben Schnittstelle von unterschiedlichen Herstellern implementiert werden. Dann wäre es schwierig alle Paare von Webclients und Webservices gegeneinander zu testen. Daher muss ihre Interoperabilität schon dadurch sichergestellt werden, dass sie der Spezifikation entsprechen.

Ob das mit WSDL, dem heute verwendeten Spezifikationsverfahren für Schnittstellen von Webkomponenten, möglich ist, soll nach Kapitel 1.2 in dieser Arbeit untersucht werden. Gegebenenfalls wäre ein neues, präziseres Spezifikationsverfahren zu entwickeln.

Auch wenn Schnittstellen präzise spezifiziert sind, kann die Stabilität des Softwaresystems dadurch gefährdet werden, dass Webkomponenten nicht der Spezifikation entsprechen. In der Regel sind solche Abweichungen Fehler in der Implementierung einer Webkomponente. Eine mögliche Ursache kann ein fehlerhaftes Verständnis der Spezifikation bei den Entwicklern sein. Das wäre insofern problematisch, weil die sie dann auch beim Testen der Webkomponente den Fehler nicht entdecken können.

Für diese Arbeit wurde daher in Kapitel 1.2 zum Ziel gesetzt, wie Abweichungen zwischen Spezifikationen und Implementierungen erkannt werden können. Benötigt wird ein Verfahren, das automatisch prüft, ob die ausgetauschten SOAP-Nachrichten der Spezifikation entsprechen. Die Prüfung soll als automatische Validation bezeichnet werden. Weicht eine SOAP-Nachricht von der Spezifikation ab, wird ein Fehler angezeigt. So können beim Testen von Webkomponenten Verstöße gegen die Spezifikation unabhängig vom Verständnis der Entwickler erkannt werden.

Beim Testen können trotzdem nur eine begrenzte Zahl von Testfällen untersucht werden. Das verhindert alle Fälle zu erkennen, in denen eine Webkomponente von der Spezifikation abweicht. Um auch nach dem Testen auftretenden Abweichungen zu erkennen, kann die automatische Validation ständig im produktiven Betrieb der Webkomponente stattfinden.

Auch das in Kapitel 1.2 gesetzte Ziel dieser Arbeit, relevante Änderungen von Schnittstellen zu erkennen, kann mit der automatischen Validation erreicht werden. Dazu validiert ein Webclient alle vom Webservice empfangenen Nachrichten gegen die Version der Spezifikation, gegen die er implementiert wurde. Diese Version enthält alle Annahmen über den Webservice, die bei der Implementierung des Webclients berücksichtigt worden sein können. Ändert ein Webservice seine Schnittstelle, ist das für den Webclient irrelevant, solange alle Nachrichten, die er vom Webservices erhält, noch der alten Version der Spezifikation entsprechen. Weichen sie aber ihr ab, können sie Annahmen widersprechen, nach denen der Webclient implementiert wurde, was zu Stabilitätsproblemen führen kann. Mit der automatischen Validation gegen die alte Version der Spezifikation kann ein solches Problem jedoch erkannt und durch Entwickler gelöst werden.

3 SOAP

Webkomponenten kommunizieren miteinander über das Protokoll SOAP. Als Basis für spätere Untersuchungen soll es in diesem Kapitel eingeführt werden. Für die Arbeit relevant ist SOAP in der Version 1.2, die sich aktuell beim W3C [W3C 02a] in der Standardisierung befindet. Verfügbare Dokumente, auf die sich die Arbeit bezieht, sind Arbeitsentwürfe aus dem Juni 2002 und müssen als „Work in Progress" angesehen werden. Es ist jedoch zu erwarten, dass der endgültige Standard nicht wesentlich von ihnen abweichen wird. Daher sollen sie nachfolgend als SOAP-Standard bezeichnet werden.

Als Alternative hätte sich diese Arbeit auf die Version 1.1 von SOAP beziehen können. Die Version wird zwar aktuell in der Industrie als De-facto-Standard verwendet, ist jedoch formal nur eine „W3C Note" und somit im Standardisierungsprozess des W3C von einem Standard weiter entfernt als ein Arbeitsentwurf („W3C Working Draft")[7].

Der SOAP-Standard (Version 1.2) wird in drei Teilen publiziert. Teil 1 [Gudgin 02a] beschreibt SOAP zunächst unabhängig vom verwendeten Protokoll zum Austausch von XML-Nachrichten und ihrer verwendeten Serialisierung. In SOAP verwendete XML-Nachrichten, die im Folgenden als SOAP-Nachrichten bezeichnet werden, sind statt dessen im Sinne des XML-Infosets beschrieben. Außerdem beschreibt Teil 1, wie SOAP-Nachrichten verarbeitet werden müssen und wie im SOAP-Standard nicht enthaltende Features und Abbildungen von SOAP auf konkrete Protokolle zum Austausch von SOAP-Nachrichten spezifiziert werden müssen.

Teil 2 [Gudgin 02b] fügt zum Teil 1 optionale Teile hinzu. Hierzu gehören Konventionen, wie Daten aus Anwendungen und Operationsaufrufe kodiert werden sollten. Außerdem wird hier beschrieben, wie SOAP-Nachrichten mit Hilfe von HTTP übertragen werden können. Neben dem Teil 1 und 2 gehört zum SOAP-Standard noch ein nicht normativer Teil 0 [Mitra 02], der als verständliches Tutorial in SOAP 1.2 einführt.

3.1 Grundbegriffe

Im Kontext von SOAP wird der Teil einer Webkomponente, der für die Verarbeitung des SOAP-Protokolls zuständig ist, als SOAP-Knoten bezeichnet, seine Implementierung als SOAP-Implementierung. Dabei wird die Asymmetrie zwischen Webclient und Webservice zunächst nicht betrachtet. SOAP-Knoten sind gleichrangige Partner (peers), die miteinander SOAP-Nachrichten austauschen. Dazu sendet ein SOAP-Knoten in seiner Rolle als SOAP-Sender Nachrichten zu einem SOAP-Knoten in dessen Rolle als SOAP-Empfänger, wie in Abbildung 7 veranschaulicht. Zur Adressierung werden SOAP-Knoten mit URIs identifiziert.

---

[7] Unterschiede zwischen Version 1.1 und 1.2 von SOAP werden in Kapitel 5 von [Mitra 02] zusammengefasst.

Abbildung 7: Direkter Nachrichtenaustausch zwischen SOAP-Sender und SOAP-Empfänger

Eine Nachricht, die zwischen zwei Webkomponenten ausgetauscht wird, kann auf ihrem Weg von SOAP-Knoten weitergeleitet werden. Solche als SOAP-Intermediaries bezeichneten SOAP-Knoten haben sowohl die Rolle des SOAP-Senders als auch des SOAP-Empfängers. Ein SOAP-Intermediary kann selbst Teile der Nachricht verarbeiten oder auch eine Nachricht umkodieren. Der Weg der SOAP-Nachricht vom ursprünglichen SOAP-Sender über potentiell mehrere SOAP-Intermediaries zum endgültigen SOAP-Empfänger wird als SOAP-Message-Path bezeichnet.

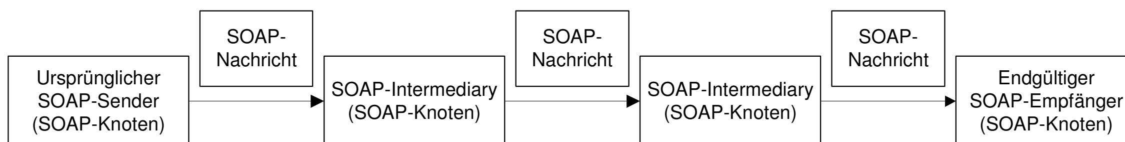

Abbildung 8: SOAP-Message-Path mit zwei SOAP-Intermediaries

Anwendungen werden in vielen Fällen den Austausch mehrerer Nachrichten zu komplexeren Kommunikationsmustern zusammenfügen. Der SOAP-Standard schreibt das aber nicht vor. Er beschreibt jedoch, was in der Spezifikation eines Kommunikationsmusters festgelegt werden muss und definiert selbst zwei Kommunikationsmuster, die beide eine Request-/Response-Semantik haben.

Der SOAP-Standard schreibt auch nicht vor, wie SOAP-Nachrichten zwischen SOAP-Knoten ausgetauscht werden müssen. Auch die Verwendung der üblichen Serialisierung nach XML 1.0 gemäß [Bray 00] ist für SOAP-Nachrichten nicht vorgeschrieben. Statt dessen wird festgelegt, wie Abbildungen von SOAP auf darunter liegende Protokolle definiert werden. Solche Protokollbindungen können dann in unabhängigen Dokumenten spezifiziert werden. HTTP wird als darunter liegendes Protokoll jedoch als so wichtig angesehen, dass für dieses eine Protokollbindung direkt im SOAP-Standard enthalten ist.

3.2 Nachrichtenstruktur

Der SOAP-Standard beschreibt die Struktur ausgetauschter SOAP-Nachrichten im Sinne von XML-Infosets. In dieser Arbeit soll sie trotzdem im Sinne von XML-Dokumenten beschrieben werden, um die Darstellung übersichtlich zu halten. Gemeint sind die durch die XML-Dokumente beschriebenen XML-Infosets. Für SOAP-Nachrichten ist festgelegt, dass sie keine Document-Type-Declarations (DTD) oder Unparsed-Entities enthalten. Auch müssen SOAP-Empfänger enthaltene Processing-Instructions ignorieren.

SOAP-Nachrichten haben das Rootelement `env:Envelope` (Envelope-Element) im Namensraum `http://www.w3.org/2002/06/soap-envelope`, für den in dieser Arbeit das Präfix `env` verwendet wird. Der gleiche Namensraum wird für die meisten qualifizierten

Namen von SOAP 1.2 verwendet. Er ist für den in dieser Arbeit verwendeten Arbeitsentwurf von SOAP 1.2 vom Juni 2002 spezifisch. Andere Versionen von SOAP können an anderen Namensräumen erkannt werden.

Abbildung 9 zeigt ein Beispiel für eine SOAP-Nachricht[8]. Das Envelope-Element hat die beiden Kindelemente `env:Header` und `env:Body`. Das Element `env:Header` (SOAP-Header) ist im Gegensatz zu `env:Body` (Body-Element) optional. Der SOAP-Body enthält die Daten mit der zentralen Semantik der Nachricht. Sie wollte der ursprüngliche SOAP-Sender an den endgültigen SOAP-Empfänger übertragen. Das schließt Daten ein, mit denen Fehler beschrieben werden. Im SOAP-Header sind dagegen Daten enthalten, die außerhalb der Schnittstellenbeschreibung des Webservices vom SOAP-Sender zugefügt wurden oder die für einen SOAP-Intermediary bestimmt sind.

Die Kindelemente des SOAP-Headers und SOAP-Bodys werden als Header- bzw. Bodyeinträge bezeichnet. Für sie ist festgelegt, dass ihre Namen in einem Namensraum liegen müssen. Der Inhalt des SOAP-Bodys ist ansonsten außer bei der Übertragung von Fehlern nicht weiter festgelegt. Lediglich optionale Konventionen für Operationsaufrufe schränken seine Verwendung genauer ein. Für Headereinträge gibt es dagegen unterschiedliche weitere Konventionen.

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope"
              xmlns:nwst="http://example.org/NewsService020917/Types"
              xmlns:nwstr="http://example.org/2002/TIP-Transactions"
              xmlns:nwfrw="http://example.org/2002/Security">
  <env:Header>
    <nwstr:TipURL env:mustUnderstand="true">
      tip://x21.example.org/?OleTx-0369c725-b1f0-4ee3-8df2-47c8a983d285
    </nwstr:TipURL>
    <nwfrw:AuthCookie
          env:role="http://example.org/2002/Security/Firewall">
      BC48-35FF-20EB-4ec1-8B7C-AD9F
    </nwfrw:AuthCookie>
  </env:Header>
  <env:Body>
    <nwst:Publish>
      <nwst:SessionID>8CCF4AB6-FF9E-451d-8675-1B4F56B05296</nwst:SessionID>
      <nwst:MessageID>80AF91EA-32EB-4691-9ED5-D0BF047B9424</nwst:MessageID>
    </nwst:Publish>
  </env:Body>
</env:Envelope>
```

Abbildung 9: Serialisierung einer SOAP-Nachricht mit zwei Headereinträgen

3.3 SOAP-Processing-Model

Der SOAP-Standard legt als SOAP-Processing-Model fest, wie SOAP-Knoten SOAP-Nachrichten verarbeiten, wobei Konventionen für Headereinträge eine wichtige Rolle spielen. Beschrieben wird, was in einem SOAP-Knoten konzeptionell zu geschehen hat, wenn eine SOAP-Nachricht empfangen wird. Das soll auch im Folgenden vorgestellt werden. Ein SOAP-Knoten darf SOAP-Nachrichten nur dann anders verarbeiten, wenn die Verarbeitung semantisch äquivalent ist.

[8] In dieser Arbeit werden XML-Dokumente stets als Serialisierung XML 1.0 nach [Bray 00] dargestellt. Das bedeutet jedoch nicht, dass die XML-Dokumente nur serialisiert werden dürfen.

Konzeptionell erfolgt die Verarbeitung einer SOAP-Nachricht in fünf Schritten:

- Prüfen, ob Nachricht eine gültige SOAP-Nachricht ist.
- Untersuchen, welche Rollen der SOAP-Knoten in Bezug auf die SOAP-Nachricht hat.
- Untersuchen, welche Headereinträge für den SOAP-Knoten bestimmt sind.
- Prüfen, ob zwingende Headereinträge bekannt sind und verarbeitet werden können.
- Verarbeiten der SOAP-Nachricht.

Im ersten Schritt untersucht der SOAP-Knoten, ob es sich bei der empfangenen Nachricht überhaupt um eine SOAP-Nachricht handelt. Dazu werden die Regeln aus dem vorherigen Kapitel 3.2 geprüft. Ist die Nachricht in XML 1.0 serialisiert, müssen auch dessen syntaktischen Regeln geprüft werden.

Handelt es sich um eine SOAP-Nachricht, muss der SOAP-Knoten prüfen, welche Rollen er in Bezug auf die SOAP-Nachricht hat. Die Rollen legen fest, welche Headereinträge in der SOAP-Nachricht für ihm bestimmt sind und ob er den SOAP-Body verarbeiten muss. Welche Rollen ein SOAP-Knoten hat, kann a priori feststehen. Die Rollen können aber auch dynamisch, z. B. abhängig von der empfangenen SOAP-Nachricht, bestimmt werden. Festgelegt ist lediglich, dass sich die Rollen eines SOAP-Knotens während der Verarbeitung einer SOAP-Nachricht nicht ändern.

Hat der SOAP-Knoten seine Rollen ermittelt, muss er untersuchen, welche Headereinträge für ihn bestimmt sind. Jede Rolle wird durch eine URI identifiziert. Headereinträge können ein Attribut `env:role` (Role-Attribut) haben, das die URI der Rolle enthält, für die der Headereintrag bestimmt ist. Fehlt das Role-Attribut, ist der Headereintrag für den endgültigen Empfänger bestimmt.

Die SOAP-Nachricht in Abbildung 9 enthält einen Headereintrag mit Role-Attribut. Es ist der Headereintrag `nwfrw:AuthCookie`. Das Role-Attribut hat den Wert `http://example.org/2002/Security/Firewall`. Der Headereintrag ist also für SOAP-Knoten mit der Rolle bestimmt, die durch diese URI identifiziert wird.

Rollen können in Standards oder anwendungsspezifisch definiert werden. Es muss lediglich eine URI festgelegt werden, die sie bezeichnet. Drei Rollen sind bereits im SOAP-Standard definiert und sollen im Folgenden vorgestellt werden.

Die Rolle `http://www.w3.org/2002/06/soap-envelope/role/next` bezeichnet den jeweils nächsten SOAP-Empfänger im SOAP-Message-Path. Jeder SOAP-Empfänger muss diese Rolle haben, da er in Bezug auf eine SOAP-Nachricht immer der nächste Empfänger ist. Das gilt sowohl für SOAP-Intermediaries als auch für den endgültigen SOAP-Empfänger.

Mit `http://www.w3.org/2002/06/soap-envelope/role/ultimateReceiver` wird der endgültige SOAP-Empfänger bezeichnet. Diese URI ist der Vorgabewert für das Role-Attribut. Fehlt das Role-Attribut, ist ein Headereintrag also für den endgültigen Empfänger bestimmt.

Die Rolle `http://www.w3.org/2002/06/soap-envelope/role/none` ist die letzte drei im SOAP-Standard definierten. Kein SOAP-Knoten darf diese Rolle haben. Ist sie im Role-Attribut eines Headereintrages enthalten, ist dieser an keinen SOAP-Knoten adressiert. Er kann jedoch Daten enthalten, auf die von anderen Headereinträgen oder aus dem SOAP-Body verwiesen wird.

Nachdem der SOAP-Knoten untersucht hat, welche Headereinträge für ihn bestimmt sind, muss er prüfen, ob er diese verarbeiten kann. Die Bedeutung eines Headereintra-

ges wird daran erkannt, welchen Namen in welchem Namensraum er hat. Die SOAP-Nachricht in Abbildung 9 enthält z. B. den Headereintrag `AuthCookie` im Namensraum `http://example.org/2002/Security`. Dieser Kombination ist eine Bedeutung zugeordnet, die ein SOAP-Knoten kennen muss, um den Headereintrag gemäß seiner Spezifikation verarbeiten zu können. Ist die Verarbeitung nach der Spezifikation nicht möglich oder kennt ein SOAP-Knoten den Headereintrag nicht, darf er ihn grundsätzlich ignorieren.

Ein SOAP-Sender kann jedoch für einen Headereintrag festlegen, dass dieser vom SOAP-Empfänger nicht ignoriert werden darf. Hierzu muss er dem Headereintrag das Attribut `env:mustUnderstand` mit dem Wert `true` zufügen. Dadurch wird dieser zu einem zwingenden Headereintrag. Die SOAP-Nachricht in Abbildung 9 enthält ein Beispiel. Dort ist der Headereintrag `nwstr:TipURL` ein zwingender Headereintrag, den der SOAP-Empfänger nicht ignorieren darf.

Kann ein SOAP-Empfänger einen zwingenden Headereintrag, der für ihn bestimmt ist, nicht nach dessen Spezifikation verarbeiten, darf er die Verarbeitung der SOAP-Nachricht überhaupt nicht beginnen. Statt dessen muss er für den SOAP-Sender eine SOAP-Nachricht erzeugen, in der diese Fehlersituation angezeigt wird, was im nachfolgenden Kapitel 3.4 erläutert wird.

Kann der SOAP-Empfänger dagegen alle für ihn bestimmten Headereinträge verarbeiten, beginnt er mit der Verarbeitung von Headereinträgen. Außerdem verarbeitet er auch den SOAP-Body, wenn er der endgültige SOAP-Empfänger der Nachricht ist. Die Reihenfolge, in der Headereinträge und SOAP-Body verarbeitet werden, ist im SOAP-Standard nicht festgelegt, sie ist implementierungsspezifisch. Eine Anwendung könnte aber einen zwingenden Headereintrag verwenden, der eine Verarbeitung in einer bestimmten Reihenfolge vorschreibt. Diese müsste dann ein SOAP-Knoten beachten oder er dürfte die gesamte SOAP-Nachricht nicht verarbeiten.

Headereinträge und der SOAP-Body müssen gemäß ihren Spezifikationen verarbeitet werden. Dabei können neue SOAP-Nachrichten erzeugt werden. Ist ein SOAP-Knoten ein SOAP-Intermediary, leitet er die SOAP-Nachricht weiter. Vorher kann er Headereinträge entfernen, ändern oder neue erzeugen.

3.4 SOAP-Fehlernachrichten

Tritt bei der Verarbeitung einer SOAP-Nachricht ein Fehler auf, wird eine SOAP-Fehlernachricht (verkürzt „Fehlernachricht“) erzeugt. Sie benachrichtigt den SOAP-Knoten, der die fehlerverursachende SOAP-Nachricht gesendet hat, über den Fehler. Sie enthält Informationen, die den Fehler beschreibt.

Allerdings wird zwischen dem Erzeugen einer Fehlernachricht und ihrem Zurücksenden unterschieden. Abhängig von der verwendeten Protokollbindung kann es unmöglich sein, die Fehlernachricht zuzustellen. Findet die Kommunikation z. B. mit Request-/Response-Semantik statt und wird durch den Response ein Fehler ausgelöst, müsste die Fehlernachricht als dritte Nachricht versendet werden. Das ist in der Request-/Response-Semantik jedoch nicht vorgesehen und geschieht daher in der Regel nicht.

Ein SOAP-Empfänger erkennt eine Fehlernachricht daran, dass der SOAP-Body genau ein Kindelement `env:Fault` (Fault-Element) enthält. Die Kindelemente des Fault-Elementes sind im SOAP-Standard festgelegt, so dass der Empfänger die Fehlernachricht in jedem Fall interpretieren kann.

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope"
              xmlns:nwst="http://example.org/NewsService020917/Types"
              xmlns:nwse="http://example.org/NewsService020917/Errors">
  <env:Body>
    <env:Fault>
      <env:Code>
        <env:Value>env:Sender</env:Value>
        <env:Subcode>
          <env:Value>nwse:InvalidMessageID</env:Value>
        </env:Subcode>
      </env:Code>
      <env:Reason>Provided MessageID(s) invalid.</env:Reason>
      <env:Detail>
        <nwst:MessageIDsSequence>
          <nwst:MessageID>80AF91EA-32EB-4691-9ED5-D0BF047B9424</nwst:MessageID>
          <nwst:MessageID>AD6F371A-ABE9-41f5-8362-AAE7640F1DDD</nwst:MessageID>
        </nwst:MessageIDsSequence>
      </env:Detail>
    </env:Fault>
  </env:Body>
</env:Envelope>
```

Abbildung 10: Beispiel für eine SOAP-Fehlernachricht

Abbildung 10 zeigt ein Beispiel für eine Fehlernachricht. Alle Kindelemente des Fault-Elementes befinden sich im Namensraum des Envelope-Elementes. Die Elemente `env:Code` und `env:Reason` müssen stets vorhanden sein, während die Elemente `env:Node`, `env:Role` und `env:Detail` optional sind. Die Elemente `env:Node` und `env:Role` wurden im Beispiel in Abbildung 10 nicht verwendet.

Die Elemente `env:Code` und `env:Reason` dienen zum anzeigen, welcher Fehler aufgetreten ist. Das Element `env:Reason` enthält eine Beschreibung in textueller Form, die für menschliche Leser bestimmt ist. Dagegen ist die Beschreibung im Element `env:Code` so kodiert, dass sie maschinell verarbeitet werden kann.

Hierzu hat das Element `env:Code` stets ein Kindelement `env:Value`. Dieses enthält einen qualifizierten Namen, der grob die Fehlersituation angibt und als Fehlercode bezeichnet wird. Die als Fehlercode erlaubten qualifizierten Namen sind im SOAP-Standard festgelegt und werden in Tabelle 1 wiedergegeben.

Fehlercode	Bedeutung
`env:Sender`	Der Aufbau oder Inhalt der SOAP-Nachricht war ungültig.
`env:Receiver`	Bei der Verarbeitung der SOAP-Nachricht trat ein Fehler auf, der vermutlich nicht auf Fehler in der SOAP-Nachricht zurückzuführen ist.
`env:VersionMismatch`	Das Envelope-Element wurden nicht (im erwarteten Namensraum) gefunden. (Siehe Kapitel 3.2.)
`env:MustUnderstand`	Ein zwingender Headereintrag war unbekannt oder konnte nicht nach seiner Spezifikation verarbeitet werden. (Siehe Kapitel 3.3.)
`env:DataEncodingUnknown`	In der SOAP-Nachricht wurde eine Kodierung gefunden, die vom SOAP-Empfänger nicht unterstützt wird. (Siehe Kapitel 3.5.)

Tabelle 1: Im SOAP-Standard festgelegte Fehlercodes

Um die Fehlersituation genauer zu beschreiben, kann dem Element `env:Code` ein zweites Kindelement `env:Subcode` zugefügt werden. Dieses hat wieder ein Kindelement `env:Value`, das den sogenannten Subfehlercode enthält. Als Subfehlercodes sind nicht nur die Fehlercodes aus Tabelle 1 erlaubt, sondern beliebige anwendungsspezifisch definierte Fehlercodes, die als qualifizierte Namen dargestellt werden. Auch das Element `env:Subcode` kann wieder ein Kindelement `env:Subcode` mit gleicher Struktur enthalten. Es kann verwendet werden, um die Fehlersituation noch genauer zu beschreiben.

Das Fault-Element kann optional Kindelemente `env:Node` und `env:Role` haben. Beide dienen dazu, den SOAP-Knoten zu identifizieren, bei dem der Fehler aufgetreten ist. Hierzu enthalten beide als Wert eine URI. Beim Element `env:Node` handelt es sich um die URI, die den SOAP-Knoten direkt identifiziert. Beim Element `env:Role` ist es dagegen die Rolle des SOAP-Knotens, in der die SOAP-Nachricht verarbeitet wurde.

Für anwendungsspezifische Erweiterungen einer Fehlernachricht kann das Fault-Element ein Kindelement `env:Detail` enthalten. Kindelemente des Elementes `env:Detail` sollen nachfolgend als Detaileinträge bezeichnet werden. Ihre Namen und ihre Struktur werden anwendungsspezifisch festgelegt. Der SOAP-Standard legt lediglich fest, dass sich ihre Namen in einem Namensraum befinden müssen, wie in der Fehlernachricht in Abbildung 10 zu erkennen.

Über das Vorhandensein des Elementes `env:Detail` wird angezeigt, ob der Fehler bei der Verarbeitung des SOAP-Bodys oder bei der Verarbeitung eines Headereintrages aufgetreten ist. Im ersten Fall muss das Element `env:Detail` vorhanden sein, im zweiten darf es nicht. Damit entfällt bei Fehlern, die bei der Verarbeitung von Headereinträgen auftreten, die Möglichkeit, das Fault-Element anwendungsspezifisch zu erweitern. Solche Erweiterungen müssen Fehlernachrichten als Headereinträge zugefügt werden.

Ebenso wird verfahren, wenn ein zwingender Headereintrag, der für einen SOAP-Knoten bestimmt ist, nicht verstanden wird oder nicht nach seiner Spezifikation verarbeitet werden kann. Dann wird eine Fehlernachricht mit Fehlercode `env:MustUnderstand` erzeugt, wie in Kapitel 3.3 beschrieben. Da der Fehler durch einen Headereintrag ausgelöst wurde, darf das Element `env:Detail` nicht im Fault-Element enthalten sein.

Trotzdem wird in der Fehlernachricht angezeigt, welcher Headereintrag den Fehler ausgelöst hat. Dazu wird ihr ein Headereintrag `flt:Misunderstood`[9] zugefügt, der in einem Attribut den qualifizierten Namen des Headereintrages enthält, der den Fehler ausgelöst hat.

Die Festlegungen über Fehlernachrichten im SOAP-Standard erlauben, dass jeder SOAP-Empfänger diese interpretieren kann. Das gemeinsame Verständnis über zumindest grobe Fehlercodes sowie die Identifikation des fehlerauslösenden Knotens erlauben es einem SOAP-Knoten, beim Empfang einer Fehlernachricht geeignet zu reagieren. Trotzdem wird mit Hilfe der Elemente `env:Subcode` und `env:Detail` erreicht, dass Fehler auch anwendungsspezifisch beschrieben werden können.

3.5 Kodierung anwendungsspezifischer Teile von SOAP-Nachrichten

Wie bei den Fehlernachrichten ist es auch bei SOAP-Nachrichten selbst die festgelegte Struktur, die es jedem SOAP-Empfänger ermöglicht, die SOAP-Nachricht als solche zu erkennen und einheitlich zu behandeln. Ist eine SOAP-Nachricht keine Fehlernachricht,

---

[9] Das Präfix `flt` stehe für den Namensraum `http://www.w3.org/2002/06/soap-faults`.

sind es besonders die anwendungsspezifischen Teile, die die Bedeutung der SOAP-Nachricht ausmachen. Diese können nicht standardisiert werden und sind damit nur für SOAP-Knoten interpretierbar, die für die spezielle Anwendung implementiert wurden.

Beim Entwurf der Schnittstellen von Webservices müssen die anwendungsspezifischen Teile der SOAP-Nachricht spezifiziert werden. Häufig wird dazu von Datenstrukturen ausgegangen, die in Implementierungssprachen für Webkomponenten verwendet werden. Diese werden auf Fragmente von XML-Infosets (XML-Fragmente) abgebildet, die typischerweise aus einem Element und dessen Nachfahren bestehen.

Die Abbildung kann mit Hilfe festgelegter Transformationsregeln geschehen, was insbesondere der Fall ist, wenn die Abbildung maschinell erfolgt. Der SOAP-Standard erlaubt es optional, dass der SOAP-Sender den SOAP-Empfänger über die Menge der verwendeten Transformationsregeln informiert, die als Kodierung bezeichnet wird. Hierzu wird die Kodierung mit einer URI identifiziert und diese dem XML-Fragment zugefügt. Der SOAP-Empfänger kann so die Umkehrtransformation auf das XML-Fragment anwenden und damit aus dem XML-Fragment die ursprüngliche Datenstruktur der Implementierungssprache wiederherstellen.

Die anwendungsspezifischen Teile einer SOAP-Nachricht befinden sich in Body-, Header- und bei Fehlernachrichten in Detaileinträgen. Diese Elemente oder Elemente, die Nachfahren dieser Elemente sind, können das Ergebnis der Transformation sein. URIs der Kodierungen können ihnen als Attribut `env:encodingStyle` (Encodingstyle-Attribut) zugefügt werden. Das Attribut enthält die URIs separiert durch Leerzeichen. Die bezeichneten Transformationsregeln beziehen sich dann auf das Element und seine Nachfahren. Abweichend davon können die Nachfahren ihrerseits wieder Encodingstyle-Attribute haben, um die für sie und ihre Nachfahren verwendeten Kodierungen zu bezeichnen.

Die URI im Encodingstyle-Attribut ist für einen SOAP-Empfänger nur dann hilfreich, wenn ihm die bezeichnete Kodierung bekannt ist. Gäbe es sehr viele Kodierungen, wäre das nicht für alle möglich. Damit wäre der Nutzen des Encodingstyle-Attributes in Frage gestellt. Besser ist es, eine Kodierung zu standardisieren, was durch den SOAP-Standard geschehen ist. Sie wird als SOAP-Encoding bezeichnet und mit der URI `http://www.w3.org/2002/06/soap-encoding` identifiziert. Es dürfen weiterhin aber auch andere Kodierungen verwendet werden.

Ein Problem beim Erstellen von standardisierten Transformationsregeln ist, dass diese mit beinhalten, wie Datenstrukturen in den Implementierungssprachen für Webkomponenten modelliert werden. Gängig sind Arrays, Strukturen (Records) oder auch Objektgraphen. Leider unterscheiden sich Implementierungssprachen hier in vielen Details. So können z. B. Arrays eine feste oder dynamische Länge haben. Objektorientierte Systeme können Mehrfachvererbung unterstützen oder nicht.

Um trotz dieser Unterschiede die Transformationsregeln des SOAP-Encodings eindeutig definieren zu können, enthält der SOAP-Standard das SOAP-Datenmodell. Mit ihm werden Daten nach Paradigmen modelliert, die auch vielen Implementierungssprachen üblich sind. Es sind Arrays und Strukturen. Auch werden zyklische Referenzen unterstützt. Im SOAP-Datenmodell modellierte Daten können dann mit den Transformationsregeln des SOAP-Encodings auf XML-Fragmente abgebildet werden und umgekehrt.

Die Verwendung des SOAP-Datenmodells erlaubt, eindeutige Transformationsregeln zu definieren. Es sei jedoch darauf hingewiesen, dass zusätzlich eine Abbildung der Datenstrukturen der Implementierungssprachen auf das SOAP-Datenmodell erforderlich ist. Diese ist nicht standardisiert.

3.6 SOAP-Datenmodell

Das SOAP-Datenmodell modelliert anwendungsspezifische Daten als gerichteten Graphen mit beschrifteten Kanten, der auch Zyklen enthalten darf. Es soll im Folgenden kurz vorgestellt werden. Graphische Beispiele werden im nachfolgenden Kapitel 3.7 gegeben.

Eine Kante eines Knotens, die zu einem anderen Knoten führt, soll als ausgehende Kante bezeichnet werden. Umgekehrt führt eine eingehende Kante von einem anderen Knoten zum Knoten hin.

Ein einfacher Wert wird im SOAP-Datenmodell als Blattknoten repräsentiert. Ein Blattknoten ist ein Knoten, der keine ausgehenden Kanten hat. Zusätzlich zum repräsentierten Wert kann ein Knoten einen Typ und einen eindeutigen Bezeichner haben. Als Typsystem wird hierzu XML-Schema verwendet. Die Typen werden daher als qualifizierte Namen dargestellt.

Strukturen und Arrays werden als Knoten dargestellt, deren ausgehende Kanten sie mit ihren Elementen verbinden. Bei Strukturen haben alle ausgehenden Kanten unterschiedliche Beschriftungen, die zur eindeutigen Identifikation der Elemente der Struktur (Strukturelemente) verwendet werden. Als Beschriftungen von Kanten werden qualifizierte Namen verwendet. Wie bei qualifizierten Namen üblich, werden sie als gleich angesehen, wenn ihr lokaler Name und der durch das Präfix bezeichnete Namensraum identisch sind. Anders als bei Strukturen spielen bei Arrays die Beschriftungen der Kanten keine Rolle. Die Elemente eines Arrays (Arrayelemente) werden durch die Position der ausgehenden Kanten unterschieden. Neben Strukturen und Arrays stellt der SOAP-Standard noch Generics vor, deren Elemente sowohl über die Beschriftung als auch durch ihre Position identifiziert werden können, in dieser Arbeit aber nicht weiter betrachtet werden sollen.

Knoten können mehrere eingehende Kanten haben (mehrfach referenzierte Knoten). Damit kann ein Knoten gleichzeitig Element mehrerer Strukturen oder Arrays sein. In Implementierungssprachen werden solche Datenstrukturen häufig mit Hilfe von Referenzen realisiert. Mit ihnen ist es auch möglich, dass Datenstrukturen Zyklen enthalten. Diese führen im SOAP-Datenmodell zu Zyklen im Graphen.

3.7 SOAP-Encoding

Ein Graph des SOAP-Datenmodells kann mit Hilfe der Transformationsregeln des SOAP-Encodings auf ein XML-Fragment abgebildet werden. Das XML-Fragment besteht dann aus einem Element und seinen Nachfahren. Allerdings ist diese Abbildung nicht eindeutig. Ein Graph kann in unterschiedliche XML-Fragmente transformiert werden. Alle können in den gerichteten Graphen des SOAP-Datenmodells zurücktransformiert werden.

Die dazu verwendeten Regeln sollen im Folgenden anhand von Beispielen erläutert werden. Auf die Vollständigkeit der Darstellung wurde dabei kein Wert gelegt. Eine vollständige Darstellung wird im 3. Kapitel des zweiten Teils des SOAP-Standards [Gudgin 02b] gegeben.

Abbildung 11 zeigt einen Ausschnitt eines Graphen des SOAP-Datenmodells, der einen einfachen Wert repräsentiert. Für den Wert gibt es im Graphen einen Knoten, der ihn enthält. Der Wert wird im XML-Fragment in eine Folge von Zeichen transformiert, wie in Abbildung 12 dargestellt. Der Knoten selbst wird mit seiner eingehenden Kante zu

einem Element, dessen Name dem der Kante entspricht. Dabei sei in Abbildung 11 und den folgenden Beispielen angenommen, dass das Präfix `my` an den Namensraum `http://example.org/2002/EncodingSample` gebunden ist.

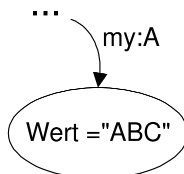

Abbildung 11: Einfacher Wert in SOAP-Datenmodell

```
<my:A xmlns:my="http://example.org/2002/EncodingSample">ABC</my:A>
```

Abbildung 12: Einfacher Wert , kodiert als XML-Fragment

Abbildung 13 zeigt eine Struktur, die ihrerseits drei Strukturelemente mit jeweils einfachen Werten enthält. Die drei Strukturelemente werden durch die qualifizierten Namen (`my:A`, `my:B`, `my:C`) der Kanten unterschieden, die zu ihnen führen. Die Struktur aus Abbildung 13 wird zusammen mit ihrer eingehenden, benannten Kante in das XML-Fragment in Abbildung 14 abgebildet. Der Knoten, der die Struktur repräsentiert, sowie eine eingehende Kante werden in ein Element abgebildet. Die drei Strukturelemente werden dem Element als Kindelemente zugefügt.

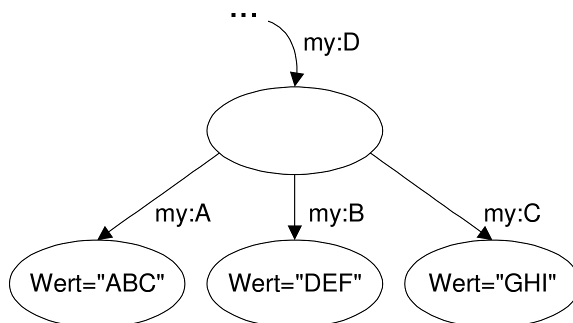

Abbildung 13: Struktur in SOAP-Datenmodell

```
<my:D xmlns:my="http://example.org/2002/EncodingSample">
  <my:A>ABC</my:A>
  <my:B>DEF</my:B>
  <my:C>GHI</my:C>
</my:D>
```

Abbildung 14: Struktur, kodiert als XML-Fragment

Arrays werden wie Strukturen in XML-Fragmente transformiert. Der Hauptunterschied ist, dass die Beschriftung der ausgehenden Kanten der Arrayelemente keine Rolle spielt, weil sie durch die Position unterschieden werden. Innerhalb eines Arrays dürfen Kanten daher auch gleich beschriftet werden. Wird das Array in ein XML-Fragment transformiert, führt das zu Kindelementen mit gleichem Namen. Die Position der ausgehenden Kanten wird dabei auf die Position der Kindelemente abgebildet.

Über diesen Hauptunterschied hinaus gibt es im XML-Fragment die Möglichkeit, die Größe von Arrays anzugeben. Hierzu dient das Attribut `enc:arraySize`[10]. Sein Wert ist eine Liste der Dimensionen des Arrays[11].

[10] Das Präfix `enc` stehe für den Namensraum `http://www.w3.org/2002/06/soap-encoding`.

Ein Knoten kann im SOAP-Datenmodell einen Typ haben. Dieser wird im XML-Fragment dem Element, das den Knoten repräsentiert, als Attribut `xsi:type`[12] zugefügt, das im Standard von XML-Schema definiert ist. Das XML-Fragment in Abbildung 15 zeigt das am Beispiel des einfachen Wertes, der bereits in Abbildung 12 als XML-Fragment dargestellt wurde. Hier ist jedoch der Typ des Wertes (`xsd:string`) explizit angegeben. Neben dieser Notation für Typen gibt es für Arrays eine weitere, die ausnutzt, dass alle Arrayelemente eines Arrays in der Regel den gleichen Typ haben.

```
<my:A xmlns:my="http://example.org/2002/EncodingSample"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xsi:type="xsd:string"
>ABC</my:A>
```

Abbildung 15: Einfacher, typisierter Wert, kodiert als XML-Fragment

Abbildung 16 zeigt einen komplexeren Graphen des SOAP-Datenmodells. Er enthält drei mehrfach referenzierte Knoten und einen Zyklus. Wird dieser Graph in ein XML-Fragment transformiert, soll die Identität von Knoten erhalten bleiben. Bei der Rücktransformation in einen Graphen soll wieder jeder Knoten nur einmal enthalten sein.

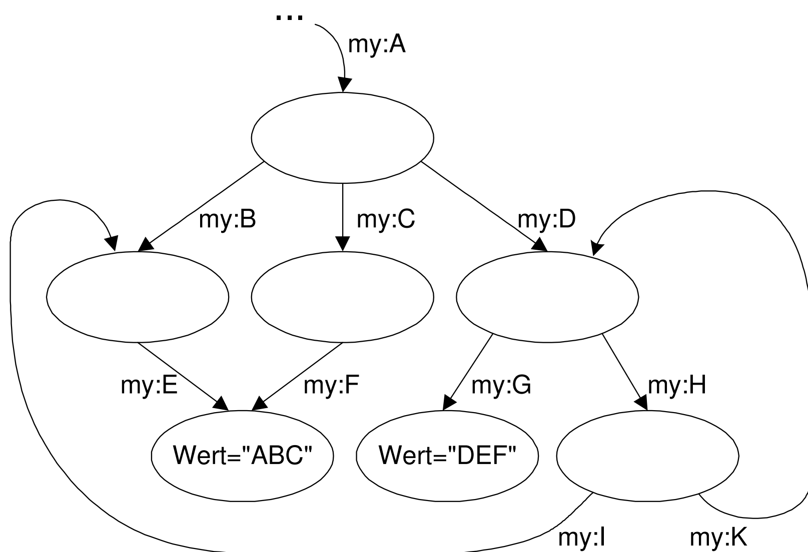

Abbildung 16: SOAP-Datenmodell mit mehrfach referenziertem Knoten und Zyklus

Das SOAP-Encoding stellt dazu jeden Knoten genau einmal im XML-Fragment als Element dar. Hat der Knoten nur eine eingehende Kante, wird er, wie oben beschrieben, zusammen mit der eingehenden Kante in ein Element transformiert. Hat er dagegen mehrere eingehende Kanten, wird er mit einer dieser Kanten in ein Element transformiert. Dem Element wird ein Attribut `id` zugefügt, das dazu dient, das Element im XML-Dokument eindeutig zu identifizieren. Hierzu erhält das Attribut einen eindeutigen Wert.

---

[11] Mehrere Dimensionen sind für mehrdimensionale Arrays erforderlich. Allerdings sagt der SOAP-Standard nichts darüber aus, wie mehrdimensionale Arrays im SOAP-Datenmodell oder in XML-Fragmenten zu kodieren sind.

[12] Das Präfix `xsi` stehe für den Namensraum `http://www.w3.org/2001/XMLSchema-instance`, der zu XML-Schema gehört.

Alle weiteren eingehenden Kanten des Knotens werden in eigene Elemente transformiert, die auf das Element verweisen, das den Knoten repräsentiert. Dazu erhalten diese Elemente jeweils ein Attribut `ref`, das den Wert des Attributes `id` des Elementes enthält, das den Knoten repräsentiert. Abbildung 17 zeigt ein mögliches Ergebnis bei der Transformation des Graphen aus Abbildung 16.

```
<my:A xmlns:my="http://example.org/2002/EncodingSample">
  <my:B id="x1">
    <my:E id="x2">ABC</my:E>
  </my:B>
  <my:C>
    <my:F ref="x2" />
  </my:C>
  <my:D id="x3">
    <my:G>DEF</my:G>
    <my:H>
      <my:I ref="x1" />
      <my:K ref="x3" />
    </my:H>
  </my:D>
</my:A>
```

Abbildung 17: Komplexer Graph, kodiert als XML-Fragment

Abbildung 17 und die davor gezeigten Beispiele zeigen XML-Fragmente, die als Ergebnis durch die Transformation nach dem SOAP-Encoding entstanden sind. Die XML-Fragmente bilden die anwendungsspezifischen Teile der SOAP-Nachricht und werden somit als Header-, Body- oder Detaileinträge verwendet. Abbildung 18 zeigt ein Beispiel für eine SOAP-Nachricht, bei der das XML-Fragment aus Abbildung 14 als Bodyeintrag verwendet wurde. Dem Bodyeintrag wurde das Encodingstyle-Attribut mit der URI des SOAP-Encodings zugefügt, um dem SOAP-Empfänger anzuzeigen, dass dieser Teil der Nachricht nach den Transformationsregeln des SOAP-Encodings entstanden ist.

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope"
  <env:Body>
    <my:D xmlns:my="http://example.org/2002/EncodingSample"
      env:encodingStyle="http://www.w3.org/2002/06/soap-encoding">
      <my:A>ABC</my:A>
      <my:B>DEF</my:B>
      <my:C>GHI</my:C>
    </my:D>
  </env:Body>
</env:Envelope>
```

Abbildung 18: SOAP-Nachricht mit durch SOAP-Encoding gebildetem Bodyeintrag

3.8 Kommunikationsmuster

Der SOAP-Standard betrachtet in weiten Teilen nur den Austausch einzelner SOAP-Nachrichten zwischen SOAP-Sendern und SOAP-Empfängern. Webkomponenten tauschen in der Regel miteinander eine Reihe von SOAP-Nachrichten aus. Die Kommunikation wird häufig nach bestimmten Kommunikationsmustern ablaufen, die einige wenige SOAP-Nachrichten zu logischen Gruppen zusammenfassen, die als Kommunikationen bezeichnet werden sollen.

Welche Kommunikationsmuster verwendet werden sollen, schreibt der SOAP-Standard nicht vor. Er beschreibt jedoch, wie Kommunikationsmuster außerhalb des SOAP-Standards definiert werden sollen.

Auf diese Art definiert er auch zwei Kommunikationsmuster selbst: das Request-/Response-Kommunikationsmuster (request-response message exchange pattern) und das SOAP-Response-Kommunikationsmuster (SOAP response message exchange pattern). Beide haben eine Request-/Response-Semantik, fassen also zwei Nachrichten (Request und Response) zusammen, die zwischen zwei Knoten in unterschiedlicher Richtung ausgetauscht werden. Der Request wird zuerst gesendet, als Antwort danach in umgekehrter Richtung der Response. Der Austausch von zwei solchen Nachrichten soll als eine Request-/Response-Kommunikation bezeichnet werden.

Das Request-/Response-Kommunikationsmuster verwendet SOAP-Nachrichten sowohl für den Request als auch für den Response, wie in Abbildung 19 veranschaulicht. Requests und Responses, die SOAP-Nachrichten sind, sollen als SOAP-Requests bzw. SOAP-Responses bezeichnet werden. Das Kommunikationsmuster legt fest, wie die beiden SOAP-Nachrichten ausgetauscht werden. Es legt jedoch nicht fest, wie mehrere Request-/Response-Kommunikationen stattfinden müssen. Z. B. wäre es möglich, dass mehrere gleichzeitig oder nur nacheinander stattfinden dürfen.

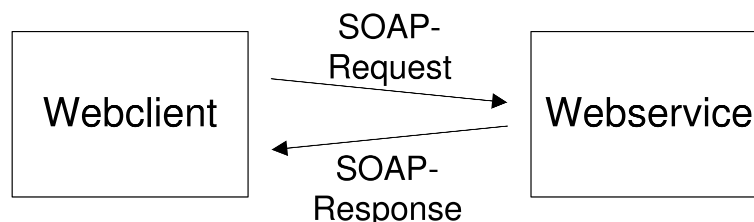

Abbildung 19: Request-/Response-Kommunikationsmuster

Das Request-/Response-Kommunikationsmuster erlaubt, dass eine Request-/Response-Kommunikation im Falle von Fehlern abgebrochen wird. Dann kann der SOAP-Response fehlen. Tritt bei der Verarbeitung eines SOAP-Requests ein Fehler auf, wird jedoch in den meisten Fällen der SOAP-Response eine SOAP-Fehlernachricht sein. Tritt umgekehrt bei der Verarbeitung eines SOAP-Responses ein Fehler auf, wird das nicht durch eine Fehlernachricht beantwortet. Das würde eine dritte Nachricht erfordern, die im Request-/Response-Kommunikationsmuster nicht vorgesehen ist.

Der SOAP-Knoten, der den SOAP-Request sendet, hat die Aufgabe, einen empfangenen SOAP-Response dem SOAP-Request zuzuordnen. Das ist auf unterschiedliche Arten möglich. Es kann vom Protokoll geleistet werden, mit dessen Hilfe die SOAP-Nachrichten ausgetauscht werden. Das geschieht z. B. bei Verwendung von HTTP. Leistet es das Protokoll nicht, kann die Zuordnung auch mit Hilfe von Headereinträgen geschehen. Diese müssen die Request-/Response-Kommunikationen identifizieren, z. B. indem sie einen Bezeichner enthalten, der im SOAP-Request und SOAP-Response den gleichen Wert hat.

Im SOAP-Standard wird festgestellt, dass im World Wide Web Ressourcen mit URIs identifiziert werden. Bezieht sich daher eine SOAP-Nachricht auf eine Ressource, sollte diese mit einer URI identifiziert werden und nicht nur durch eine andere Darstellung innerhalb der SOAP-Nachricht. An diese URI sollte die SOAP-Nachricht adressiert werden[13]. Damit wird die Ressource zumindest konzeptionell zum SOAP-Knoten.

Besonders interessant ist diese Tatsache in Bezug auf Request-/Response-Kommunikationen, die lediglich dazu dienen, die Ressource abzufragen, ohne diese zu

[13] In Kapitel 4.1.1 des 2. Teils des SOAP-Standards [Gudgin 02b] wird diese Aussage auf Remote Procedure Calls bezogen. Wegen ihrer dort gegebenen, grundsätzlichen Begründung sollte sie jedoch in dieser Allgemeinheit gültig sein.

verändern (Seiteneffektfreiheit). Die Ressource wird durch die URI identifiziert, an die der Request gesendet wird. Im Request muss sie dann nicht mehr beschrieben sein. Müssen darüber hinaus auch keine weiteren Daten z. B. im SOAP-Header übertragen werden, muss der Request überhaupt keine Daten enthalten. Alle benötigten Daten befinden sich dann in der URI, die die Ressource identifiziert.

Diese Idee führt zum SOAP-Response-Kommunikationsmuster, das neben dem Request-/Response-Kommunikationsmuster im SOAP-Standard definiert ist. Es kann nur in der oben beschriebenen Situation verwendet werden, in der eine Ressource lediglich abgefragt wird und im Request keine Daten enthalten sind. Für den Request wird daher auf eine SOAP-Nachricht verzichtet. Im darunter liegenden Protokoll wird jedoch trotzdem ein bindungsspezifischer Request geschickt, der die URI enthält, die die Ressource identifiziert. Für den Response wird auch in diesem Kommunikationsmuster eine SOAP-Nachricht verwendet, wie in Abbildung 20 veranschaulicht.

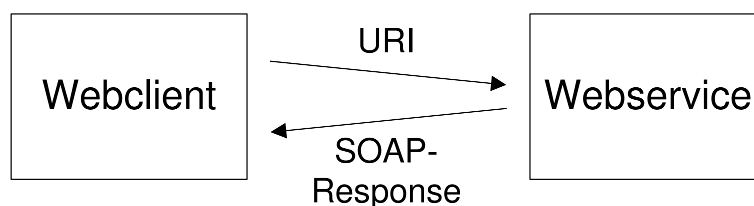

Abbildung 20: SOAP-Response-Kommunikationsmuster

Abgesehen davon, dass für den Request keine SOAP-Nachricht verwendet wird, unterscheidet sich das SOAP-Response-Kommunikationsmuster vom Request-/Response-Kommunikationsmuster wenig. Auch hier kann die Request-/Response-Kommunikation im Falle von Fehlern abgebrochen oder der SOAP-Response eine Fehlernachricht sein. Ebenso wird ein durch den SOAP-Response ausgelöster Fehler nicht mit einer Fehlernachricht beantwortet. Nebenläufige Request-/Response-Kommunikationen sind möglich und der SOAP-Response muss dem Request zugeordnet werden können.

Das SOAP-Response-Kommunikationsmuster bietet den Vorteil, dass festgelegt ist, dass Ressourcen nicht verändert werden. Insbesondere unter Verwendung von HTTP wird so der Einsatz von Caches möglich, wie sie im World Wide Web üblich sind. Sie erkennen die URI im Request und können den SOAP-Response direkt zurücksenden, wenn dieser vorher gespeichert wurde. Wird eine Operation dagegen mit Hilfe des Request-/Response-Kommunikationsmusters realisiert, können Cachemechanismen wegen möglicher Seiteneffekte nicht verwendet werden.

3.9 Konventionen für einheitliche SOAP-Nachrichten

Der SOAP-Standard enthält optionale Konventionen, wie SOAP-Nachrichten einheitlich gebildet werden, wenn Operationen Remote Procedure Calls (RPCs) realisieren, also Aufrufe von Prozeduren über ein Netzwerk. SOAP-Nachrichten lassen sich jedoch auch unabhängig von RPCs nach diesen Konventionen strukturieren. In diesem Sinne sind es allgemein Konventionen für einheitliche SOAP-Nachrichten. Trotzdem sollen sie in Anlehnung an den SOAP-Standard als RPC-Repräsentation bezeichnet werden.

SOAP-Nachrichten nach der RPC-Repräsentation können nach den beiden in Kapitel 3.8 vorgestellten Kommunikationsmustern ausgetauscht werden, ebenso mit anderen außerhalb des SOAP-Standards definierten. Benötigt wird ein Kommunikationsmuster mit Request-/Response-Semantik, da Operationsaufrufe auf jeweils einen Request und einen Response abgebildet werden.

Hierzu werden die In- und Inout-Parameter (siehe Kapitel 2.1.2) einer Operation vom Webclient in einem Request kodiert und dann zum Webservice geschickt. Der Webservice führt mit den Parametern aus dem Request die Operation aus. Danach kodiert er deren Out- und Inout-Parameter in einer Response-Nachricht und schickt sie an den Webclient zurück.

Nach der RPC-Repräsentation werden im SOAP-Body nur die Parameter der Operation übertragen. Die einzige Ausnahme sind SOAP-Fehlernachrichten, bei denen im Fehlerfalle im SOAP-Body ein Fault-Element übertragen wird. Sollen neben Parametern zusätzliche Daten übertragen werden, müssen diese als Headereinträge kodiert werden.

Für Operationen eines Webservices muss entschieden werden, welches der beiden Kommunikationsmuster aus dem SOAP-Standard verwendet wird. Das SOAP-Response-Kommunikationsmuster kann dabei nur in bestimmten Fällen verwendet werden. Zunächst muss die Operation seiteneffektfrei sein. Außerdem dürfen alle In- und Inout-Parameter der Operation lediglich die Ressource identifizieren, die mit der Operation abgefragt werden soll, so dass sie mit in der URI des Webservices kodiert werden. Schließlich darf es nicht erforderlich sein, dass neben den Parametern zusätzliche Daten als Headereinträge mit im Request übertragen werden. Triff nur eine der Bedingungen nicht zu, kann das SOAP-Response-Kommunikationsmuster nicht verwendet werden. Die Verwendung des Request-/Response-Kommunikationsmusters ist dagegen immer möglich.

In den folgenden Absätzen sei daher angenommen, dass das Request-/Response-Kommunikationsmuster verwendet wird und damit der Request und der Response SOAP-Nachrichten sind. Das Gesagte gilt jedoch auch für das SOAP-Response-Kommunikationsmuster, soweit es sich nicht auf den SOAP-Request bezieht.

Wie die Parameter im SOAP-Body kodiert werden, ist durch die RPC-Repräsentation mit Hilfe von Strukturen bzw. Arrays festgelegt, die auch im SOAP-Datenmodell verwendet werden. (Siehe Kapitel 3.6.) Die In- und Inout-Parameter für den SOAP-Request werden zusammen als eine Struktur oder ein Array modelliert. Jeder Parameter ist ein Struktur- bzw. ein Arrayelement und wird durch seinen Namen bzw. eine Position identifiziert. Die Struktur bzw. das Array hat den Namen der Operation[14].

Die so erzeugte Datenstruktur wird mit Hilfe einer Kodierung in ein XML-Fragment transformiert. Grundsätzlich können beliebige Transformationen verwendet werden, mit denen Strukturen und Arrays transformiert werden können. Das leistet vor allem das SOAP-Encoding aus Kapitel 3.7. Das Encodingstyle-Attribut wird verwendet, um in der SOAP-Nachricht anzugeben, welche Kodierung verwendet wurde.

Wurde das SOAP-Encoding verwendet, ist das Ergebnis ein XML-Fragment, das aus einem Element mit seinen Nachfahren besteht. Das Element hat den Namen der Operation. Der Name befindet sich in einem Namensraum. Jedes seiner Kindelemente entspricht einem In- oder Inout-Parameter. Die Kindelemente werden durch ihre Namen oder ihre Position unterschieden.

Das XML-Fragment wird zum einzigen Bodyeintrag des SOAP-Requests. Es sei festgestellt, dass bei Verwendung der RPC-Repräsentation und dem SOAP-Encoding damit jede SOAP-Nachricht genau einen Body-Eintrag hat. Hierzu ist wichtig, dass der Response eine vergleichbare Struktur hat wie der Request und auch in Fehlernachrichten der SOAP-Body genau einen Bodyeintrag hat, nämlich das Fault-Element.

[14] Selbstverständlich sind es eigentlich die eingehenden Kanten, die Namen haben. Um den Text verständlich zu halten, wurde hier vereinfacht.

Die SOAP-Nachricht in Abbildung 18 auf Seite 40 kann als SOAP-Request eines Operationsaufrufs nach der RPC-Repräsentation aufgefasst werden. Der Name der Operation ist dann `D` im Namenraum `http://example.org/2002/EncodingSample`. Die Operation hat drei In- oder Inout-Parameter mit den Namen `A`, `B` und `C` im gleichen Namenraum. Mit dem Encodingstyle-Attribut wird angezeigt, dass das SOAP-Encoding verwendet wurde.

Der SOAP-Response wird im wesentlichen auf die gleiche Art gebildet wie der SOAP-Request. Statt der In- und Inout-Parameter werden im SOAP-Response selbstverständlich die Out- und Inout-Parameter kodiert. Der Name der Struktur, die die Parameter umschließt, wird auch aus dem Operationsnamen gebildet. Es wird die Konkatenation aus dem Operationsnamen und der Zeichenkette „`Response`“ verwendet.

Implementierungssprachen für Webkomponenten erlauben häufig, einen Out-Parameter einer Operation als Rückgabeparameter zu kennzeichnen. Die Operation wird dann als Funktion bezeichnet. Mit der RPC-Repräsentation kann der Rückgabeparameter in der SOAP-Nachricht markiert werden. Hierzu wird er zunächst wie jeder Out- und Inout-Parameter in der SOAP-Nachricht kodiert. Zusätzlich wird dem Bodyeintrag noch ein Element `rpc:result`[15] (Result-Element) zugefügt, das den qualifizierten Namen des Rückgabeparameters enthält. Das Result-Element ist jedoch nur vorhanden, wenn die Operation einen Rückgabeparameter hat.

Abbildung 21 zeigt ein Beispiel für einen SOAP-Response. Für ihn sei angenommen, dass er der SOAP-Response zum SOAP-Request aus Abbildung 18 ist. Der Name der Operation ist `D`, daher hat der Bodyeintrag den Namen `DResponse`. Es gibt drei Out- und Inout-Parameter. Ihre Namen sind `R`, `B` und `E`. `R` und `E` sind Out-Parameter, `B` ist ein Inout-Parameter. Das ist daran zu erkennen, dass `B` auch im SOAP-Request vorhanden ist, `R` und `E` dagegen nicht. `R` ist zusätzlich der Rückgabeparameter der Operation, was mit dem Result-Element angezeigt wird, das den qualifizierten Namen des Rückgabeparameters `my:R` enthält. Mit dem Encodingstyle-Attribut wird wieder angezeigt, dass das SOAP-Encoding verwendet wurde.

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope">
  <env:Body>
    <my:DResponse
          xmlns:my="http://example.org/2002/EncodingSample"
          xmlns:rpc="http://www.w3.org/2002/06/soap-rpc"
          env:encodingStyle="http://www.w3.org/2002/06/soap-encoding">
      <rpc:result>my:R</rpc:result>
      <my:R>42</my:R>
      <my:B>DEF</my:B>
      <my:E>GHI</my:E>
    </my:DResponse>
  </env:Body>
</env:Envelope>
```

Abbildung 21: SOAP-Response, gebildet nach der RPC-Repräsentation

3.10 Protokollbindungen

SOAP-Nachrichten, wie die eben gezeigten, werden über das Internet zwischen Webkomponenten ausgetauscht. Hierzu werden Protokolle benötigt, die die SOAP-Nachrichten transportieren. Der SOAP-Standard schreibt nicht vor, welche Protokolle

[15] Das Präfix `rpc` stehe für den Namensraum `http://www.w3.org/2002/06/soap-rpc`.

verwendet werden müssen. Statt dessen legt er generelle Regeln für die Definition von Protokollbindungen fest, die jeweils die Abbildung von SOAP auf ein Protokoll beschreiben. Außerdem enthält er selbst eine Protokollbindung für HTTP.

Für jede Protokollbindung muss festgelegt werden, wie SOAP-Nachrichten mit den Mitteln des verwendeten Protokolls übertragen werden. Da SOAP-Nachrichten zunächst XML-Infosets sind, müssen diese für die Übertragung serialisiert werden. Der SOAP-Standard schreibt nicht vor, wie die SOAP-Nachrichten serialisiert werden müssen. Eine übliche Serialisierung ist XML 1.0 gemäß [Bray 00]. Eine Protokollbindung kann jedoch auch eine andere verwenden, wenn trotzdem sichergestellt ist, dass der Empfänger der Nachricht den XML-Infoset der SOAP-Nachricht wieder ohne Informationsverlust rekonstruieren kann.

Neben der Serialisierung muss für eine Protokollbindung auch festgelegt werden, welche Kommunikationsmuster und sonstigen Features sie unterstützt. Sie wird in der Regel nur bestimmte Kommunikationsmuster unterstützen. Nahe liegend ist die Unterstützung der im SOAP-Standard definierten Kommunikationsmuster. Zu sonstigen Features gehört z. B. die Frage, ob es das Protokoll unterstützt Requests und Responses einander zuzuordnen.

Die einzige Protokollbindung, die bereits im SOAP-Standard festgelegt wurde, ist die für das Protokoll HTTP, die als HTTP-Bindung bezeichnet wird. Bereit bei der Verwendung von SOAP 1.1 hat sich gezeigt, dass dieses Protokoll am häufigsten mit SOAP eingesetzt wird. Trotzdem ist es auch mit SOAP 1.2 optional.

Mit der HTTP-Bindung sollte die Version 1.1 von HTTP verwendet werden, wie sie in [Fielding 99] spezifiziert wird oder spätere, kompatible Versionen mit der gleichen Haupt-Versionsnummer. Einige optionale Features der HTTP-Bindung basieren auf Eigenschaften dieser Version. Bei Verwendung der früheren Version 1.0 wären sie nicht verfügbar.

HTTP ist ein Request-/Response-Protokoll. Seine Adressierung basiert auf URIs. Will ein HTTP-Client auf eine durch eine URI identifizierte Ressource zugreifen, ermittelt er aus der URI zunächst die IP-Adresse des HTTP-Servers, der die Ressource verwaltet. Zu diesem baut er eine TCP/IP-Verbindung auf oder verwendet eine bereits bestehende. Dann sendet er über sie einen HTTP-Request. Der HTTP-Request hat unterschiedliche Felder. Sie enthalten die URI der zuzugreifenden Ressource, eine Webmethode, die angibt was mit der Ressource geschehen soll, zu übertragende Nutzdaten und weitere z. T. optionale HTTP-Header. Außerdem ist die verwendete Version von HTTP angegeben.

Der HTTP-Server verarbeitet den HTTP-Request und schickt danach auf der gleichen TCP/IP-Verbindung einen HTTP-Response zurück. Auch der HTTP-Response ist aus unterschiedlichen Feldern aufgebaut. Ein Statuscode gibt an, ob ein Fehler aufgetreten ist. Diese Information wird zusätzlich auch in für Menschen lesbarer Form übertragen. Neben der verwendeten Version von HTTP und z. T. optionalen HTTP-Headern werden auch im HTTP-Response wieder Nutzdaten übertragen.

Welche Typen die Nutzdaten in HTTP-Nachrichten haben, wird in ihnen mit Hilfe eines HTTP-Headers angegeben. Zur Bezeichnung werden von der IETF in [Freed 96] standardisiert Medientypen (media types) verwendet.

SOAP-Nachrichten werden in HTTP-Nachrichten als Nutzdaten übertragen. Welcher Medientyp für sie angegeben wird, hängt von der verwendeten Serialisierung ab. Grundsätzlich sind beliebige Serialisierungen erlaubt, die es ermöglichen XML-Infosets von SOAP-Nachrichten zu übertragen. Jede Implementierung der HTTP-Bindung muss

aber zumindest die Serialisierung von XML 1.0 gemäß [Bray 00] unterstützen. Für diese Art von Nutzdaten hat die XML Protocol Working Group der IETF in [Baker 02] den Medientyp mit Namen `application/soap+xml` vorgeschlagen.

Die HTTP-Bindung unterstützt beide Kommunikationsmuster, die im SOAP-Standard definiert sind, also das Request-/Response- und das SOAP-Response-Kommunikationsmuster. Beide lassen sich durch ihre Request-/Response-Semantik gut auf HTTP abbilden. Beim Request-/Response-Kommunikationsmuster wird der SOAP-Request in den Nutzdaten des HTTP-Requests übertragen und der SOAP-Response in den Nutzdaten des HTTP-Response. Analog wird beim SOAP-Response-Kommunikationsmuster der SOAP-Response in den Nutzdaten des HTTP-Response übertragen. Die URI der Ressource, die abgefragt werden soll, ist die URI, die im HTTP-Request übertragen wird.

Diese Abbildung von SOAP-Nachrichten auf HTTP-Nachrichten ermöglicht, SOAP-Request und SOAP-Response einander zuzuordnen. Auf einer TCP/IP-Verbindung wird in HTTP stets nur ein HTTP-Request übertragen und dann der zugehörige HTTP-Response. Erst danach darf ein weiterer HTTP-Request übertragen werden und dann wieder der zugehörige HTTP-Response. Durch Fehlen von Nebenläufigkeit auf den verwendeten TCP/IP-Verbindungen ist somit die Zuordnung von HTTP-Request und HTTP-Response eindeutig möglich und damit die Zuordnung von SOAP-Request und SOAP-Response.

Mit Hilfe von Webmethoden wird in HTTP angegeben, welche Operation auf einer Ressource durchgeführt werden soll. Eine „kleine Menge" von Webmethoden ist in HTTP definiert, z. B. `GET`, `POST`, `HEAD`, `PUT` und `DELETE`. Die Webmethoden `GET` und `POST` sind die am weitesten verbreiteten. Mit `GET` wird die mit der URI beschriebene Ressource abgefragt. `GET` impliziert, dass diese Abfrage seiteneffektfrei ist und somit Caches verwendet werden können, um HTTP-Responses zwischenzuspeichern. Die Webmethode `POST` wird dagegen verwendet, wenn eine Nachricht an eine Ressource geschickt und von dieser verarbeitet werden soll, wobei die Verarbeitung auch Seiteneffekte haben darf.

Die HTTP-Bindung legt für die Verwendung von SOAP fest, welche Webmethoden zu verwenden sind. Mit dem Request-/Response-Kommunikationsmuster muss stets die Webmethode `POST` verwendet werden, mit dem SOAP-Response-Kommunikationsmuster die Webmethode `GET`. Diese Wahl erscheint offensichtlich. Bei der Verwendung der Webmethode `GET` wird eine Ressource seiteneffektfrei abgefragt, ebenso bei Verwendung des SOAP-Response-Kommunikationsmusters. Dagegen wird bei der Webmethode `POST` eine an die Ressource geschickte Nachricht von dieser mit Seiteneffekten verarbeitet, was auch das Request-/Response-Kommunikationsmuster erlaubt.

Fehlersituationen werden im HTTP-Response in einer für Menschen lesbaren Form und maschinenlesbar als dreistellige numerische Statuscodes angegeben. Die Statuscodes von 200 bis 299 bedeuten, dass die Ausführung erfolgreich war. Statuscodes von 400-499 weisen auf einen Fehler im HTTP-Request hin, während Statuscodes von 500-599 für vom Request unabhängige Fehler bei der Verarbeitung stehen. Die HTTP-Bindung legt fest, welche Statuscodes bei der Verwendung mit SOAP anzugeben sind.

Abbildung 22 bis Abbildung 24 zeigen drei Beispiele von HTTP-Nachrichten nach der HTTP-Bindung für SOAP. Abbildung 22 zeigt einen HTTP-Request, der als Nutzlast den SOAP-Request aus Abbildung 18 enthält. Verwendet wurde hier das Request-/Response-Kommunikationsmuster. Daher wird die Webmethode `POST` verwendet. Die

Nutzlast wird durch die HTTP-Header `Content-Type` und `Content-Length` beschrieben. Der Header `Content-Type` zeigt mit dem Wert `application/soap+xml` an, dass die Nutzlast eine nach XML 1.0 serialisierte SOAP-Nachricht ist. Der gleiche Typ wird auch im HTTP-Header `Accept` angegeben, um anzuzeigen, welcher Typ für die Nutzlast im HTTP-Response erwartet wird. Der Header `Content-Length` gibt an, wie lang die Nutzlast ist.

In der ersten Zeile wird der hintere Teil der URI angegeben. Bei der URI handelt es sich um eine URL. In der ersten Zeile wird nur der Teil der URL angegeben, der hinter dem Hostnamen und der Portnummer folgt. Hostname und die optionale Portnummer werden in der Version 1.1 von HTTP im HTTP-Header Host angegeben.

```
POST /example.org/SOAPService?Ressource=ABC HTTP/1.1
Host: example.org
Accept: application/soap+xml
Content-Type: application/soap+xml
Content-Length: 367

<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope"
  <env:Body>
    <my:D xmlns:my="http://example.org/2002/EncodingSample"
          env:encodingStyle="http://www.w3.org/2002/06/soap-encoding">
      <my:A>ABC</my:A>
      <my:B>DEF</my:B>
      <my:C>GHI</my:C>
    </my:D>
  </env:Body>
</env:Envelope>
```

Abbildung 22: HTTP-Request mit SOAP-Nachricht für Webmethode `POST`

In der URI ist zu erkennen, dass sie den Wert des Parameters `A` mit enthält. Sie ist dort als `Ressource=ABC` kodiert. Dazu sei angenommen, dass der Parameter `A` die Ressource identifiziert, auf die sich die Operation bezieht. Wäre dieser Parameter der einzige, würden also die Parameter `B` und `C` fehlen und wäre die Abfrage der Ressource seiteneffektfrei, dann könnte die Webmethode `GET` und das SOAP-Response-Kommunikationsmuster verwendet werden. Abbildung 23 zeigt den dazu verwendeten HTTP-Request. In ihm fehlt die Nutzlast und die ihn beschreibenden HTTP-Header.

```
GET /example.org/SOAPService?Ressource=ABC HTTP/1.1
Host: example.org
Accept: application/soap+xml
```

Abbildung 23: HTTP-Request für Webmethode `GET`

Auf beide HTTP-Requests wird mit einem HTTP-Response geantwortet, der einen SOAP-Response enthält. Abbildung 24 zeigt, wie er aussehen könnte. Der eingebettete SOAP-Response wurde bereits in Abbildung 21 gezeigt. Die erste Zeile des HTTP-Responses enthält neben der Version von HTTP den Statuscode in für Maschinen und Menschen verständlicher Form. Der Statuscode hat den Wert 200, was eine fehlerfreie Ausführung anzeigt. Die für Menschen lesbare Form zeigt das mit der Zeichenkette „`OK`“ an. Im Request sind zwei HTTP-Header angegeben, die die Nutzlast beschreiben.

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml
Content-Length: 487

<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope">
  <env:Body>
    <my:DResponse
           xmlns:my="http://example.org/2002/EncodingSample"
           xmlns:rpc="http://www.w3.org/2002/06/soap-rpc"
           env:encodingStyle="http://www.w3.org/2002/06/soap-encoding">
      <rpc:result>my:R</rpc:result>
      <my:R>42</my:R>
      <my:B>DEF</my:B>
      <my:E>GHI</my:E>
    </my:DResponse>
  </env:Body>
</env:Envelope>
```

Abbildung 24: HTTP-Response mit SOAP-Nachricht

4 Spezifikation von Webservices mit WSDL

Entwickler von Webclient und Webservice müssen das gleiche Verständnis darüber haben, welche SOAP-Nachrichten zwischen Webclient und Webservice ausgetauscht werden, also das gleiche Verständnis über die Schnittstelle zwischen diesen. Weicht das Verständnis voneinander ab, ist das eine Quelle von Interoperabilitätsproblemen. Daher wird eine Sprache benötigt, mit der solche Schnittstellen beschrieben werden. Damit alle Entwickler sowie Werkzeuge zur Entwicklung von Webkomponenten diese Sprache verstehen, ist es wichtig, sich auf eine zu einigen.

Die Sprache, für die am ehesten ein Konsens besteht, ist die Web Services Description Language (WSDL), die sich beim W3C [W3C 02a] in der Standardisierung befindet. Bereits für die Version 1.1, die als W3C-Note [Christensen 01] in Kooperation der Unternehmen Ariba, IBM und Microsoft entstand, besteht in der Industrie ein breite Unterstützung. Das gleiche ist auch für die Version 1.2 zu erwarten, wenn für sie der Standardisierungsprozess abgeschlossen ist. Daher soll WSDL in dieser Arbeit als die Sprache angesehen werden, mit der heute Webkomponenten spezifiziert werden. Als Grundlage für spätere Untersuchungen wird sie in diesem Kapitel eingeführt.

4.1 Überblick

WSDL erlaubt mehr, als nur Webservices zu beschreiben, die mit SOAP und einem darunter liegenden Protokoll wie HTTP kommunizieren. Wie SOAP ist auch WSDL sehr breit angelegt. Es erlaubt Webservices zu beschreiben, die mit Webclients Nachrichten beliebiger Datenformate mit beliebigen Protokollen austauschen. Dazu wird der Austausch von Nachrichten zunächst abstrakt beschrieben. Mit Hilfe von protokollspezifisch definierten, sogenannten Bindungen, kann dann die Abbildung auf ein konkretes Protokoll sowie dort verwendete Nachrichtenformate beschrieben werden.

Zur Beschreibung von Nachrichten wird ein Typsystem benötigt, um Werte in Nachrichten zu typisieren und Werte zu komplexeren Werten zusammenzufassen. WSDL enthält kein eigenes Typsystem, sondern erlaubt die Verwendung beliebiger anderer Typsysteme.

Für die Interoperabilität ist es problematisch, dass in WSDL ein beliebiges Typsystem verwendet werden darf. Der aktuelle Arbeitsentwurf des WSDL-Standards [Chinnici 02] stellt dazu fest, dass die Verwendung von XML-Schema als Typsystem für WSDL zu maximaler Interoperabilität führt. Obwohl auch andere Typsysteme verwendet werden dürfen, ist XML-Schema das einzige, für das im Arbeitsentwurf des WSDL-Standards beschrieben ist, wie es verwendet wird. Dabei kann XML-Schema nicht nur verwendet werden, um anwendungsspezifische XML-Fragmente von z. B. SOAP-Nachrichten zu beschreiben. Ebenso können mit ihm Datenstrukturen von Nachrichten beschrieben werden, die nicht XML-Fragmente bzw. XML-Dokumente sind.

Aufgrund seiner wichtigen Rolle von XML-Schema für die Beschreibung von Webservices mit WSDL soll es nachfolgend beschrieben werden. Erst danach wird WSDL selbst im Detail vorgestellt.

4.2 XML-Schema

XML-Schema ist das Typsystem für XML. Gleichzeitig wird mit dieser Sprache die Struktur von XML-Dokumenten beschrieben und Namensräume formal definiert. Beschreibungen von Namensräumen befinden sich in XML-Dokumenten, den sogenannten Schemadokumenten. In ihnen werden Elemente und Attribute deklariert und Typen definiert. Typen können von anderen abgeleitet sein. Außerdem stellt XML-Schema eine Reihe vordefinierter Typen zur Verfügung.

XML-Schema wurde wie WSDL vom W3C standardisiert. Die Standardisierung der Version 1.0 ist abgeschlossen. Der Standard besteht aus zwei Teilen. Der erste [Thompson 01] beschreibt die Sprachelemente von XML-Schema, während der zweite [Brion 01] das Typsystem einschließlich vordefinierter Typen festlegt. Der Standard wird durch [Fallside 01] als verständliches Tutorial ergänzt.

Außer im Standard ist XML-Schema in unterschiedlichen Quellen beschrieben. Die Website [RefsnesData 02] enthält ein Tutorial. Die Referenz [Skonnard 02] wurde für diese Arbeit als Quelle verwendet. Quelle waren außerdem die Kursunterlagen [Developmentor 01].

4.2.1 Ziele von XML-Schema

Mit XML-Schema werden mehrere Ziele gemeinsam erreicht. Es ist Typsystem für XML, so dass Werte in XML-Dokumenten als typisiert betrachtet werden können, statt ohne XML-Schema nur als Zeichenketten. Gleichzeitig wird die Struktur in XML-Dokumenten festgelegt. Schließlich werden mit XML-Schema Namensräume formal definiert, die z. B. die Namen von Elementen und Attributen enthalten.

XML-Schema ist also ein Typsystem für XML. Elemente und Attribute haben Typen, die bestimmen, welche Werte sie enthalten dürfen. Für Elemente werden außerdem ihre Kindelemente und Attribute festgelegt. Werte sind damit nicht mehr nur Zeichenketten, wie es ohne XML-Schema der Fall ist. Es ist festgelegt, wie sie zu interpretieren sind. Hat ein Attribut z. B. einen numerischen Typ, macht es für seinen Wert keine Unterschied, ob „100“ oder „0000100“ angegeben wird. Ohne XML-Schema handelt sich dagegen um unterschiedliche Werte.

Anwendungen können diese Typisierung nutzen. APIs zum Zugriff auf XML-Dokumente können statt Zeichenketten typisierte Werte liefern. Das vereinfacht die Implementierung von Anwendungen, weil auf Code für eine explizite Typwandlung verzichtet werden kann. Generatoren können die Typbeschreibungen lesen, um z. B. automatisch Klassen im objektorientierten Sinne zu generieren, mit denen auf XML-Dokumente zugegriffen werden kann. XML-Editoren könnten außerdem Anwendern nur Eingaben erlauben, die der Typisierung entsprechen.

Durch die Möglichkeit für Elemente deren Kindelemente und Attribute anzugeben, wird mit XML-Schema eine Struktur in XML-Dokumenten festgelegt. Daher wird XML-Schema häufig als Nachfolger der in [Bray 00] beschriebenen Document-Type-Declarations (DTDs) angesehen, mit denen das ebenfalls möglich ist. Anders als bei DTDs wird jedoch nicht für XML-Dokumente die Struktur angegeben, sondern lediglich ein Vokabular von Elementen und Attributen definiert, die in XML-Dokumenten verwendet werden dürfen.

Anwendungen können mit Hilfe von Validatoren prüfen, ob das in XML-Dokumenten verwendete Vokabular korrekt verwendet wird. Dabei wird die Struktur in den XML-Dokumenten geprüft und ob Werte in den festgelegten Wertemengen enthalten sind.

Ohne Validation müssten Anwendungen Code enthalten, der die erforderlichen Prüfungen durchführt.

XML-Schema kann auch als Sprache angesehen werden, mit der Namensräume von XML formal definiert werden. Die in [Bray 99] beschriebenen Namensräume erlauben Eindeutigkeit von in XML-Dokumenten verwendeten Namen, die z. B. Elemente und Attribute bezeichnen. Hierzu wird der Namensraum mit einer URI eindeutig identifiziert und in ihm sogenannte lokale Namen definiert. Das Paar aus URI und lokalem Namen ist eindeutig und wird abgekürzt als qualifizierter Name notiert, wozu die URIs an sogenannte Präfixe gebunden werden. Die Kenntnis dieser Vorgehensweise sei dem Leser dieser Arbeit unterstellt. Mit XML-Schema kann definiert werden, welche lokalen Namen ein Namensraum enthält und wie sie verwendet werden können.

4.2.2 Instanz- und Schemadokumente

Jeweils ein Namensraum wird mit XML-Schema in einem Schemadokument festgelegt. Es enthält unter anderem Deklarationen von Elementen, Attributen und Typen. Elemente und Attribute in XML-Dokumenten werden ihren Deklarationen in Schemadokumenten über ihre qualifizierten Namen zugeordnet. Von Elementen kann auch direkt auf einen im Schemadokument definierten Typ verwiesen werden. Damit Definitionen aus einem Schemadokument in einem anderen verwendet werden können, müssen sie importiert werden.

Durch Schemadokumente definierte XML-Dokumente werden als Instanzdokumente bezeichnet. Dieser Begriff ist für eine klare Darstellung erforderlich, weil Schemadokumente auch selbst XML-Dokumente sind. Damit steht der Begriff XML-Dokument sowohl für das Instanz- als auch das Schemadokument. Jedes Schemadokument definiert genau einen Namensraum. Sind in einem Instanzdokument verwendete Namen in unterschiedlichen Namensräumen, wurden die Namen in unterschiedlichen Schemadokumenten definiert.

Das Rootelement eines Schemadokumentes hat den Namen `xsd:schema`. Das Präfix `xsd` steht in dieser Arbeit für den Namensraum `http://www.w3.org/2001/XMLSchema`, in dem sich die Namen zur Definition von Schemadokumenten befinden. Im Attribut `xsd:schema/@targetNamespace`[16] wird die URI des Namensraumes angegeben, der durch das Schemadokument definiert wird, wie in Abbildung 26 gezeigt.

Mit Kindelementen des Elementes `xsd:schema` werden XML-Elemente[17] und Attribute deklariert sowie Typen definiert, von denen es einfache und komplexe gibt. In Deklarationen von XML-Elementen und Attributen wird dabei auf deren Typ verwiesen. Auch Typen werden mit qualifizierten Namen identifiziert, die sich im Namensraum befinden, der durch das Schemadokumente definiert wird. Unterschieden werden einfache von komplexen Typen. Komplexe Typen werden für XML-Elemente verwendet und beschreiben deren Attribute und Kindelemente. Einfache Typen beschreiben dagegen eine Menge von Zeichenketten und deren Interpretation. Es gibt einfache Typen z. B. für Zahlen, Uhrzeiten und URIs. Sie können sowohl für XML-Elemente als auch für Attribute verwendet werden, wie in Abbildung 25 veranschaulicht.

---

[16] In dieser Arbeit werden XPath-Ausdrücke verwendet, um prägnant über Elemente und Attribute in XML-Dokumenten sprechen zu können. Das wird genauer im A erläutert. In diesem Fall steht „Attribut `xsd:schema/@targetNamespace`" für das Attribut `targetNamespace` im Element `xsd:schema`.

[17] Für die Klarheit der Darstellung sollen nachfolgend Elemente in Instanzdokumenten als XML-Elemente bezeichnet werden und solche in Schemadokumenten weiterhin einfach als Elemente.

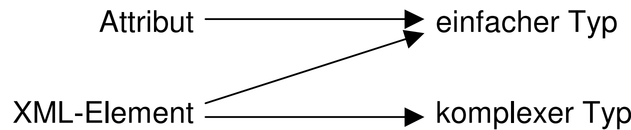

Abbildung 25: Verwendbarkeit einfacher bzw. komplexer Typen für Attribute bzw. Elemente

XML-Elemente und Attribute werden durch ihre qualifizierten Namen ihren Deklarationen in Schemadokumenten zugeordnet. Der qualifizierte Name enthält die URI, die auch im Schemadokument im Attribut `targetNamespace` angegeben ist. Das identifiziert das Schemadokument mit der Deklaration des Namen. Der im qualifizierten Namen enthaltene lokale Name ist in einer Deklaration des XML-Elementes bzw. Attributes enthalten. Abbildung 26 veranschaulicht das an einem Beispiel. Der qualifizierte Name `nwst:Public` enthält den lokalen Namen Public. Im Schemadokument ist dieser in der Deklaration des XML-Elementes im Attribut `xsd:element/@name` enthalten. Der dort angegebene Typ `xsd:boolean` bestimmt die gültigen Werte der im XML-Element enthaltenen Zeichenkette.

Schemadokument:

```
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://example.org/NewsService020917/Types">
  ...
  <xsd:element name="Public" type="xsd:boolean" />
  ...
</xsd:schema>
```

Instanzdokument:

```
<nwst:Public xmlns:nwst="http://example.org/NewsService020917/Types">
  true
</nwst:Public>
```

Abbildung 26: Beispiel für Verweis von Instanzdokument auf Schemadokument

Alternativ zu dieser Vorgehensweise kann auch der Typ eines XML-Elementes im Instanzdokument angegeben werden. Hierzu wird dem XML-Element das Attribut `xsi:type` zugefügt, das als Wert den qualifizierten Namen des Typs enthält. Das Präfix `xsi` bezeichne den Namensraum `http://www.w3.org/2001/XMLSchema-instance`, in dem sich alle Namen befinden, die XML-Schema für Instanzdokumente definiert.

Um in einem Schemadokument Definitionen anderer Schemadokumente wiederzuverwenden, müssen deren Namensräume importiert werden. Hierzu wird das Element `xsd:import` verwendet, in dem als Attribute die URI des importierten Namensraumes sowie optional die URI angegeben wird, die den Ort des Schemadokumentes bezeichnet.

4.2.3 XML-Elemente und Attribute

In Deklaration von XML-Elementen und Attributen werden deren Namen Typen zugeordnet. Abhängig vom Ort im Schemadokument sind solche Deklarationen global sichtbar oder nur im Kontext eines Typs gültig. Ebenso können Typen global definiert werden oder nur lokal für die Deklaration eines XML-Elementes bzw. Attributes gültig sein. Mit Nullwerten kann ausgedrückt werden, dass ein XML-Element abweichend von seinem Typ weder einen Wert noch Kindelemente oder Attribute hat.

XML-Elemente werden mit dem Element `xsd:element` deklariert, Attribute mit dem Element `xsd:attribute`. In der Deklarationen eines XML-Elementes enthält das Attribut `xsd:element/@name` dessen lokalen Namen, das Attribut `xsd:element/@type` optional den qualifizierten Namen seines Typs, wie bereits in Abbildung 26 gezeigt. Attribute werden analog definiert. Das Attribut `xsd:attribute/@name` enthält seinen Namen, im Attribut `xsd:attribute/@type` kann sein Typ angegeben werden.

Auf Typen kann nicht nur über deren qualifizierte Namen verweisen werden, Typen können auch lokal für die Deklaration des XML-Elementes bzw. Attributes definiert werden. Hierzu wird dem Element `xsd:element` bzw. `xsd:attribute` die Definition des Typs direkt als Kindelement zugefügt. Da auf den Typ nicht mit einem Namen verwiesen wird, wird dieser nicht benannt. Daher wird er als anonymer Typ bezeichnet. Umgekehrt werden Typen, die unabhängig von einer Deklaration eines XML-Elementes oder Attributes, global als Kindelemente des Rootelementes `xsd:schema` definiert werden, als benannte Typen bezeichnet. Nur benannte Typen können für mehrere Deklarationen wiederverwendet werden.

Auch XML-Elemente und Attribute können global oder aber lokal innerhalb der Definition eines Typs deklariert werden. Globale Deklarationen sind wieder Kindelemente des Rootelementes `xsd:schema`. Auf sie kann in Definitionen von Typen über ihre qualifizierten Namen verwiesen werden. Lokale Deklarationen sind dagegen nur innerhalb der Definition des Typs gültig, der die Deklaration als Kindelement zugefügt ist. Nur innerhalb von XML-Elementen dieses Typs dürfen so deklarierte Attribute und Kindelemente im Instanzdokument verwendet werden.

Anders als bei anonymen Typen haben lokal deklarierte XML-Elemente und Attribute immer einen Namen, sie sind also nicht anonym. Das ist erforderlich, weil ihr Name im Instanzdokument angegeben wird. Lokal deklarierte Namen sind jedoch nicht Teil des Namensraumes, der durch das Schemadokument definiert wird. Im Instanzdokument wird ihr lokaler Name ohne Präfix angegeben. Sollen solche Namen Teil des Namensraumes sein, muss der lokalen Deklaration das Attribut `form` mit dem Wert `qualified` zugefügt werden.

Für ein XML-Element können Nullwerte erlaubt werden. Hierzu muss in dessen Deklaration dem Attribut `xsd:element/@nillable` der Wert `true` zugeordnet werden. Dann darf im Instanzdokument das deklarierte XML-Element einen Nullwert haben. Das bedeutet, dass gemäß dem Typ erforderliche Werte, Kindelemente und Attribute entfallen und statt dessen das Attribut `xsi:nil` zugefügt wird, um den Nullwert anzuzeigen.

4.2.4 Vordefinierte Typen

XML-Schema stellt eine Menge von vordefinierten, einfachen Typen zur Verfügung. Sie bilden eine Hierarchie, in die sich auch anwendungsspezifisch definierte einfache und komplexe Typen einordnen. Die vordefinierten Typen sind die Basis für anwendungsspezifisch definierte. Es gibt allgemeinere vordefinierte Typen z. B. für Zeichenketten und Zahlen ebenso wie XML-spezifisches z. B. für qualifizierte Namen oder URIs.

Die qualifizierten Namen aller vordefinierten Typen befinden sich im Namensraum von XML-Schema `http://www.w3.org/2001/XMLSchema`. Nachfolgend wird ihnen daher das Präfix `xsd` vorangestellt.

Ein Ausschnitt der Hierarchie der vordefinierten Typen wird in Abbildung 27 gezeigt. Bei den gezeigten Typen ist die Wertemenge von Nachfahren in der Hierarchie stets eine Teilmenge der Wertemenge ihrer Vorfahren. Abbildung 27 zeigt nur beispielhaft

wesentliche Typen. Ausgelassene Teile sind mit drei Punkten („...“) angedeutet. Der zweite Teil des Standards von XML-Schema [Brion 01] enthält eine ähnliche, aber vollständige Abbildung in seinem dritten Kapitel. Dort und in [Skonnard 02] sind auch alle vordefinierten Typen im Detail beschrieben.

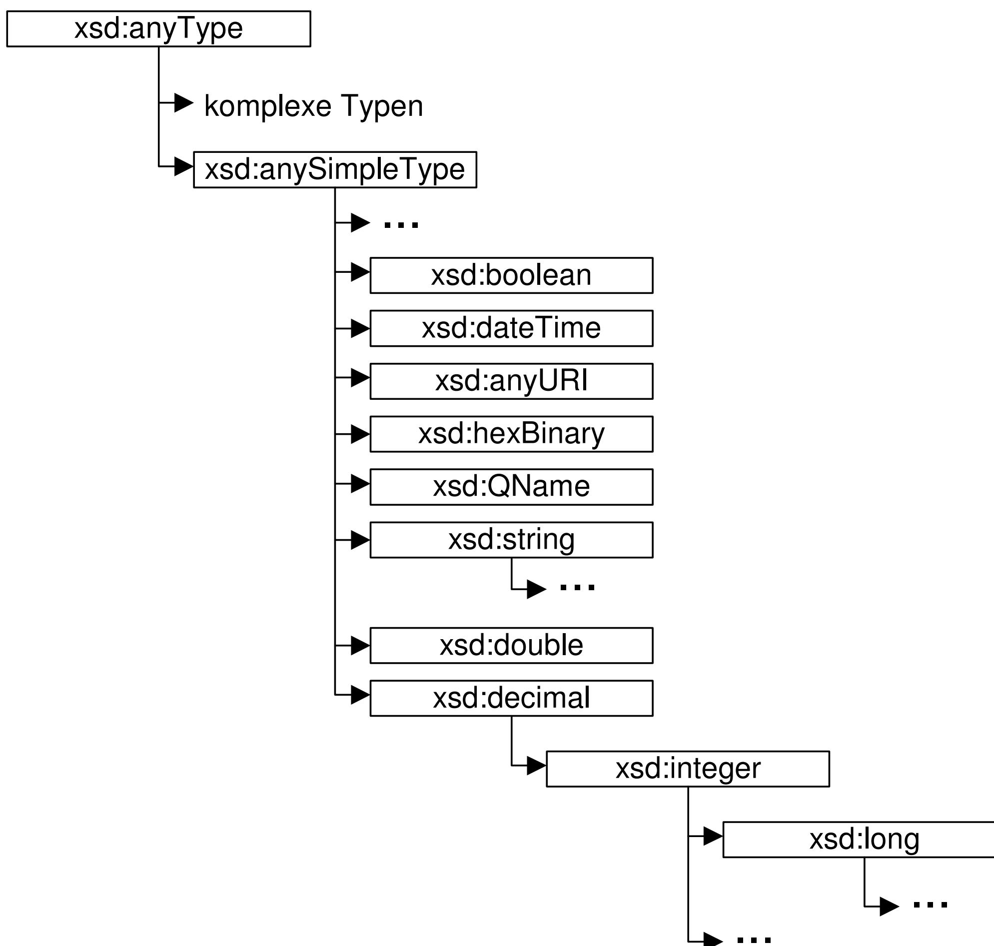

Abbildung 27: Ausschnitt der Hierarchie vordefinierter Typen von XML-Schema

Die Wurzel der Hierarchie wird durch den Typ `xsd:anyType` gebildet. Er ist der einzige Typ in XML-Schema, der keinen Vorfahren hat. Er selbst ist Vorfahre von allen anderen Typen, sowohl einfachen als auch komplexen. Sein Untertyp `xsd:anySimpleType` ist Vorfahre aller einfachen Typen. Ein analoger Typ für komplexe Typen exsistiert nicht.

Eine Reihe von Typen sind direkte Nachfahren von `xsd:anySimpleType`. Hierzu gehören `xsd:boolean`, der in seiner Wertemenge lediglich die Werte `true` und `false` enthält. Außerdem gibt es verschiedene Typen für Zeit- und Datumsangaben, von denen in Abbildung 27 nur `xsd:dateTime` gezeigt ist. Zur Repräsentation von URIs wird der Typ `xsd:anyURI` verwendet. Für binäre Daten in Instanzdokumente gibt es zwei Typen `xsd:hexBinary` und `xsd:base64Binary`, die sich darin unterscheiden, wie die binären Daten im Instanzdokument kodiert sind.

Ein direkter Nachfahre von `xsd:anySimpleType` ist auch der Typ `xsd:QName` mit dem qualifizierte Namen repräsentiert werden. Ein qualifizierter Name ist ein Paar aus einer URI eines Namensraumes und einem lokalen Namen. Trotzdem wird er in Instanzdokumenten als Zeichenkette aus einem Präfix, einem Doppelpunkt und dem lokalen Namen repräsentiert. Das Präfix wird mit den in XML üblichen Mechanismen auf die URI des Namensraumes abgebildet.

Zur Repräsentation von Zeichenketten und Bezeichnern gibt es eine Reihen von Typen: `xsd:string` und seine Nachfahren. Der Typ `xsd:string`, direkter Nachfahre von `xsd:anySimpleType`, hat als Wertemenge beliebige Zeichenketten. In seinen Nachfahren haben Leer- und Tabulatorzeichen sowie Zeilenumbrüche nur die Bedeutung als Trennzeichen zwischen Bezeichnern. Mit solchen Typen werden häufig Namen und Bezeichner repräsentiert.

Auch die Typen `xsd:ID` und `xsd:IDREF` sind Nachfahren von `xsd:string` und werden für Verweise zwischen Elementen im Instanzdokument verwendet. Damit auf ein Element verwiesen werden kann, muss ihm ein Attribut mit Typ `xsd:ID` zugefügt werden. Dessen Wert muss im Instanzdokument unter allen Werten dieses Typs eindeutig sein. Für Verweise auf so bezeichnete Elemente werden Attribute vom Typ `xsd:IDREF` verwendet. Es enthält den Wert des Attributs mit Typ `xsd:ID`, auf dessen Element verwiesen wird.

Verschiedene Typen gibt es zur Repräsentation von Zahlen. Die drei Typen `xsd:float`, `xsd:double` und `xsd:decimal` sind direkte Nachfahren von `xsd:anySimpleType`, die übrigen Nachfahren von `xsd:decimal`. Dabei dienen `xsd:float` und `xsd:double` zur Repräsentation von Fließkommazahlen mit beschränkter Genauigkeit. Mit `xsd:decimal` können dagegen Zahlen mit einer endlichen Anzahl von Vor- und Nachkommastellen präzise gespeichert werden. Die Wertemenge ist also die Menge der rationalen Zahlen aus der Mathematik. In Nachfahren dieses Typs wird die Wertemenge weiter beschränkt. Der Typ `xsd:integer` erlaubt z. B. ganze Zahlen beliebiger Länge, dessen Nachfahre `xsd:long` beschränkt die Wertemenge auf ganze Zahlen von -2^{63} bis $(2^{63} - 1)$ und dessen Nachfahren noch weiter.

4.2.5 Ableiten einfacher Typen

Aus vorhandenen einfachen Typen können in XML-Schema neue definiert werden. Es können Typen für Listen von Werten vorhandener Typen definiert werden und Typen deren Wertemenge die Vereinigung von Wertemengen andere Typen sind. Außerdem gibt es die häufig verwendete Restriction, mit der ein neuer Typ aus einem vorhandenen abgeleitet wird, dessen Wertemenge eine Teilmenge des vorhandenen Typs ist.

Für die Definition von einfachen Typen wird im Schemadokument das Element `xsd:simpleType` angegeben. Es ist Kindelement vom Element `xsd:schema`, wenn ein benannter Typ definiert werden soll. Dann wird im Attribut `xsd:simpleType/@name` sein lokaler Name angegeben. Ansonsten ist das Element `xsd:simpleType` Kindelement der Deklaration, für das der Typ anonym definiert wird. Kindelemente von `xsd:simpleType` bestimmen, was den definierten Typ ausmacht.

Hat `xsd:simpleType` das Kindelement `xsd:list`, wird festgelegt, dass Werte des neu definierten Typs Listen von Werten eines vorhandenen Typs sind. Der qualifizierte Name des vorhandenen Typs wird in einem Attribut angegeben. Im Instanzdokument enthalten Listen dann Werte des vorhandenen Typs getrennt durch Leerzeichen, Tabulatorzeichen oder Zeilenumbrüche.

Ist das Kindelement von `xsd:simpleType` dagegen `xsd:union`, wird ein Typ definiert, dessen Wertemenge die Vereinigung der Wertemengen anderer Typen ist. In einem Attribut werden die qualifizierten Namen der Typen angegeben, deren Wertemengen vereinigt werden sollen. Außerdem können weitere anonyme Typen auch als Kindelemente zugefügt werden.

Häufiger als `xsd:list` und `xsd:union` wird `xsd:restriction` verwendet, mit der ein neuer Typ aus einem vorhandenen abgeleitet wird, wobei die Wertemenge des neuen

eine Teilmenge der des vorhandenen ist. Das wird als Restriction bezeichnet. Der Name des vorhandenen Typs (Basistyp), wird im Attribut `xsd:restriction/@base` angegeben. Kindelemente sind sogenannte Facetten, mit denen angegeben wird, in welcher Weise die Wertemenge beschränkt werden soll.

Unterschiedliche Facetten stehen zur Verfügung, mit denen die Anzahl der Zeichen bzw. Ziffern von Werten beschränkt oder für numerische, Zeit- oder Datumstypen minimale oder maximale Werte angegeben werden können. Abbildung 28 zeigt das an einem Beispiel. Es wird ein Typ für Zeichenketten definiert, die mindestens ein und höchstens 50 Zeichen lang sind. Der Basistyp ist `xsd:string`. Mit der Facette `xsd:minLength` wird festgelegt, wie lang ein Wert mindestens sein muss, mit `xsd:maxLength` seine maximale Länge.

```
<xsd:simpleType name="restrictedString">
  <xsd:restriction base="xsd:string">
    <xsd:minLength value="1" />
    <xsd:maxLength value="50" />
  </xsd:restriction>
</xsd:simpleType>
```

Abbildung 28: Beispiel für Restriction zur Festlegung der Zeichenkettenlänge in XML-Schema

Aufzählungstypen können mit der Facette `xsd:enumeration` definiert werden. Jeder in der Wertemenge erlaubte Wert des Basistyps wird dann explizit angegeben. Für jeden gibt es ein Kindelement `xsd:restriction/xsd:enumeration`[18], der Wert ist im Attribut `value` angegeben. Ein Beispiel wird in Abbildung 29 gezeigt. Im dort definierten Typ wird der Basistyp `xsd:string` auf die Werte `Business`, `Culture`, `Politics` und `Sports` eingeschränkt.

```
<xsd:simpleType name="categories">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Business" />
    <xsd:enumeration value="Culture" />
    <xsd:enumeration value="Politics" />
    <xsd:enumeration value="Sports" />
  </xsd:restriction>
</xsd:simpleType>
```

Abbildung 29: Beispiel für Restriction zur Definition eines Aufzählungstyps in XML-Schema

Die Facette `xsd:pattern` erlaubt, die Wertemenge des Basistyps auf solche Werte zu beschränken, die einem regulären Ausdruck entsprechen. Ein regulärer Ausdruck ist eine Zeichenfolge, mit der Mengen von Werten definiert werden kann. Allgemein werden reguläre Ausdrücke in [Engesser 88] beschrieben, die speziell für XML-Schema verwendete Syntax im Anhang F des zweite Teils des Standards von XML-Schema [Brion 01]. Abbildung 30 zeigt ein Beispiel für eine Definition mit einem regulärem Ausdruck. Der Basistyp `xsd:string` wird auf die Zeichenketten beschränkt, die dem folgenden Muster entsprechen: Sie fangen an mit der Zeichenkette „20“, dann folgen zwei Ziffern (`\d{2}`), danach ein Bindestrich (`\-`),wieder zwei Ziffern, ein Bindestrich und noch einmal zwei Ziffern.

---

[18] „Kindelement `xsd:restriction/xsd:enumeration`“ bezeichnet das Kindelement `xsd:enumeration` im bekannten Element `xsd:restriction`. XPath-Ausdrücke dieser Art werden in dieser Arbeit verwendet, um über Elemente und Attribute in XML-Dokumenten sprechen zu können. Das wird genauer im A erläutert.

```
<xsd:simpleType name="messageDate">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="20\d{2}\-\d{2}\-\d{2}" />
  </xsd:restriction>
</xsd:simpleType>
```

Abbildung 30: Beispiel für Restriction mit einem regulären Ausdruck in XML-Schema

4.2.6 Definition komplexer Typen

Komplexe Typen können als Komposition von XML-Elementen und Attributen definiert werden. Alternativ können sie ebenso wie einfache Typen aus vorhandenen abgeleitet werden, was im nachfolgenden Kapitel 4.2.7 vorgestellt wird. Wird ein Typ aus XML-Elementen und Attributen komponiert, wird festgelegt, welche Kindelemente ein XML-Element dieses Typs wie oft und in welcher Reihenfolge haben darf oder muss. Außerdem kann festgelegt werden, welche Attribute das XML-Element haben darf oder muss.

Komplexe Typen werden in einem Schemadokument stets mit dem Element `xsd:complexType` definiert. Wie bei einfachen Typen ist es Kindelement von `xsd:schema`, um einen benannten Typ zu definieren. Dann wird sein lokaler Name im Attribut `xsd:complexType/@name` angegeben. Ist der komplexe Typ anonym, ist das Element `xsd:complexType` Kindelement der Deklaration, für die der Typ definiert wurde. Kindelemente bestimmen wieder, was den komplexen Typ ausmacht.

Zentrale Begriffe bei der Komposition von komplexen Typen sind: Partikel, Kompositoren und Modelgroups. Partikel beschreiben im einfachsten Fall, dass im Instanzdokument ein bestimmtes Kindelement vorhanden sein darf oder muss. Hierzu kann eine lokale Deklaration eines XML-Elementes angegeben oder auf eine globale verwiesen werden. Mit Hilfe von drei Kompositoren können mehrere Partikel zu Modelgroups zusammengefasst werden. Die drei Kompositoren beschreiben, dass die Partikel hintereinander in angegebener bzw. beliebiger Reihenfolge enthalten sein müssen oder nur genau eines der Partikel. Kompositoren sind selbst wieder Partikel, so dass auch sie rekursiv mit Kompositoren wieder zu Modelgroups zusammengefasst werden können. Ein Kompositor ist Kindelement von `xsd:complexType`. Die durch ihn beschriebene Modelgroup beschreibt, welche Kindelemente ein XML-Element des beschriebenen komplexen Typs hat.

4.2.6.1 Kompositoren

Der gebräuchlichste Kompositor ist `xsd:sequence`, weniger häufig werden `xsd:choice` und `xsd:all` verwendet. Partikel werden den Kompositoren als Kindelemente zugefügt. Bei `xsd:sequence` werden sie in der Reihenfolge zugefügt, in der das durch sie beschriebene im Instanzdokument vorhanden sein muss. Abbildung 31 und Abbildung 32 veranschaulichen das an einem Beispiel. Abbildung 31 zeigt ein Schemadokument mit einem komplexen Typ der `xsd:sequence` verwendet. Als Partikel sind drei lokale Deklarationen für die Elemente `a`, `b` und `c` enthalten. Im Schemadokument befindet sich zusätzlich eine Deklaration für ein Element `my:d`, dass den komplexen Typ verwendet. Das in Abbildung 32 gezeigte Instanzdokument hat dieses Element als Rootelement. Für jedes Partikel des Kompositors `xsd:sequence` im Schemadokument hat das Element `my:d` ein Kindelement in der gleichen Reihenfolge.

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:my="http://example.org/Sample"
            targetNamespace="http://example.org/Sample" >

  <xsd:complexType name="exampleType">
    <xsd:sequence>
      <xsd:element name="a" type="xsd:string" />
      <xsd:element name="b" type="xsd:long" />
      <xsd:element name="c" type="xsd:boolean" />
    </xsd:sequence>
  </xsd:complexType>

  <xsd:element name="d" type="my:exampleType" />
</xsd:schema>
```

Abbildung 31: Beispiel für `xsd:sequence` in XML-Schema

```
<?xml version="1.0" encoding="UTF-8" ?>
<my:d xmlns:my="http://example.org/Sample">
  <a>ABCDE</a>
  <b>42</b>
  <c>false</c>
</my:d>
```

Abbildung 32: Beispiel für Instanzdokument zum Schemadokument in Abbildung 31

Der wenig verwendete Kompositor `xsd:all`, der große Ähnlichkeit mit `xsd:sequence` hat, unterschiedet sich von diesem darin, dass die Reihenfolge der Partikel keine Rolle spielt. Für jedes Partikel muss das durch ihn beschriebene im Instanzdokument vorhanden sein, anders als bei `xsd:sequence` jedoch in beliebiger Reihenfolge.

Der Kompositor `xsd:choice` wird verwendet, um Alternativen im Instanzdokument auszudrücken. Auch er enthält Partikel als Kindelemente. Nur das von einem Partikel beschriebene muss im Instanzdokument enthalten sein. Abbildung 33 zeigt als Beispiel eine Erweiterung des komplexen Typs aus Abbildung 31. Dem Kompositor `xsd:sequence` wird der Kompositor `xsd:choice` als Partikel zugefügt. Durch den Kompositor `xsd:sequence` müssen im Instanzdokument zunächst die Kindelemente `a` und `b` vorhanden sein, danach das, was der Kompositor `xsd:choice` beschreibt. Er beschreibt, dass genau eines der deklarierten XML-Elemente `c`, `e` oder `f` als Kindelement vorhanden sein muss. Damit ist das Instanzdokument in Abbildung 32 weiterhin gültig, zusätzlich jedoch auch das Instanzdokument in Abbildung 34, bei dem statt einem Kindelement `c` ein Kindelement `e` angegeben wurde.

```
<xsd:complexType name="exampleType">
  <xsd:sequence>
    <xsd:element name="a" type="xsd:string" />
    <xsd:element name="b" type="xsd:long" />
    <xsd:choice>
      <xsd:element name="c" type="xsd:boolean" />
      <xsd:element name="e" type="xsd:long" />
      <xsd:element name="f" type="xsd:boolean" />
    </xsd:choice>
  </xsd:sequence>
</xsd:complexType>
```

Abbildung 33: Beispiel für `xsd:choice` in XML-Schema

```
<?xml version="1.0" encoding="UTF-8" ?>
<my:d xmlns:my="http://example.org/Sample">
  <a>QWERT</a>
  <b>1</b>
  <e>1000</e>
</my:d>
```

Abbildung 34: Beispiel für Instanzdokument zum Schemadokument in Abbildung 33

4.2.6.2 Partikel

Die am häufigsten verwendeten Partikel sind neben Kompositoren Deklarationen von XML-Elementen, wobei auch auf globale Deklarationen verwiesen werden kann. In den Beispielen in Abbildung 31 und Abbildung 33 wurden XML-Elemente lokal innerhalb des komplexen Typs deklariert. Wie in Kapitel 4.2.3 beschrieben, können Elemente aber auch global deklariert werden. Auf solche Deklarationen wird dann mit dem ebenfalls für lokale Deklarationen von XML-Elementen verwendeten Partikel `xsd:element` verwiesen. Statt Name und Typ wird dann im Attribut `xsd:element/@ref` der qualifizierte Name des global deklarieren XML-Elementes angegeben. Als Beispiel zeigt Abbildung 35 den komplexen Typ aus Abbildung 31, wobei die lokalen Deklarationen der Elemente durch globale ersetzt sind. Zu beachten ist, dass dadurch die Namen der Elemente `a`, `b` und `c` Teil des Namenraumes werden und im Instanzdokument ihre qualifizierten Namen angegeben werden müssen, wie in 4.2.3 ausgeführt.

```
<xsd:complexType name="exampleType">
  <xsd:sequence>
    <xsd:element ref="my:a" />
    <xsd:element ref="my:b" />
    <xsd:element ref="my:c" />
  </xsd:sequence>
</xsd:complexType>
```

Abbildung 35: Beispiel für Referenz auf globale Deklaration eines Elementes in XML-Schema

Mit dem Partikel `xsd:any` kann angegeben werden, dass ein beliebiges Kindelement im Instanzdokument zugelassen wird. Das kann verwendet werden, um gezielt an bestimmten Stellen in Instanzdokumenten Erweiterungen zuzulassen, was auch mit dem Typ `xsd:anyType` möglich wäre. Mit dem Attribut `xsd:any/@namespace` können dabei die Namensräume eingeschränkt werden, in denen sich der Name des Kindelementes befinden muss. Verwendet wird `xsd:any` z. B. im Schemadokument zum SOAP-Standard [W3C 02b], um in SOAP-Nachrichten innerhalb der Elemente `env:Header` und `env:Body` beliebige Kindelemente zuzulassen.

Um Modelgroups für unterschiedliche Typdefinitionen wiederverwenden zu können, können diese explizit als benannte Modelgroup definiert und dann mit Partikeln referenziert werden. Eine benannte Modelgroup wird als Kindelement von `xsd:schema` mit dem Element `xsd:group` definiert und ihr Name im Attribut `xsd:group/@name` angegeben. Das Element `xsd:group` hat als Kindelement den Kompositor, der Modelgroup bildet. Benannte Modelgroups können mit Partikeln referenziert werden. Als Partikel dient ebenfalls das Element `xsd:group`, bei dem jedoch im Attribut `xsd:group/@ref` der qualifizierte Name der benannten Modelgroup angegeben wird.

4.2.6.3 Attribute `minOccurs` und `maxOccurs`

Für ein Partikel kann angegeben werden, dass es im Instanzdokument mehrfach vorkommen darf oder optional ist. Hierzu kann ihm im Attribut `minOccurs` zugefügt werden, wie oft das Partikel mindestens vorkommen muss und im Attribut `maxOccurs` wie oft maximal. Für `maxOccurs` bedeutet dabei der Wert `unbounded`, dass das Partikel beliebig oft vorkommen darf. Wird das Attribut `minOccurs` nicht angegeben wird der Wert 1 angenommen, bei `maxOccurs` der Wert von `minOccurs` oder 1, je nachdem welcher Wert größer ist. Entfallen beide Attribute, muss also das Partikel im Instanzdokument genau einmal vorkommen, wie es bisher in diesem Kapitel angenommen wurde.

Abbildung 36 zeigt den komplexen Typ aus Abbildung 31, erweitert um Attribute `minOccurs` und `maxOccurs`. Beim Element `a` sind beide Attribute nicht angegeben, so dass es genau einmal vorkommen muss. Das Attribut `b` muss dagegen mindestens einmal und darf beliebig oft vorkommen. Das Element `c` ist optional, darf also einmal vorkommen oder fehlen.

```
<xsd:complexType name="exampleType">
  <xsd:sequence maxOccurs="2">
    <xsd:element name="a" type="xsd:string" />
    <xsd:element name="b" type="xsd:long"
                 minOccurs="1" maxOccurs="unbounded" />
    <xsd:element name="c" type="xsd:boolean"
                 minOccurs="0" />
  </xsd:sequence>
</xsd:complexType>
```

Abbildung 36: Beispiel für Attribut `minOccurs` und `maxOccurs` in XML-Schema

Zu beachten ist, dass auch Kompositoren Partikel sind und ihnen daher ebenfalls die Attribute `minOccurs` und `maxOccurs` zugefügt werden dürfen. Damit kann festgelegt werden, dass das durch den Kompositor beschriebene, im Instanzdokument mehrfach vorkommen kann oder optional sein darf. Ist der Kompositor optional, können alle seine Partikel im Instanzdokument zusammen fehlen. Kommt der Kompositor mehrfach vor, werden seine Partikel zusammen wiederholt. Der Kompositor `xsd:sequence` in Abbildung 36 darf ein- oder zweimal vorkommen. Damit ist das Instanzdokument in Abbildung 32 weiterhin gültig, ebenso wie das Instanzdokument in Abbildung 37.

```
<?xml version="1.0" encoding="UTF-8" ?>
<my:d xmlns:my="http://example.org/Sample">
  <a>ABCDE</a>
  <b>42</b>
  <b>7</b>
  <b>-10</b>
  <a>QWERT</a>
  <b>15</b>
  <c>true</c>
</my:d>
```

Abbildung 37: Beispiel für Instanzdokument zum Schemadokument in Abbildung 36

4.2.6.4 Zuordnung von Attributen zu komplexen Typen

Welche Attribute ein XML-Element eines komplexen Typs haben darf oder muss, wird in der Definition des komplexen Typs mit weiteren Kindelementen festgelegt. Hierzu werden Elemente `xsd:complexType/xsd:attribute` verwendet, mit denen ein Attribut lokal deklariert oder auf ein global deklariertes Attribut verwiesen werden kann.

Wie bei Deklarationen von Elementen wird für eine lokale Deklaration eines Attributes dessen Name und Typ angegeben. Zum Verweisen auf ein global deklariertes Attribut wird in `xsd:attribute/@ref` dessen qualifizierter Name angegeben.

Für ein Attribut kann festgelegt werden, ob es optional ist und welcher Wert im Falle des Fehlens angenommen werden soll. Ob ein Attribut in Instanzdokument optional ist, wird im Schemadokument mit dem Attribut `xsd:attribute/@use` festgelegt. Wird der Wert `required` angegeben, muss es vorhanden sein, bei `optional` darf es auch fehlen. Letzteres ist auch der Fall, wenn das Attribut `xsd:attribute/@use` nicht angegeben wurde. Mit dem Attribut `xsd:attribute/@default` kann angegeben werden, welcher Wert bei einem fehlenden optionalen Attribut angenommen werden soll.

Abbildung 38 zeigt erneut eine Erweiterung des komplexen Typs aus Abbildung 31, hier ist er um Attribute erweitert. Elemente des komplexen Typs müssen das Attribut `x` haben. Das Attribut `y` ist dagegen optional. Fehlt es, wird der Wert 0 angenommen. Auch das Attribut `my:z` ist optional. Anders als bei den lokal deklarieren Attributen `x` und `y`, wird für dieses Attribut auf eine globale Deklaration verwiesen. Daher befindet sich sein Name im durch das Schemadokument definierten Namensraum, so dass im Instanzdokument ein Präfix angegeben werden muss, wie in Abbildung 39 gezeigt.

```
<xsd:complexType name="exampleType">
  <xsd:sequence>
    <xsd:element name="a" type="xsd:string" />
    <xsd:element name="b" type="xsd:long" />
    <xsd:element name="c" type="xsd:boolean" />
  </xsd:sequence>
  <xsd:attribute name="x" type="xsd:long" use="required" />
  <xsd:attribute name="y" type="xsd:long" default="0" />
  <xsd:attribute ref="my:z" />
</xsd:complexType>
```

Abbildung 38: Beispiel für komplexen Typ mit Attributen in XML-Schema

```
<?xml version="1.0" encoding="UTF-8" ?>
<my:d xmlns:my="http://example.org/Sample"
      x="100"
      my:z="50">
  <a>ABCDE</a>
  <b>42</b>
  <c>false</c>
</my:d>
```

Abbildung 39: Beispiel für Instanzdokument zum Schemadokument in Abbildung 38

Für einen komplexen Typ kann festgelegt werden, dass neben den deklarierten Attributen auch nicht deklarierte erlaubt sind. Angezeigt wird das mit dem Kindelement `xsd:complexType/xsd:anyAttribute`. Wie beim Element `xsd:any` kann mit dem Attribut `xsd:anyAttribute/@namespace` eingeschränkt werden, aus welchen Namensräumen die Attribute stammen müssen.

4.2.7 Ableiten komplexer Typen

Komplexe Typen können auch aus bereits vorhandenen Typen abgeleitet werden. Wie bei einfachen Typen ist es dabei mit der Restriction möglich, einen neuen komplexen Typen zu definieren, bei dem die Wertemenge eines vorhandenen eingeschränkt wird. Zusätzlich gibt es bei komplexen Typen die Extension, mit der einem vorhandenen Ty-

pen zusätzliche Attribute oder Kindelemente zugefügt werden können. Die Extension ist auch mit vorhandenen einfachen Typen möglich, was erlaubt Elementen solcher Typen Attribute zuzufügen.

Wie bei der Komposition aus XML-Elementen und Attributen werden komplexe Typen auch beim Ableiten mit dem Element `xsd:complexType` definiert. Mit dessen Kindelementen `xsd:complexContent` bzw. `xsd:simpleContent` wird angezeigt, ob von einem komplexen oder einfachen Typ abgeleitet werden soll. Diese Elemente haben wiederum die Kindelemente `xsd:restriction` im Falle der Restriction oder `xsd:extension` für die Extension.

Bei der Restriction muss mit Kindelementen von `xsd:restriction` der komplexe Typ erneut definiert werden, so dass er eine gegenüber dem Basistyp eingeschränkte Wertemenge hat. Um die Wertemenge einzuschränken, können z. B. in der Definition des Basistyps verwendete einfache Typen durch weiter eingeschränkte ersetzt werden. Auch können Werte des Attributes `minOccurs` vergrößert oder Werte des Attributes `maxOccurs` verkleinert werden. In jedem Fall bleiben XML-Elemente des abgeleiteten Typs auch gültige XML-Elemente des Basistyps.

Das ist bei der Extension nicht der Fall, weil zusätzliche Kindelemente und Attribute zugefügt werden können. Die Extension hat Ähnlichkeiten mit der Vererbung von Klassen in objektorientierten Modellen, bei denen auch in Unterklassen zusätzliche Eigenschaften zugefügt werden können. Das hierzu verwendeten Element `xsd:extension` kann als Kindelemente einen Kompositor sowie Deklarationen von Attributen oder Verweise auf solche haben. Dadurch deklarierte XML-Elemente dürfen im Instanzdokument zusätzlich als Kindelemente vorhanden sein, ebenso deklarierte Attribute.

Die Extension erlaubt, komplexe Typen aus einfachen abzuleiten, so dass ihnen Attribute zugefügt werden können. Ein XML-Element, das einen einfachen Typ hat, hat keine Kindelemente, sondern enthält statt dessen die Werte des einfachen Typs als Zeichenketten. Es hat aber keine Attribute. Um XML-Elemente zu deklarieren, die Werte eines einfachen Typs enthalten und trotzdem Attribute haben dürfen, müssen mit der Extension geeignete komplexe Typen abgeleitet werden.

Abbildung 40 zeigt ein Beispiel für einen solchen Typ. Er erweitert den numerischen Typ `xsd:decimal` um ein Attribut mit Namen `unit`. Der Typ könnte z. B. zur Speicherung einer physikalischen, skalaren Größe verwendet werden, deren Einheit im Attribut `unit` angegeben wird. Abbildung 41 zeigt ein Beispiel für ein XML-Element dieses Typs. Es enthält den Wert 10. Im Attribut `unit` ist „`m`“ für Meter angegeben.

```
<xsd:complexType name="ValueWithUnit">
  <xsd:simpleContent>
    <xsd:extension base="xsd:decimal">
      <xsd:attribute name="unit" type="xsd:string" />
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>
```

Abbildung 40: Beispiel für komplexen, aus einem einfachen abgeleiteten Typ in XML-Schema

```
<my:d unit="m">10</my:d>
```

Abbildung 41: Beispiel für Instanzdokument zum Schemadokument in Abbildung 40

4.3 Web Services Description Language (WSDL)

Die Web Services Description Language (WSDL) erlaubt, Webservices und ihre Schnittstellen zu beschreiben. XML-Schema hat dabei eine wesentliche Rolle zur Beschreibung von Typen. WSDL-Beschreibungen befinden sich in WSDL-Dokumenten, die vom Webservice verbreitet werden. Für Entwickler bilden sie die Grundlage für die Implementierung von Webclients.

Diese Arbeit betrachtet WSDL in der Version 1.2 (WSDL 1.2), gemäß des zweiteiligen Arbeitsentwurfes ihres Standards vom Juli 2002. WSDL erlaubt auch die Beschreibung von Webservices, auf die statt mit SOAP mit einem anderen Protokoll zugegriffen wird. Der erste Teil[19] des WSDL-Standards [Chinnici 02] beschreibt daher die Sprachelemente von WSDL, mit denen ein Webservice unabhängig vom verwendeten Protokoll und Nachrichtenformat beschrieben wird. Der zweite Teil [Moreau 02] fügt Sprachelemente hinzu, mit denen unter anderem protokoll- und nachrichtenformatspezifisches für SOAP definiert wird. Später soll noch ein dritter Teil ergänzt werden, der als verständliches Tutorial dienen soll. Von diesem wurde bisher jedoch nicht einmal ein Arbeitsentwurf veröffentlicht. Ein Tutorial für WSDL enthält aber [RefsnesData 02].

Bei [Chinnici 02] und [Moreau 02] handelt es sich um den ersten Arbeitsentwurf des Standards von WSDL 1.2. Daher ist zu erwarten, dass sich beide noch inhaltlich ändern werden. Insbesondere der zweite Teil ist noch weitgehend mit dem entsprechenden Teil der W3C-Note von WSDL 1.1 [Christensen 01] identisch. Damit ist es z. B. noch nicht möglich, die Verwendung von SOAP 1.2 zu beschreiben, sondern nur die von SOAP 1.1. Trotzdem dient dieser Arbeitsentwurf als Basis für diese Arbeit. In Kapitel 4.3.7 der nachfolgenden Beschreibung von WSDL wird daher die Beschreibung von SOAP 1.1 auf die von SOAP 1.2 übertragen.

4.3.1 Grundbegriffe

WSDL-Dokumente sind XML-Dokumente der Sprache WSDL, mit denen Webservices beschrieben werden. Das umfasst die Beschreibung ihre Schnittstellen, welche Schnittstellen ein Webservice hat und über welche Netzwerkadressen der Webservice erreichbar ist. Schnittstellen enthalten Operationen, für die beschrieben wird, welche Typen für einen Operationsaufruf ausgetauschte Nachrichten haben. Sie werden zunächst unabhängig von Protokoll beschrieben und dann um protokollspezifisches ergänzt.

Bereits in Kapitel 2.5 wurde festgestellt, was für einen Webservice beschrieben werden muss, der mit seinen Webclients über SOAP kommuniziert. Es müssen Nachrichtentypen von ausgetauschten Nachrichten definiert werden. Wegen der Verwendung von SOAP sind dazu nur die anwendungsspezifischen Teile der SOAP-Nachricht zu beschreiben. Es muss festgelegt werden, Nachrichten welcher Typen zusammen Request-/Response-Paare bilden, wobei Fehlernachrichten berücksichtigt werden müssen. Solche Paare müssen zu Schnittstellen zusammengefasst werden. Dabei wird zwischen abstrakten, also protokollunabhängigen Schnittstellen und gebundenen unterschieden, für die Protokoll und Nachrichtenformat festliegt. Für einen Webservice muss angegeben werden, welche Schnittstellen er hat und über welche Netzwerkadressen über sie Nachrichten ausgetauscht werden können.

---

[19] In [Chinnici 02] und [Moreau 02] werden die beiden Teile nicht zwei Teile eines Standards bezeichnet, sondern als unabhängig Standards, von denen [Moreau 02] [Chinnici 02] ergänzt. Für die Klarheit der Darstellung wird hier trotzdem von Teil 1 und Teil 2 gesprochen.

WSDL erlaubt Webservices in diesem Sinne zu beschreiben, wozu in [Chinnici 02] einerseits ein abstraktes Modell von WSDL vorgestellt wird, sowie eine XML-Darstellung für WSDL-Dokumente, die in [Moreau 02] ergänzt wird. Im abstrakten Modell wird jeder Teil der Beschreibung des Webservices durch eine Beschreibungskomponente ausgedrückt. Beschreibungskomponenten haben Eigenschaften, die Werte enthalten. Qualifizierte Namen werden als Werte verwendet, um auf andere Beschreibungskomponenten zu verweisen.

Diese Arbeit beschreibt WSDL direkt im Sinne der XML-Darstellung und verzichtet auf die detaillierte Darstellung des abstrakten Modells. Jede Beschreibungskomponente wird darin als ein XML-Element dargestellt und um Attribute und Kindelemente ergänzt. Ein XML-Dokument das die Beschreibungskomponenten enthält und als WSDL-Dokument bezeichnet wird, dient zur Beschreibung eines Webservices.

Abbildung 42 zeigt grob die Beschreibungskomponenten und weitere Elemente, mit denen ein Webservice in WSDL beschrieben wird. Zunächst werden XML-Elemente und Typen mit Beschreibungssprachen wie XML-Schema beschrieben. Diese werden in Beschreibungskomponenten für Nachrichtentypen von WSDL verwendet, um ausgetauschte Nachrichten zu beschreiben. Typen der Nachrichten, die zusammen einen Operationsaufruf bilden, werden zu einer Operation zusammengefasst. Eine Beschreibungskomponente für abstrakte Schnittstellen enthält die Definitionen der Operationen, die zu ihr gehören. Eine Menge von abstrakten Schnittstellen wird zu einem Webservicetyp zusammengefasst, der in Abbildung 42 nicht mit dargestellt wurde. Gebundene Schnittstellen erweitern abstrakte um protokoll- und nachrichtenformatspezifisches. Schließlich gibt es eine Beschreibungskomponente in der ein einzelner Webservice beschrieben wird. Sie enthält Beschreibungen von Endpunkten. Jeder Endpunkt beschreibt eine vom Webservice unterstützte gebundene Schnittstelle und ihre Netzwerkadresse. Außerdem wird für einen Webservice sein Webservicetyp angegeben.

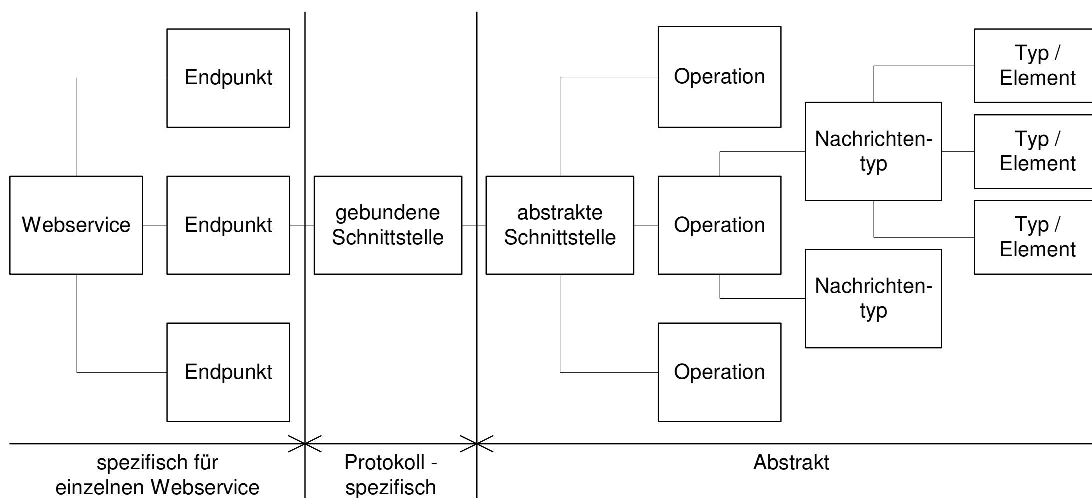

Abbildung 42: Struktur von WSDL-Beschreibungen

4.3.2 WSDL-Dokumente

Die Beschreibungskomponenten sind kodiert als XML-Elemente in WSDL-Dokumenten enthalten. Sie sind Kindelemente des Rootelementes `wsdl:definitions`[20]. Das Attribut `wsdl:definitions/@targetNamespace` enthält die URI des Namensraumes, in dem sich die Namen von Beschreibungskomponenten befinden. Andere Namensräu-

[20] Das Präfix `wsdl` stehe für den Namensraum `http://www.w3.org/2002/07/wsdl`, der zu WSDL gehört.

me können in das WSDL-Dokument importiert werden, so dass enthaltene qualifizierte Namen verwendet werden dürfen.

Der grundsätzliche Aufbau eines WSDL-Dokumentes wird in Abbildung 43 veranschaulicht. Es zeigt das Rootelement `wsdl:definitions` sowie die Beschreibungskomponenten als Kindelemente. Ausgelassene Teile sind durch drei Punkte („...") ersetzt. Mit den ersten, optionalen Kindelementen `wsdl:definitions/wsdl:import` werden Namensräume importiert. Ihnen folgt das ebenfalls optionale Kindelement `wsdl:types`, in das Beschreibungen von Elementen und Typen, formuliert in Beschreibungssprachen wie XML-Schema, eingebettet werden. Danach darf in beliebiger Reihenfolge eine beliebige Zahl von Beschreibungskomponenten `wsdl:message`, `wsdl:portType`, `wsdl:binding`, `wsdl:serviceType` und `wsdl:service` folgen, mit denen Nachrichten, abstrakte und gebunden Schnittstellen und einzelne Webservices beschrieben werden.

```
<?xml version="1.0" encoding="UTF-8" ?>
<wsdl:definitions
    targetNamespace="http://example.org/NewsService020917/Interface"
    xmlns:nwsi="http://example.org/NewsService020917/Interface"
    xmlns:wsdl="http://www.w3.org/2002/07/wsdl">
  <wsdl:import namespace="..." location="..." />

  <wsdl:types> ... </wsdl:types>

  <wsdl:message name="..." >
    <wsdl:part name="..." ... />
  </wsdl:message>

  <wsdl:portType name="...">
    <wsdl:operation name="..."> ... </wsdl:operation>
    ...
  </wsdl:portType>

  <wsdl:binding name="..." type="..." >
    <wsdl:operation name="..." > ... </wsdl:operation>
    ...
  </wsdl:binding>

  <wsdl:serviceType name="...">
    <wsdl:portType name="..." />
    ...
  </wsdl:serviceType>

  <wsdl:service name="..." serviceType="...">
    <wsdl:port name="..." binding="..."> ... </wsdl:port>
    ...
  </wsdl:service>
</wsdl:definitions>
```

Abbildung 43: Grundsätzlicher Aufbau von WSDL-Dokumenten

Abbildung 43 zeigt das Attribut `wsdl:definitions/@targetNamespace`, das die URI des Namensraumes enthält, in dem sich die Namen der Beschreibungskomponenten befinden. Die lokalen Namen sind in den Beschreibungskomponenten (außer in `wsdl:types`) im Attribut `name` enthalten. Ähnlich wie bei XML-Schema werden ihre qualifizierten Namen verwendet, um auf sie von anderen Beschreibungskomponenten zu verweisen.

Das Element `wsdl:import` erlaubt, Namensräume zu importieren, die außerhalb des WSDL-Dokumentes definiert wurden, so dass dort enthaltene qualifizierte Namen verwendet werden dürfen. Beschreibungen von Webservices können so in mehrere WSDL-Dokumente aufgeteilt und Teile wiederverwendet werden. Z. B. könnte eine gebundene Schnittstelle in einem eigenen WSDL-Dokument definiert werden. Alle Webservices, die sie unterstützen, werden in anderen WSDL-Dokumenten beschrieben. Die Definition der gebundenen Schnittstelle wird in diese WSDL-Dokumente importiert und kann so wiederverwendet werden.

Das Element `wsdl:import` ist dem Element `xsd:import` von XML-Schema ähnlich. Im Attribut `namespace` wird die URI des zu importierenden Namensraumes angegeben. Das Attribut `location` enthält eine URI, die den Ort beschreibt, an dem sich eine Beschreibung des Namensraumes befindet. Anders als bei XML-Schema ist das Attribut `location` jedoch nicht optional.

Um WSDL-Dokumente für Menschen verständlicher zu machen, erlaubt es WSDL ihnen Kommentare zuzufügen. Hierzu ist in jedem Element von WSDL als erstes Kindelement ein Element `wsdl:documentation` erlaubt, das den Kommentar enthält.

4.3.3 Typen

Für die Definition von Nachrichten benötigte Typen und XML-Elemente werden mit Typsystemen definiert, die nicht Teil des WSDL-Standards sind. Solche Definitionen sind im Element `wsdl:definitions/wsdl:types` als Kindelemente enthalten. Empfohlen wird als Typsystem XML-Schema. Für dieses wird das Rootelement eines Schemadokumentes `xsd:schema` zum Kindelement von `wsdl:types`.

Das Element `wsdl:types` kann beliebig viele Kindelemente haben, in denen Typen und XML-Elemente beschrieben werden. Selbst darf das Element `wsdl:types` nur einmal im WSDL-Dokument enthalten sein oder fehlen. Es gibt keine Beschränkungen, welche Typsysteme verwendet werden dürfen. Daher ist nicht festgelegt, welches die Kindelemente von `wsdl:types` sind. Im Teil 1 des Arbeitsentwurfes des WSDL-Standards [Chinnici 02] wird jedoch festgestellt, dass die Verwendung von XML-Schema zu maximaler Interoperabilität führt. Daher wird dort XML-Schema als Typsystem für WSDL empfohlen.

```
<wsdl:types>
  <xsd:schema
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:nwst="http://example.org/NewsService020917/Types"
      targetNamespace="http://example.org/NewsService020917/Types" >
    ...
    <xsd:element name="Name"     type="nwst:restrictedString" />
    <xsd:element name="Password" type="nwst:password" />
    ...
    <xsd:complexType name="loginRequest">
      <xsd:sequence>
        <xsd:element ref="nwst:Name" />
        <xsd:element ref="nwst:Password" minOccurs="0"/>
      </xsd:sequence>
    </xsd:complexType>

    <xsd:element name="Login" type="nwst:loginRequest" />
    ...
  </xsd:schema>
</wsdl:types>
```

Abbildung 44: Beispiel für in WSDL-Dokument eingebettetes XML-Schemadokument

Wird XML-Schema als Typsystem verwendet, hat das Element `wsdl:types` Kindelemente `xsd:schema`, in denen Namensräume definiert werden, wie in Kapitel 4.2 beschrieben. Das Element `xsd:schema` hat weiterhin das Attribut `targetNamespace`, in dem die URI des durch das Schemadokument definierten Namensraumes angegeben wird. Im Schemadokument definierte Typen und XML-Elemente befinden sich damit in einem anderen Namenraum als die Beschreibungskomponenten des WSDL-Dokumentes.

Abbildung 44 zeigt ein Beispiel für ein Element `wsdl:types`, in das ein Schemadokument eingebettet ist. Ebenso wie die nachfolgenden Beispiele ist es ein Ausschnitt des WSDL-Dokumentes der Webkomponente für eine Internetzeitung, die in Kapitel 2.6 vorgestellt wurde. Ausgelassene Teile wurden durch drei Punkte („...") ersetzt.

4.3.4 Nachrichten

Nachrichtentypen werden mit Elementen `wsdl:definitions/wsdl:message` definiert. Nachrichten können aus mehreren Teilen (Parts) bestehen, die einzeln typisiert sind. Jedem Part wird ein Typ oder eine Deklaration eines XML-Elementes des verwendeten Typsystems zugeordnet. Obwohl damit Parts XML-Fragmente sind, handelt es sich um abstrakte Nachrichten, die erst in durch die Definition der gebundenen Schnittstelle auf das konkrete Nachrichtenformat ausgetauschter Nachrichten abgebildet werden.

Jeder Part des abstrakten Nachrichtentyps wird beschrieben durch ein Kindelement `wsdl:message/wsdl:part`. Das Element `wsdl:part` hat ein Attribut `name`, das einen Namen enthält, der den Part eindeutig in der Nachrichtendeklaration bezeichnet. Solche Namen werden nicht Teil des durch das WSDL-Dokument definierten Namensraumes. Um einen Part eindeutig zu identifizieren, muss daher neben seinem Namen auch der qualifizierte Name des Nachrichtentyps angegeben werden, dessen lokaler Teil im Attribut `wsdl:message/@name` angegeben wird.

Mit einem weiteren Attribut muss für einen Part festgelegt werden, welchen Typ er hat. Nur für die Verwendung von XML-Schema als Typsystem sind solche Attribute im WSDL-Standard definiert. Für andere Typsysteme müssen zusätzlich zum WSDL-Standard neue Attribute definiert werden. Wird XML-Schema als Typsystem verwendet, kann ein einfacher oder komplexer Typ einem Part zugeordnet werden, indem im Attribut `wsdl:part/@type` der qualifizierte Name des Typs angegeben wird. Alternativ kann im Attribut `wsdl:part/@element` der qualifizierte Name eines XML-Elementes angegeben werden. Dann ist es die Deklaration des XML-Elementes, die als Typ des Parts fungiert.

Abbildung 45 zeigt zwei Beispiele für Definitionen von Nachrichtentypen. Beide haben nur einen Part dessen Typ eine Deklaration eines XML-Elementes ist, die im Schemadokument in Abbildung 44 enthalten ist. Im Attribut `wsdl:part/@name` wird sein Name angegeben.

```
<wsdl:message name="LoginRequest">
  <wsdl:part name="FirstBodyEntry" element="nwst:Login" />
</wsdl:message>

<wsdl:message name="FaultDetailsMessageIDs">
  <wsdl:part
      name="FaultDetailsMessageIDs"
      element="nwst:MessageIDsSequence" />
</wsdl:message>
```

Abbildung 45: Zwei Beispiele für Deklarationen von Nachrichten in WSDL

Definierte Nachrichtentypen beschreiben abstrakte Nachrichten. Auch wenn Typen von Parts mit XML-Schema beschrieben werden, müssen zwischen Webkomponenten ausgetauschte Nachrichten die Parts nicht als XML-Fragmente enthalten. Nur im abstrakten Modell der Nachrichten sind es XML-Fragmente. Erst durch die Definition der gebundenen Schnittstelle wird festgelegt, wie das abstrakte Modell auf konkrete Nachrichtenformate ausgetauschter Nachrichten abgebildet wird. Das konkrete Nachrichtenformat kann sich stark vom abstrakten unterscheiden oder dieses direkt übernehmen. Für diese Arbeit wird letzteres nachfolgend in Kapitel 5.3 festgeschrieben und die XML-Fragmente der Parts direkt auf Body-, Header- und Detaileinträge in SOAP-Nachrichten abgebildet.

4.3.5 Abstrakte Schnittstellen

Abstrakte Schnittstellen werden als Mengen von Operationen definiert, die ihrerseits mit Hilfe von Nachrichtentypen definiert werden. Für jede Operation wird beschrieben, wie viele Nachrichten welcher Typen für einen Operationsaufruf vom Webservice empfangen oder gesendet werden, wobei vier Kommunikationsmuster unterstützt werden. Es kann auch festgelegt werden, welche Typen die Nachrichten haben, mit denen in Fehlerfällen geantwortet wird.

In WSDL-Dokumenten werden abstrakte Schnittstellen als Elemente `wsdl:definitions/wsdl:portType` definiert. Im WSDL-Standard werden sie daher als „Port Types“ bezeichnet. Für jede ihrer Operationen gibt es ein Kindelement `wsdl:portType/wsdl:operation`, das im Attribut `name` den Namen der Operation enthält.

Für jede Operation kann eines von vier Kommunikationsmustern festgelegt werden, die bestimmen wie viele Nachrichten in welcher Richtung für einen Operationsaufruf ausgetauscht werden. Die vier Kommunikationsmuster ergeben sich aus der Festlegung, dass für jeden Operationsaufruf mindestens eine Nachricht ausgetauscht werden muss, aber nicht mehr als eine Nachricht pro Richtung ausgetauscht werden darf. Der erste Teil des WSDL-Standards [Chinnici 02] unterscheidet daher vier Arten von Operationen mit unterschiedlichen Kommunikationsmustern:

- Input-Output-Operationen: Zunächst empfängt der Webservice eine Nachricht, danach antwortet er mit einer zweiten.
- Input-Only-Operationen: Der Webservice empfängt eine Nachricht.
- Output-Input-Operationen: Zunächst sendet der Webservice eine Nachricht, der Webclient antwortet danach mit einer zweiten.
- Output-Only-Operationen: Der Webservice sendet eine Nachricht.

Interessant ist die Beobachtung, dass beide im SOAP-Standard vorgestellten und in Kapitel 3.8 wiedergegebenen Kommunikationsmuster, Input-Output-Operationen realisieren. Sowohl beim Request-/Response- als auch beim SOAP-Response-Kommunikationsmuster von SOAP wird zunächst ein Request vom Webclient zum Webservice gesendet, der danach mit einem Response antwortet. Die unterschiedlichen Nachrichtenformate für den Request, in denen sich beide Kommunikationsmuster unterscheiden, haben keine Auswirkung auf die abstrakte Schnittstelle. Über diese beiden Kommunikationsmuster hinaus kann der Austausch einer einzelnen SOAP-Nachricht in WSDL mit Input-Only- oder Output-Only-Operationen beschrieben werden.

Welches der Kommunikationsmuster für einen Operationsaufruf verwendet wird und welche Typen für ihn ausgetauschte Nachrichten haben, wird mit Kindelementen von

wsdl:operation beschrieben. Ein Kindelement wsdl:input beschreibt eine vom Webservice empfangene Nachricht, ein Kindelement wsdl:output eine ausgehende. Jedes dieser beiden Kindelemente darf höchstens einmal vorkommen. Ihre Reihenfolge bestimmt die Reihenfolge der ausgetauschten Nachrichten.

Werden für einen Operationsaufruf zwei Nachrichten ausgetauscht, können zusätzlich Fehlernachrichten deklariert werden. Fehlernachrichten können in Fehlersituationen alternativ zur zweiten Nachricht gesendet bzw. empfangen werden. Deklariert werden sie mit Kindelementen wsdl:operation/wsdl:fault. Von ihnen sind mehrere in der gleichen Operation erlaubt, um unterschiedliche Fehlernachrichten beschreiben zu können. Damit sie unterschieden werden können, müssen Deklarationen von Fehlernachrichten benannt werden. Dazu wird ihr Name in einem Attribut wsdl:fault/@name angegeben.

Um festzulegen, welchen Typ eine ausgetauschte Nachricht hat, wird auf Nachrichtentypen verwiesen. In jedem der Elemente wsdl:input, wsdl:output und wsdl:fault gibt es hierzu ein Attribut message, dass den qualifizierten Namen des Nachrichtentypen enthält.

Ein Beispiel für eine Definition einer abstrakten Schnittstelle wird in Abbildung 46 gegeben. Es ist ein Ausschnitt der abstrakten Schnittstelle der Webkomponente für eine Internetzeitung. Gezeigt werden zwei Operationen, weitere sind mit drei Punkten („...“) angedeutet. Beide Operationen sind Input-Output-Operationen und deklarieren Fehlernachrichten. Die Operation Login deklariert die Fehlernachricht AccessDenied, die Operation Get die drei Fehlernachrichten InvalidSessionID, InvalidMessageID und MessageLocked. In Attributen mit Namen message wird auf die Typen der Nachrichten verwiesen. In Abbildung 45 wurden bereits die Definitionen der Nachrichtentypen nwsi:LoginRequest und nwsi:FaultDetailsMessageIDs gezeigt. Das Präfix nwsi steht hier für den Namensraum den das WSDL-Dokument definiert.

```
<wsdl:portType name="NewsServicePortType">
  <wsdl:operation name="Login">
    <wsdl:input  message="nwsi:LoginRequest" />
    <wsdl:output message="nwsi:LoginResponse" />
    <wsdl:fault
        name="AccessDenied"
        message="nwsi:FaultDetailsEmpty" />
  </wsdl:operation>

  <wsdl:operation name="Get">
    <wsdl:input  message="nwsi:GetRequest" />
    <wsdl:output message="nwsi:GetResponse" />
    <wsdl:fault
        name="InvalidSessionID"
        message="nwsi:FaultDetailsEmpty" />
    <wsdl:fault
        name="InvalidMessageID"
        message="nwsi:FaultDetailsMessageIDs" />
    <wsdl:fault
        name="MessageLocked"
        message="nwsi:FaultDetailsMessageIDs" />
  </wsdl:operation>
  ...
</wsdl:portType>
```

Abbildung 46: Ausschnitt von Definition einer abstrakten Schnittstelle in WSDL

4.3.6 Gebundene Schnittstellen

Eine gebundene Schnittstelle legt für eine abstrakte fest, mit welchem Protokoll Nachrichten welcher Nachrichtenformate ausgetauscht werden. Der Arbeitsentwurfs des WSDL-Standards stellt dazu protokollunabhängige Elemente zur Verfügung, mit denen wie in den abstrakten Schnittstellen Operationen und für Operationsaufrufe ausgetauschte Nachrichten aufgelistet werden. Diesen Elementen werden protokollspezifisch definierte Kindelemente zugefügt, die das Protokoll und Nachrichtenformat identifizieren und ihre Details festlegen. Solche Kindelemente werden für SOAP im zweiten Teil des WSDL-Standards definiert. Für andere Protokolle muss das in anderen Spezifikationen geschehen.

Durch eine gebundene Schnittstelle, die im WSDL-Standard als „Binding“ bezeichnet wird, wird eine abstrakte Schnittstelle an ein Protokoll gebunden. So werden die Details festgelegt, wie Nachrichten über die Schnittstelle ausgetauscht werden, wozu auch das Nachrichtenformat gehört. Soll eine abstrakte Schnittstelle an mehrere Protokolle gebunden werden, muss für jedes Protokoll eine gebundene Schnittstelle definiert werden.

Unabhängig von einem bestimmten Protokoll wird im Teil 1 des Arbeitsentwurfes des WSDL-Standards [Chinnici 02] festgelegt, mit welchen Elementen eine gebundene Schnittstelle definiert wird. Diesen werden Kindelemente zugefügt, die protokollspezifisch definiert werden, z. B. für SOAP 1.2. Solche Erweiterungen müssen anzeigen, um welches Protokoll es sich handelt. Außerdem enthalten sie alle protokollspezifischen Informationen zur Beschreibung der Schnittstelle. Das schließt auch Informationen über das protokollspezifische Nachrichtenformat mit ein.

Die Struktur, mit der gebundene Schnittstellen in WSDL-Dokumenten definiert werden, ist der von abstrakten Schnittstellen ähnlich. Eine gebundene Schnittstelle wird mit einem Element `wsdl:definitions/wsdl:binding` definiert. Sein Attribut `type` enthält den qualifizierten Namen der abstrakten Schnittstelle für die das Protokoll festgelegt werden soll. Für jede Operation der abstrakten Schnittstelle gibt es in der gebundenen ein Kindelement `wsdl:binding/wsdl:operation`. In dessen Attribut `name` wird wie in der abstrakten Schnittstelle der Namen der Operation angegeben. Kindelemente von `wsdl:operation` sind auch hier `wsdl:input`, `wsdl:output` und `wsdl:fault`, mit denen wieder die vom Webservice empfangenen oder gesendeten Nachrichten bzw. Fehlernachrichten beschrieben werden. Ihnen werden protokollspezifische Kindelemente zugefügt, um ausgetauschte Nachrichten genauer zu beschreiben. Um Operationen oder die gebundenen Schnittstelle als ganzes genauer zu beschreiben, werden protokollspezifische Kindelemente dem Element `wsdl:operation` bzw. `wsdl:binding` zugefügt.

Abbildung 47 zeigt ein Beispiel für eine Definition einer gebundenen Schnittstelle. Erweitert wird die abstrakte Schnittstelle aus Abbildung 46. Wieder werden nur die Operationen `Login` und `Get` gezeigt und weitere mit drei Punkten („...“) angedeutet. Zusätzlich zu den eben beschriebenen Elementen sind bereits Erweiterungen für SOAP enthalten. Diese Elemente haben qualifizierte Namen mit dem Präfix `soap12`.

Erweiterungen von WSDL zu Beschreibung eines bestimmten Protokolls werden als Bindung bezeichnet. Die in Abbildung 47 gezeigten Erweiterungen der Bindung für SOAP (SOAP-Bindung) werden im zweiten Teil des Arbeitsentwurfes des WSDL-Standards [Moreau 02] definiert. Sie werden im nachfolgenden Kapitel 4.3.7 genauer erläutert. Neben der SOAP-Bindung definiert [Moreau 02] auch ein Bindung für den Nachrichtenaustausch direkt mit HTTP ohne SOAP. Außerdem werden Elemente definiert, mit denen abstrakte Nachrichten Nachrichtenformaten zugeordnet werden können, die mit MIME-Typen (siehe [Freed 96]) beschrieben sind. Bindungen für andere Protokolle müssen unabhängig vom WSDL-Standard in anderen Spezifikationen beschrieben werden.

```
<wsdl:binding
    name="NewsServiceBinding"
    type="nwsi:NewsServicePortType"
    xmlns:soap12="http://www.w3.org/2002/07/wsdl/soap12">
  <soap12:binding
     transport="http://www.w3.org/2002/06/soap/bindings/HTTP/" />

  <wsdl:operation name="Login" >
    <soap12:operation style="document"/>
    <wsdl:input> <soap12:body use="literal" /></wsdl:input>
    <wsdl:output><soap12:body use="literal" /></wsdl:output>
    <wsdl:fault name="AccessDenied">
      <soap12:fault name="AccessDenied" use="literal" />
    </wsdl:fault>
  </wsdl:operation>

  <wsdl:operation name="Get" >
    <soap12:operation style="document"/>
    <wsdl:input> <soap12:body use="literal" /></wsdl:input>
    <wsdl:output><soap12:body use="literal" /></wsdl:output>
    <wsdl:fault name="InvalidSessionID">
      <soap12:fault name="InvalidSessionID" use="literal" />
    </wsdl:fault>
    <wsdl:fault name="InvalidMessageID">
      <soap12:fault name="InvalidMessageID" use="literal" />
    </wsdl:fault>
    <wsdl:fault name="MessageLocked">
      <soap12:fault name="MessageLocked" use="literal" />
    </wsdl:fault>
  </wsdl:operation>
  ...
</wsdl:binding>
```

Abbildung 47: Ausschnitt von Definition einer gebundenen Schnittstelle in WSDL

4.3.7 SOAP-Bindung

Die SOAP-Bindung ermöglicht in der gebundenen Schnittstelle anzuzeigen, dass SOAP verwendet wird und über welches Protokoll SOAP-Nachrichten ausgetauscht werden. Für Body- und Headereinträge sowie Detaileinträge in Fehlernachrichten kann festgelegt werden, wie sie sich aus den abstrakten Nachrichten ergeben. Es gibt unterschiedliche Wege, das zu beschreiben.

Bei der Darstellung der SOAP-Bindung ergibt sich für diese Arbeit das Problem, dass sich der erste Arbeitsentwurf des zweites Teils des WSDL-Standards [Moreau 02] noch in einer frühen Entwicklungsphase befindet. Er ist noch weitgehend wörtlich identisch mit dem entsprechenden Kapitel der W3C-Note von WSDL 1.1 [Christensen 01]. Bei der Übertragung auf SOAP 1.2 sind offensichtlich Fehler aufgetreten. So ist in [Moreau 02] im Kapitel 2.5 der Begriff „Body“ aus [Christensen 01] mehrfach durch den Begriff „Fault“ ersetzt worden, der im verwendeten Kontexten nicht sinnvoll erscheint. Daher wurde auch die W3C-Note von WSDL 1.1 [Christensen 01] für die nachfolgende Darstellung berücksichtigt.

Da sich die SOAP-Bindung von WSDL 1.1 auf SOAP 1.1 bezieht, können bisher auch mit WSDL 1.2 nur an SOAP 1.1 gebundene Schnittstellen beschrieben werden. Spätere Arbeitsentwürfe und der entgültige Standard von WSDL 1.2 werden dagegen eine Bindung für SOAP 1.2 enthalten. Diese Arbeit versucht in der nachfolgenden Darstellung die SOAP-Bindung von SOAP 1.1 grob auf SOAP 1.2 zu übertragen. Das ist jedoch nicht vollständig möglich. So lässt sich die Verwendung des SOAP-Response-

Kommunikationsmusters nicht beschreiben, weil mit der SOAP-Bindung nur SOAP-Nachrichten beschrieben werden können. Der Request in SOAP-Response-Kommunikationsmuster ist jedoch keine SOAP-Nachricht.

Die SOAP-Bindung stellt für jedes im vorherigen Kapitel 4.3.6 vorstellte Element zur Beschreibung von gebundenen Schnittstellen Kindelemente zur Verfügung. Die Elemente `wsdl:input` und `wsdl:output` werden um das Kindelement `soap12:body`[21] sowie optional um Kindelemente `soap12:header` ergänzt, mit denen Body- und Headereinträge beschrieben werden. Analog erhalten Elemente `wsdl:fault` ein Kindelement `soap12:fault` zur Beschreibung der Detaileinträge in SOAP-Fehlernachrichten. Das Element `wsdl:operation` enthält das Kindelement `soap12:operation`, das die Operation als ganzes SOAP-spezifisch beschreibt.

Das Kindelement von `wsdl:binding` heißt `soap12:binding` und dient vor allem zum Anzeigen, dass die SOAP-Bindung verwendet wird. Daneben enthält es im Attribut `transport` eine URI, die die Protokollbindung von SOAP bezeichnet (siehe Kapitel 3.10). Wird z. B. die URI `http://www.w3.org/2002/06/soap/bindings/HTTP/` angegeben, wie in Abbildung 47 geschehen, legt das fest, dass SOAP-Nachrichten mit HTTP ausgetauscht werden.

Wie die abstrakte Nachricht, die in der abstrakten Schnittstelle festgelegt wird, auf Bodyeinträge der SOAP-Nachricht abgebildet wird, wird durch Attribute von `soap12:body` sowie ein Attribut von `soap12:operation` bestimmt. Dabei bestimmt das Attribut `soap12:body/@parts` welche Parts der abstrakten Nachricht im SOAP-Body enthalten sein sollen. Die Namen der Parts werden durch Leerzeichen separiert im Attribut angegeben. Fehlt das Attribut wie im Beispiel in Abbildung 47 sind alle Parts der abstrakten Nachricht im SOAP-Body enthalten.

4.3.7.1 Style-Attribut

Das Attribut `soap12:operation/@style` (Style-Attribut) bestimmt, ob die Parts direkt dem SOAP-Body zugefügt werden oder einem im SOAP-Body enthaltenen Bodyeintrag. Hat das Style-Attribut den Wert `document`, werden die Parts direkt dem SOAP-Body zugefügt. Hat es dagegen den Wert `rpc` wird ein Bodyeintrag erzeugt und diesem die Parts zugefügt. Der erzeugte Bodyeintrag hat einen qualifizierten Namen, dessen lokaler Teil der Name der Operation ist. Die URI seines Namensraumes wird im Attribut `soap12:body/@namespace` (Namespace-Attribut) angegeben. Das Namespace-Attribut wird nicht benötigt, wenn kein Bodyeintrag erzeugt wird.

In der gebundenen Schnittstelle in Abbildung 47 hat das Style-Attribut der Operation `Login` den Wert `document`. Das Namespace-Attribut wurde nicht benötigt und ist daher nicht angegeben. Zusammen mit der abstrakten Schnittstelle in Abbildung 46, der Nachrichtendeklaration in Abbildung 45 sowie dem Schemadokument in Abbildung 44 ergibt sich so die SOAP-Nachricht in Abbildung 48. Wäre für das Style-Attribut der Wert `rpc` angegeben worden und ein Namespace-Attribut mit dem Wert `http://example.org/sample`, hätte sich statt dessen die SOAP-Nachricht in Abbildung 49 ergeben. Sie unterscheidet sich von der in Abbildung 48 durch das zusätzlich erzeugte Element `my:Login`, dem der einzige Part der abstrakten Nachricht als Kindelement zugefügt wurde.

---

[21] Das Präfix `soap12` stehe für den Namensraum `http://www.w3.org/2002/07/wsdl/soap12` der SOAP-Bindung.

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope"
        xmlns:nwst="http://example.org/NewsService020917/Types">
  <env:Body>
    <nwst:Login>
      <nwst:Name>Venzke</nwst:Name>
      <nwst:Password>igelgruen</nwst:Password>
    </nwst:Login>
  </env:Body>
</env:Envelope>
```

Abbildung 48: Beispiel für SOAP-Nachricht beschrieben mit `style="document"`

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope"
        xmlns:nwst="http://example.org/NewsService020917/Types">
  <env:Body>
    <my:Login xmlns:my="http://example.org/sample">
      <nwst:Login>
        <nwst:Name>Venzke</nwst:Name>
        <nwst:Password>igelgruen</nwst:Password>
      </nwst:Login>
    </my:Login>
  </env:Body>
</env:Envelope>
```

Abbildung 49: Beispiel für SOAP-Nachricht beschrieben mit `style="rpc"`

4.3.7.2 Use-Attribut

Das Attribut `soap12:body/@use` bestimmt, ob die XML-Fragmente der Parts mit Transformationsregeln umgewandet oder direkt in den SOAP-Body übernommen werden. Es soll als Use-Attribut bezeichnet werden und wird auch zur Beschreibung von Header- und Detaileinträgen verwendet.

Hat das Use-Attribut den Wert `encoded`, werden auf das XML-Fragment jedes Parts eine Menge von Transformationsregeln (Kodierung) angewendet und das Ergebnis dem SOAP-Body zugefügt. Abhängig vom Wert des Style-Attributes wird es direkt dem SOAP-Body oder dem erzeugten Bodyeintrag zugefügt. Die zu verwendenden Kodierungen werden im Attribut `soap12:body/@encodingStyle` (Encodingstyle-Attribut) angegeben. Ebenso wie das Encodingstyle-Attribut von SOAP aus Kapitel 3.5 enthält es leerzeichensepariert URIs, die jeweils eine Kodierung bezeichnen. Es gibt keine Beschränkungen, wer Kodierungen definiert und was sie ausdrücken dürfen. Der Name des Parts und der Wert des Namespace-Attributes können für die Kodierungen als Parameter verwendet werden.

Hat das Use-Attribut dagegen den Wert `literal`, werden die XML-Fragmente der Parts nicht transformiert, sondern direkt in den SOAP-Body übernommen. Wieder werden sie abhängig vom Wert des Style-Attributes direkt im SOAP-Body oder im erzeugten Bodyeintrag zugefügt. Wurde der Part mit dem Attribut `wsdl:part/@element` definiert, wird für ihn ein Element zugefügt, das den im Attribut enthaltenen qualifizierten Namen hat. Wurde der Part dagegen mit `wsdl:part/@type` definiert, wird der dort angegebene Typ zum Typ des SOAP-Bodys bzw. des erzeugten Bodyeintrages.

Bei der Operation `Login` in der gebundenen Schnittstelle aus Abbildung 47 hat das Use-Attribut den Wert `literal`. Da außerdem das Style-Attribut den Wert `document` hat und der Part mit dem Attribut `wsdl:part/@element` definiert wurde, bildet der Part den einzigen Bodyeintrag. Es ergibt sich die SOAP-Nachricht in Abbildung 48. Der Body-

eintrag hat den qualifizierter Namen `nwst:Login`, weil dieser Name im Attribut `wsdl:part/@element` angegeben wurde. Das Schemadokument in Abbildung 44 enthält die Deklaration des Elementes mit diesem Namen.

4.3.7.3 Headereinträge

Headereinträge werden ähnlich den Bodyeinträgen definiert. Im Unterschied zu Bodyeinträgen ist es aber nicht notwendig, jeden Headereintrag zu definieren. Headereinträge dürfen SOAP-Nachrichten auch dann zugefügt werden, wenn sie im WSDL-Dokument nicht beschrieben sind.

Statt mit einem Element `soap12:body` werden Headereinträge mit dem Element `soap12:header` definiert. Jedes dieser Elemente beschreibt einen Headereintrag. Sollen einer SOAP-Nachricht mehrere Headereinträge zugefügt werden, müssen dem Element `wsdl:input` bzw. `wsdl:output` mehrere Kindelemente `soap12:header` zugefügt werden.

In einer abstrakten Nachricht wird ein Headereintrag als ein Part beschrieben. Dessen Name wird im Attribut `soap12:header/@part` angegeben. Der Part muss nicht in der abstrakten Nachricht enthalten sein, in der auch die Parts für den SOAP-Body beschrieben sind. Daher muss für Headereinträge neben dem Namen des Parts auch der qualifizierte Name der abstrakten Nachricht im Attribut `soap12:header/@message` angegeben werden.

Das Use-Attribut hat zur Beschreibung von Headereinträgen die gleiche Bedeutung wie bei Bodyeinträgen. Auch Headereinträge können mit Kodierungen aus dem Part der abstrakten Nachricht gebildet werden, wobei das Encodingstyle-Attribut wieder auf die verwendeten Kodierungen verweist. Alternativ kann das XML-Fragment des Parts auch wieder direkt als Headereintrag verwendet werden. Für das Style-Attribut wird bei Headereinträgen stets der Wert `document` angenommen.

Abbildung 67 auf Seite 113 gibt ein Beispiel für eine gebundene Schnittstelle in der ein Headereintrag deklariert wird. Requests der Operation `Get` wird jeweils ein Headereintrag `nwsi:TipURLHeader` zugefügt, ein Beispiel wird in Abbildung 68 auf Seite 114 gezeigt. Der Headereintrag wird in einer eigenen abstrakten Nachricht definiert, unabhängig von der für den Bodyeintrag verwendeten.

4.3.7.4 Fehlernachrichten

Da die Struktur von SOAP-Fehlernachrichten bereits durch den SOAP-Standard festgelegt ist, muss sie in gebundenen Schnittstellen nicht erneut definiert werden. Definiert wird lediglich, welche Detaileinträge in SOAP-Fehlernachrichten enthalten sind. Es gibt keine Möglichkeit andere Teile der SOAP-Fehlernachricht festzulegen, wie z. B. den angegebenen Fehlercode.

Anders als Headereinträge werden Fehlernachrichten schon in der abstrakten Schnittstelle beschrieben und wie Bodyeinträge in der gebundenen Schnittstelle nur genauer beschrieben. In der abstrakten Schnittstelle werden zur Fehlerbehandlung abstrakte Nachrichten festgelegt, die bei einem Fehler als zweite Nachricht des Operationsaufrufes gesendet werden. Für die SOAP-Bindung ist festgelegt, dass solche abstrakten Nachrichten nur einen Part haben dürfen. Wie dieser in der SOAP-Nachricht im Element `env:Body/env:Fault/env:Detail` repräsentiert wird, ist in WSDL mit einem Element `wsdl:fault/soap12:fault` festgelegt.

Wie bei Body- und Headereinträgen kann auch hier festgelegt werden, ob auf den Part Kodierungen angewendet werden sollen oder nicht. Das wird wieder mit dem Use-Attribut (`soap12:fault/@use`) gesteuert. Dazu können wieder im Encodingstyle-Attribut (`soap12:fault/@encodingStyle`) die Kodierungen festgelegt werden, wobei der Inhalt des Namespace-Attributes (`soap12:fault/@namespace`) als zusätzlicher Parameter dienen kann. Wie bei Headereinträgen wird für das Style-Attribut stets der Wert `document` angenommen.

Für die beiden in Abbildung 47 gezeigten Operationen `Login` und `Get` sind Fehlernachrichten deklariert. Die Operation `Get` darf drei Fehlernachrichten als Response zurückliefern. In der gebundenen Schnittstelle wird mit dem Use-Attribut festgelegt, dass für den Part der abstrakten Nachricht keine Kodierung verwendet wird. In der abstrakten Schnittstelle aus Abbildung 46 wird für die zweite Fehlernachricht `InvalidMessageID` festgelegt, dass sie durch die abstrakte Nachricht `nwsi:FaultDetailsMessageIDs` beschrieben wird. Deren Deklaration wurde bereits in Abbildung 45 gezeigt. Der einzige Part verweist darin mit dem Attribut `wsdl:part/@element` auf die Deklaration eines Elementes `nwst:MessageIDsSequence`. Dieses Element ist daher der Detaileintrag in der SOAP-Fehlernachricht, wie im Beispiel in Abbildung 10 auf Seite 34 zu erkennen.

4.3.8 Webservice

Die bisher beschriebenen Sprachelemente von WSDL dienen zum Beschreiben der Schnittstellen von Webservices. Neben der Kenntnis von Schnittstellen, muss einem Webclient auch bekannt sein, welche Schnittstellen von einem bestimmten Webservice unterstützt werden und über welche Netzwerkadressen er mit diesen kommunizieren kann. Das wird in Elementen `wsdl:definitions/wsdl:service` festgelegt.

Jedes Element `wsdl:service` deklariert genau einen Webservice, dessen Kindelemente `wsdl:portType` jeweils eine vom Webservice unterstützte, gebundene Schnittstelle beschreiben, die als Endpunkt bezeichnet wird. Im Attribut `wsdl:portType/@binding` wird der qualifizierte Name der gebundenen Schnittstelle angegeben. Kindelemente von `wsdl:portType` müssen bindungsspezifisch definiert werden, so dass mit ihnen genau eine Netzwerkadresse für den Endpunkt festgelegt werden kann. Die SOAP-Bindung definiert hierzu das Element `soap12:address`, in dessen Attribut `location` eine URI für den Endpunkt angegeben wird.

Gegenüber WSDL 1.1 zugefügt wurden in WSDL 1.2 Webservicetypen. Ein Webservicetyp beschreibt eine Menge abstrakter Schnittstellen. Sein qualifizierter Namen wird bei der Deklaration eines Webservices im Attribut `wsdl:service/@serviceType` angegeben, um festzulegen, welche Menge von abstrakten Schnittstellen der Webservice unterstützt. Definiert wird ein Webservicetyp im WSDL-Dokument mit einem Element `wsdl:definitions/wsdl:serviceType`. Jedes seiner Kindelemente `wsdl:portType` beschreibt eine abstrakte Schnittstelle, die zum Webservicetyp gehört. Ihr qualifizierter Name wird im Attribut `wsdl:portType/@name` angegeben.

Abbildung 50 zeigt ein Beispiel für die Deklaration eines Webservices sowie seines Webservicetyp. Der Webservice hat nur einen Endpunkt. Daher hat das Element `wsdl:service` nur ein Kindelement `wsdl:port`. Es verweist auf die in Abbildung 47 gezeigte gebundene Schnittstelle. Sie verwendet die SOAP-Bindung mit HTTP zur Übertragung von SOAP-Nachrichten. Daher wird das Element `soap12:address` zum Festlegen der Netzwerkadresse verwendet. Im Attribut `location` ist die URL des Endpunktes angegeben. Wegen der Verwendung von HTTP beginnt sie mit „`http:`“.

```
<wsdl:serviceType name="NewsServiceType">
  <wsdl:portType name="nwsi:NewsServicePortType" />
</wsdl:serviceType>

<wsdl:service
    name="NewsService"
    serviceType="nwsi:NewsServiceType">
  <wsdl:port
      name="NewsServicePort"
      binding="nwsi:NewsServiceBinding">
    <soap12:address
        location="http://soap.example.org/NewsService/News.aspx" />
  </wsdl:port>
</wsdl:service>
```

Abbildung 50: Beispiel für Deklaration eines Dienstes und Definition eines Diensttyps in WSDL

5 Präzisierungsstandard für Interoperabilität

Das in den Kapitel 3 bzw. 4 vorgestellte SOAP und WSDL hat eine Vielzahl von Optionen. So können z. B. in SOAP unterschiedliche Kommunikationsmuster verwendet werden, ebenso wie unterschiedliche Kodierungen von Nachrichten und Protokollbindungen. WSDL erlaubt die Verwendung unterschiedlicher Typsysteme und die Definition der gleichen Schnittstellen auf unterschiedliche Arten.

Werkzeuge mit denen Webkomponenten implementiert werden, können nicht alle denkbaren Optionen implementieren. Verwenden unterschiedliche Werkzeuge unterschiedliche Optionen, kann das dazu führen, dass mit ihnen implementierte Webkomponenten nicht miteinander interoperabel sind. Offensichtlich ist das z. B. der Fall, wenn unterschiedliche Protokollbindungen verwendet werden und damit unterschiedliche Protokolle zum Transport von SOAP-Nachrichten.

Dieses Kapitel schlägt daher vor, den Standards von SOAP und WSDL Präzisierungsstandards zuzufügen. Präzisierungsstandards legen Optionen fest, die Implementierungen verwenden sollten, um mit anderen interoperabel zu sein. Diese Festlegungen werden danach in dieser Arbeit verbindlich vorausgesetzt. Zunächst soll untersucht werden, warum Optionen in den Standards von SOAP und WSDL überhaupt erforderlich sind.

5.1 Grundidee

5.1.1 Breite Standards

Der SOAP- und WSDL Standard enthalten Optionen, damit sie langfristig stabil gehalten werden können. Langfristige Stabilität des SOAP-Standards wird z. B. in [Apparao 02][22] gefordert, wo ausgeführt wird, dass sie durch Modularität und Schichtung erreicht werden soll. Optionen wie Protokollbindungen erlauben diese Schichtung. Die von SOAP-Standard unabhängige Definition von z. B. Kommunikationsmustern erlaubt Modularität. Sollte nun der technische Forschritt andere, solche Optionen fordern, kann der SOAP-Standard stabil bleiben, lediglich die verwendeten Optionen ändern sich.

Ein anderer Grund für Optionen im SOAP- und WSDL Standard sind unterschiedliche Interessengruppen, die an der Erstellung der Standards beteiligt sind. Tim Ewald sieht in [Ewald 01] drei Interessengruppen, er bezeichnet sie als „Camps“[23]. Es sind:

- das Distributed Object Camp,
- das Workflow Camp,
- das Simple Interop Camp.

Mitglieder des Distributed Object Camps sehen SOAP und WSDL als eine Möglichkeit ein besseres Framework für verteilte Objektumgebungen zu schaffen. Hierzu wird ein

---

[22] In [Apparao 02] wird SOAP als XMLP bezeichnet.

[23] Die Darstellung in [Ewald 01] bezieht sich auf SOAP 1.1 bzw. WSDL 1.1. Die Interessengruppen haben sich jedoch mit der Fortentwicklung zu SOAP 1.2 und WSDL 1.2 nicht geändert.

reichhaltiges Programmiermodell für verteilte Objekte benötigt, das z. B. Ereignisse, Konstruktoren, Ausnahmen, Vererbung und Dienstreferenzen enthält. Das verwendete Objektmodell muss zwischen den Kommunikationspartnern einheitlich sein. Wie unterschiedlich Objektmodelle dagegen sein können wurde bereits in Kapitel 2.3.2 aufgezeigt. Tim Ewald stellt in [Ewald 01] fest, dass bereits in den 90'er Jahren eine Einigung auf ein einheitliches Objektmodell gescheitert ist.

Mitglieder des Workflow Camps haben andere Anforderungen an SOAP und WSDL. Sie sehen SOAP und WSDL als eine Möglichkeit, Mechanismen für komplexe, nachrichtenbasierte Workflows zu standardisieren. Dazu benötigen sie ein reichhaltiges nachrichtenorientiertes Programmiermodell. In den Nachrichtenaustausch müssen auch bereits exsistierende Nachrichtenstandards und -formate integriert werden können. Außerdem wird ein gemeinsames Routingmodell für Nachrichten mit Intermediaries und Transaktionen gefordert.

Interoperabilität ist weder für das Distributed Object Camp noch für das Workflow Camp das entscheidende Kriterium. Das ist bei Mitgliedern des Simple Interop Camps anders. Sie sehen SOAP und WSDL als eine Möglichkeit den Austausch von Daten zwischen heterogenen Endpunkten zu ermöglichen. Solche Endpunkte können durch eine Vielzahl unterschiedlicher Einheiten realisiert werden. Diese können mit oder auch ohne spezielle Infrastrukturen implementiert worden sein. Alle diese Einheiten sollen miteinander interoperabel sein. Hierzu reicht ein einfaches nachrichtenorientiertes oder RPC basiertes Programmiermodell. Es sei klargestellt, dass sich der Autor dieser Arbeit als Mitglied des Simple Interop Camps sieht.

Bei der Standardisierung von SOAP und WSDL müssen sich die Mitglieder der drei Interessengruppen auf einheitliche Standards einigen. Dazu müssen ihre unterschiedlichen Ziele beachtet werden, was zu Kompromissen führt, die sich in Optionen und zusätzlicher Komplexität von SOAP und WSDL äußern.

Die Optionen führen auch dazu, dass Implementierungen von SOAP-Knoten (SOAP-Implementierungen) SOAP nicht vollständig implementieren können. Eine SOAP-Implementierung kann z. B. nur eine begrenze Anzahl von Protokollbindungen, Kommunikationsmustern und Kodierungen implementieren. Solche können jedoch beliebig, auch anwendungsspezifisch neu definiert werden. Keine SOAP-Implementierung kann daher alle unterstützen. Darüber hinaus werden einige SOAP-Implementierungen auch aus Gründen der Einfachheit auf Optionen verzichten. Trotzdem sind sie nach den SOAP-Standard konform. Analog verhält es sich auch mit WSDL-Implementierungen.

Für jede SOAP- oder WSDL-Implementierung werden also Annahmen über den SOAP- bzw. WSDL-Standard hinaus verwendet. Es gibt viele solcher Annahmen. Sie können dazu führen, dass zwei Implementierungen nicht miteinander interoperabel sind. Werden Implementierungen unabhängig voneinander erstellt, ist davon auszugehen, dass sie auf unterschiedlichen Mengen von Annahmen beruhen. Damit ist auch nicht unwahrscheinlich, dass sie nicht miteinander interoperabel sind.

Besser wäre es daher, wenn sie auf den gleichen Annahmen beruhen würden. Das wäre für SOAP-Implementierungen problematisch, die mit Sicht des Distributed Object Camps oder des Workflow Camps implementiert werden, weil diese spezielle Anforderungen haben. Für SOAP- und WSDL-Implementierungen mit Sicht des Simple Interop Camps, die vor allem die Interoperabilität als Anforderung haben, wäre es dagegen ein großer Vorteil.

5.1.2 Präzisierungsstandards

Als Konsequenz aus diesen Überlegungen könnte gefordert werden, dass für die Interoperabilität der SOAP- und WSDL-Standard präziser gefasst werden müsste, so dass für die Implementierung keine oder nur sehr wenige Annahmen getroffen werden müssten. Das widerspricht jedoch der oben beschriebenen Anforderung für eine langfristige Stabilität auch bei technischem Fortschritt. Außerdem wäre es durch die verschiedenen Interessengruppen viel schwieriger, wenn nicht unmöglich, einen Konsens über den Standard zu erzielen.

Um einerseits Stabilität und Konsens über Standards zu ermöglichen und andererseits für interoperable Implementierungen einheitliche Annahmen zur Verfügung zu stellen, soll in dieser Arbeit eine zweistufige Standardisierung vorgeschlagen werden. Sie beruht auf breiten Standards und auf ihnen aufbauenden Präzisierungsstandards.

Die erste Stufe bilden die breiten Standards, im Falle von Webkomponenten sind das der SOAP- und WSDL-Standard. Sie enthalten viele Optionen, so dass ihre langfristige Stabilität sicherstellt werden kann und für sie ein breiter Konsens möglich ist.

In einer zweiten Stufe werden breite Standards durch Präzisierungsstandards ergänzt. Ein Präzisierungsstandard schränkt einen oder mehrere breite Standards für die Interoperabilität genauer ein. Er enthält also eine Menge von Annahmen, die über den oder die breiten Standards hinaus erforderlich sind, damit unabhängig erstellte Implementierungen miteinander interoperabel sind[24]. Präzisiert er mehrere Standards, kann er auch Anforderungen enthalten, die sich erst aus ihrer Kombination ergeben.

Ideal wäre, wenn es in einem Anwendungsbereich, wie dem der Webkomponenten, für die Interoperabilität nur einen Präzisierungsstandard gäbe und nicht mehrere alternative. Sollte ein Konsens jedoch nicht möglich sein, wäre die Folge, dass es mehrere Präzisierungsstandards gäbe, deren Implementierungen möglicherweise nicht interoperabel sind. Solche miteinander konkurrierenden Präzisierungsstandards können auch durch den technischen Fortschritt entstehen, wenn die Annahmen eines Präzisierungsstandards veraltet sind.

Obwohl es dann mehrere konkurrierende Präzisierungsstandards gäbe, wäre die Situation besser als ohne Präzisierungsstandards. Eine Implementierung könnte dann nach einem der wenigen Präzisierungsstandards implementiert werden. Es könnte auch versucht werden, die Implementierung so zu erstellen, dass sie mehreren Präzisierungsstandards entspricht. Ohne Präzisierungsstandard muss dagegen für die Implementierung eine eigene Menge von Annahmen über dem breiten Standard gewählt werden, die sich von denen anderer Implementierungen mit großer Wahrscheinlichkeit unterscheidet.

Der wichtiger Vorteil von Präzisierungsstandards ergibt sich schon dadurch, dass für eine Implementierung angegeben werden kann, welchem oder welchen Präzisierungsstandards sie entspricht. Die Benennung aller über den breiten Standard getroffenen Annahmen wäre viel aufwendiger. Für die Angabe eines Präzisierungsstandards reicht dagegen die Angabe seines Namens.

[24] Die zweistufige Standardisierung könnte auch auf mehr Stufen erweitert werden, wenn Präzisierungsstandards mit zusätzlichen Präzisierungsstandards weiter präzisiert werden. Das soll in dieser Arbeit jedoch nicht betrachtet werden.

5.1.3 Präzisierungsstandard für SOAP und WSDL

In dieser Arbeit sollen nachfolgendend Annahmen für SOAP und WSDL als Grundlage für einen Präzisierungsstandard vorgestellt werden. Dazu werden wichtige, für die Interoperabilität relevante Optionen von SOAP und WSDL untersucht und vorgeschlagen, welche verwendet werden sollten. Ziel für die Wahl von Optionen muss stets die Frage sein, wie sich am zuverlässigsten Interoperabilität zwischen Webkomponenten erreichen lässt.

Die Einfachheit einer Option ist von Vorteil, insbesondere wenn sie den Spielraum für die Interpretation des Präzisierungsstandards einschränkt. Es sollte nicht unnötig mehrere Optionen geben, wenn diese für den SOAP-Empfänger einer SOAP-Nachricht relevant sind. Dagegen ist es kein Problem, wenn ein SOAP-Sender eine SOAP-Nachricht in einer Form erweitert, so dass der SOAP-Empfänger die Erweiterungen unproblematisch ignorieren kann.

Als weiteres Ziel wird gefordert, dass klar zwischen der Kommunikation zwischen Webkomponenten und Beschreibung der Webservices unterschieden wird. Die Beschreibung eines Webservice ist bereits für die Entwicklung von Webclients erforderlich, also bevor die Kommunikation stattfindet. Sie sollte daher unabhängig von der Kommunikation abgefragt werden können. Das ist die Aufgabe von WSDL-Dokumenten. Die Aufgabe von SOAP ist dagegen die Kommunikation zwischen Webkomponenten, also Webclients und Webservices.

5.2 Präzisierung von SOAP

5.2.1 Kommunikationsmuster

Die Kommunikation zwischen Webclients und Webservices findet durch Operationsaufrufe statt, für die mehrere SOAP-Nachrichten ausgetauscht werden. Das Muster nach denen SOAP-Nachrichten für einen Operationsaufruf ausgetauscht werden, wird als Kommunikationsmuster bezeichnet. Der SOAP-Standard erlaubt beliebige. (Siehe Kapitel 3.8).

Grundsätzlich sollten nur Kommunikationsmuster verwendet werden, die im SOAP-Standard enthalten sind, also das Request-/Response- und das SOAP-Response-Kommunikationsmuster. Alternativ kann für einen Operationsaufruf auch nur eine einzelne Nachricht ausgetauscht werden. Bei Kommunikationsmustern, die in anderen Spezifikationen oder anwendungsspezifisch definiert sind, kann nicht davon ausgegangen werden, dass sie jeder SOAP-Implementierung bekannt sind. Ihre Verwendung würde daher zu Interoperabilitätsproblemen führen.

Um Operationsaufrufe von Webkomponenten zu realisieren, müssen Nachrichten in Request-/Response-Semantik übertragen werden, wie in Kapitel 2.4 festgestellt wurde. Die benötigte Request-/Response-Semantik wird sowohl vom Request-/Response-Kommunikationsmuster als auch vom SOAP-Response-Kommunikationsmuster unterstützt, jedoch nicht von einzelnen Nachrichten.

Wie bereits in Kapitel 3.8 ausgeführt, kann das SOAP-Response-Kommunikationsmuster nur in bestimmten Fällen verwendet werden, das Request-/Response-Kommunikationsmuster jedoch in allen. Das SOAP-Response-Kommunikationsmuster kann nur zur Abfrage einer Ressource ohne Seiteneffekte verwendet werden. Alle In- und Inout-Parameter der Operation dürfen nur die abzufragende Ressource identifizieren. Neben diesen Parametern sind im Request keine weiteren Daten möglich, auch nicht als Headereinträge.

Diese Einschränkungen sprechen auf Grund fehlender Universalität gegen das SOAP-Response-Kommunikationsmuster. Hinzu kommt, dass es auf unterschiedlichen Nachrichtentypen beruht. Nur für den Response wird eine SOAP-Nachricht verwendet, während der Request für die Protokollbindung spezifisch ist.

Um nicht unnötig mehrere Optionen zuzulassen, sollte der Präzisierungsstandard nur ein Kommunikationsmuster vorschreiben. Im SOAP-Standard nicht beschriebene Kommunikationsmuster können SOAP-Implementierungen in vielen Fällen unbekannt sein. Einzelne SOAP-Nachrichten unterstützen die benötigte Request-/Response-Semantik nicht. Für das SOAP-Response-Kommunikationsmuster ist die Universalität nicht gegeben. Daher sollte der Präzisierungsstandard die Verwendung des Request-/Response-Kommunikationsmusters vorschreiben.

5.2.2 Encodingstyle-Attribut

Für XML-Fragmente in einer ausgetauschten SOAP-Nachricht, kann mit Hilfe des Encodingstyle-Attributes die Kodierung angegeben werden, die bestimmt, nach welchen Transformationsregeln die XML-Fragmente aus Datenstrukturen der Implementierungssprache gebildet wurden. Der SOAP-Standard erlaubt beliebige Kodierungen. Der SOAP-Empfänger kann diese Information verwenden, um aus einer empfangenen SOAP-Nachricht wieder die Datenstrukturen der Implementierungssprache aufzubauen.

Selbstverständlich ist das nur möglich, wenn der SOAP-Empfänger die verwendete Kodierung und damit den Wert des Encodingstyle-Attributes kennt. Ist es erforderlich, dass SOAP-Empfänger den Wert interpretieren können, dann sollte wie bei den Kommunikationsmustern nur das SOAP-Encoding verwendet werden, das im SOAP-Standard definiert wurde. Für andere Kodierungen kann nicht davon ausgegangen werden, dass sie bekannt sind. Allerdings sei noch einmal darauf hingewiesen, dass das SOAP-Encoding nur die Transformation zwischen dem SOAP-Datenmodell und XML-Fragmenten in der SOAP-Nachricht beschreibt. Abbildungen zwischen Implementierungssprachen und dem SOAP-Datenmodell sind nicht standardisiert.

Auch wenn das SOAP-Encoding verwendet wird und es somit jedem SOAP-Empfänger bekannt ist, bleibt zweifelhaft, wie sinnvoll das Encodingstyle-Attribut für ihn tatsächlich ist. Er kann zwar möglicherweise die Datenstruktur aufbauen, die beim SOAP-Sender in ein XML-Fragment transformiert wurde. Für die Implementierung des SOAP-Empfängers ist jedoch eine bestimmte Datenstruktur erforderlich. Diese muss nicht mit der identisch sein, die im SOAP-Sender verwendet wird.

Zumindest muss der SOAP-Empfänger prüfen, ob die Rücktransformation zu benötigten Datenstrukturen führt. Hierzu verwendet er sein a priori Wissen über die benötigten Datenstrukturen, die in jedem Fall schon zur Entwicklungszeit vorhanden sein müssen, weil dann die Implementierung gegen diese Datenstrukturen implementiert wird.

Zur Entwicklungszeit ist auch schon a priori Wissen darüber vorhanden, wie ausgetauschte SOAP-Nachrichten aufgebaut sind. Es ist in der Spezifikation des Webservices enthalten. Ein SOAP-Empfänger kann mit diesem Wissen prüfen, ob empfangene SOAP-Nachrichten der Spezifikation entsprechen. Außerdem kann damit Code erzeugt werden, der die XML-Fragmente in der SOAP-Nachricht in die Datenstrukturen umwandelt, die die Implementierung benötigt. Ein auf diese Art konstruierter SOAP-Empfänger benötigt das Encodingstyle-Attribut nicht, weil für ihn nicht wichtig ist, welche Datenstrukturen beim SOAP-Sender verwendet wurden, sondern nur welche Struktur die erwartet SOAP-Nachricht hat.

Die Diskussion zeigt, dass das Encodingstyle-Attribut nicht erforderlich ist. Zusätzlich muss festgestellt werden, dass es auch dem Ziel widerspricht, dass klar zwischen der Kommunikation zwischen Webkomponenten und der Beschreibung von Webservices unterschieden werden soll. Das Encodingstyle-Attribut beschreibt den Inhalt der SOAP-Nachricht, indem es angibt, wie diese gebildet wurde. Es gehört also nicht zu Kommunikation selbst, sondern zu ihrer Beschreibung und ist somit Teil der Schnittstellenbeschreibung.

Ein Präzisierungsstandard sollte daher verbieten, dass das Encodingstyle-Attribut für einen SOAP-Empfänger relevant ist. Allerdings kann das SOAP-Encoding selbst als Konvention hilfreich sein, um zu beschreiben, wie SOAP-Nachrichten einheitlich gebildet werden sollten. Der SOAP-Standard empfiehlt, dass auf die Verwendung des SOAP-Encodings mit dem Encodingstyle-Attribut hingewiesen wird. Der Präzisierungsstandard sollte das daher nicht verbieten. Statt dessen sollte gefordert werden, dass ein Empfänger das Encodingstyle-Attribut ignorieren sollte oder seine Verarbeitung zumindest nicht auf dessen Wert angewiesen sein sollte.

5.2.3 Verwendung von xsi:type

Der SOAP-Standard erlaubt, den Typ eines Elementes innerhalb einer SOAP-Nachricht mit Hilfe des Attributes `xsi:type` anzugeben, dass in Kapitel 4.2.2 vorgestellt wurde. Noch offensichtlicher als beim Encodingstyle-Attribut wird damit der Inhalt der SOAP-Nachricht in der SOAP-Nachricht selbst beschrieben. Das widerspricht wieder der Trennung zwischen Nachrichtenaustausch und Beschreibung des Webservices.

Statt Typinformationen in der SOAP-Nachricht zu speichern, werden Header-, Body- und Detaileinträge in SOAP-Nachrichten mit Hilfe von qualifizierten Namen identifiziert. Qualifizierte Namen geben zwar nicht direkt Auskunft über Typen, erlauben es aber, Elemente oder Attribute eindeutig ihrer Beschreibung zuzuordnen. Die Beschreibung in XML-Schema bzw. in der Beschreibung des Webservices ermöglicht es, ihre Typen zu bestimmen. Ein Präzisierungsstandard sollte diese Vorgehensweise vorschreiben und die Verwendung des Attributes `xsi:type` verbieten.

5.2.4 Repräsentation von Operationen

Der SOAP-Standard enthält Konventionen wie Remote Procedure Calls repräsentiert werden sollten. Sie wurden in Kapitel 3.9 vorgestellt. Ihre Verwendung ist optional. Es wäre naheliegend, sie für die Repräsentation von Operationen von Webkomponenten zu verwenden.

Zu Kritisieren ist an der RPC-Repräsentation jedoch, dass sie nur mit einer Kodierung wie dem SOAP-Encoding verwendet werden können. Der Grund liegt in der Tatsache, dass zu übertragene Parameter zunächst zu einer Struktur oder einem Array zusammengefasst werden, wie es im SOAP-Datenmodell verwendet wird. Erst mit Hilfe einer Kodierung wird dann die Struktur bzw. das Array in ein XML-Fragment gewandelt, das im SOAP-Body übertragen wird.

Welche Kodierung verwendet wurde, muss dem SOAP-Empfänger bekannt sein, damit er überhaupt in der Lage ist, die Parameter voneinander zu isolieren. Diese Information ist in der SOAP-Nachricht im Encodingstyle-Attribut vorhanden. Allerdings wurde in Kapitel 5.2.2 gefordert, dass der SOAP-Empfänger das Encodingstyle-Attribut nicht für seine Verarbeitung benötigen darf, wie es hier der Fall wäre.

Die einzige Kodierung, die mit Blick auf Interoperabilität in Frage kommt, ist das SOAP-Encoding, weil nur diese Kodierung im Standard definiert ist und daher als bekannt vorausgesetzt werden kann. Das SOAP-Encoding setzt jedoch die Verwendung des SOAP-Datenmodells voraus. Daraus folgt, dass auch die Parameter im SOAP-Datenmodell vorliegen müssen, damit sie zu einer Struktur oder einem Array zusammengefasst und dann in ein XML-Fragment transformiert werden können.

Das schließt aus Parameter zu verwenden, die nicht zweckmäßig auf das SOAP-Datenmodell abgebildet werden können oder sollten, weil sie zum Beispiel bereits als XML-Fragment vorliegen. Somit widerspricht die RPC-Repräsentation sogar dem in der Charter der Web Services Architecture Working Group [Champion 02] formulierten Ziel, dass die Architektur kein Programmiermodell ausschließen darf[25].

Tatsächlich wäre es nicht notwendig gewesen, die RPC-Repräsentation so zu formulieren, dass sie nur zusammen mit Kodierungen verwendet werden kann. Eine Kodierung ist nur deshalb zwingend notwendig, weil Parameter mit Hilfe von Strukturen oder Arrays zusammengefasst werden. Wäre das auf Basis von XML geschehen, könnte die RPC-Repräsentation auch ohne Kodierung verwendet werden.

Im Folgenden soll grob skizziert werden, wie die RPC-Repräsentation in diesem Sinne umformuliert werden könnte. Statt die Parameter zu einer Struktur oder einem Array zusammenzufassen, werden sie zu Kindelementen in einem Bodyeintrag. Der Bodyeintrag ist seinerseits das einzige Kindelement des SOAP-Bodys. Sein qualifizierter Name identifiziert die Operation. Im Request ist es der Name der Operation, ergänzt um ein Präfix. Im Response wird dieser Namen mit der Zeichenkette „`Response`" konkateniert.

Im Request gibt es im Bodyeintrag für jeden In- bzw. Inout-Parameter ein Kindelement (Parametereintrag), analog im Response für jeden Out- bzw. Inout-Parameter. Soll eine Kodierung verwendet werden, wird jeder Parameter einzeln mit Hilfe der Kodierung in ein XML-Fragment transformiert, das aus dem Parametereintrag und seinen Nachfahren besteht. Wenn das Encodingstyle-Attribut angegeben werden soll, würde es daher zum Attribut des Parametereintrages[26].

Bildet man SOAP-Request und SOAP-Response mit diesen Regeln, erhält man weitgehend die gleichen SOAP-Nachrichten, wie mit der RPC-Repräsentation des SOAP-Standards, wenn das SOAP-Encoding angenommen wird. Das zeigt auch ein Vergleich mit den Beispielen für SOAP-Nachrichten in Abbildung 18 auf Seite 40 und Abbildung 21 auf Seite 44. Lediglich das Encodingstyle-Attribut befindet sich statt im Bodyeintrag in jeden Parametereintrag oder entfällt ganz.

Eine weitere Kritik an der RPC-Repräsentation des SOAP-Standards betrifft das Element `rpc:result`, mit dem der Rückgabeparameter einer Operation angezeigt wird. Ein Rückgabeparameter wird in der SOAP-Nachricht zunächst wie ein Out-Parameter kodiert. Mit dem Element `rpc:result` wird zusätzlich angezeigt, dass dieser Parameter der Rückgabeparameter der Operation ist. Diese Vorgehensweise widerspricht der Trennung zwischen dem Austausch von Nachrichten und der Beschreibung des Webservices, weil das Element `rpc:result` den Inhalt der Nachricht beschreibt. Das ist jedoch Aufgabe der Beschreibung des Webservices in WSDL. Daher sollten in der SOAP-Nachricht nur die Parameter der Operation selbst, aber nicht das Element `rpc:result` angegeben sein.

---

[25] Das Ziel wurde in Kapitel 1.1 von [Champion 02] genannt.

[26] Davon unbenommen bleibt die Aussage aus Kapitel 5.2.2, dass die Verarbeitung von SOAP-Empfängern jedoch nicht auf das Vorhandensein des Encodingstyle-Attributes abhängig sein sollte.

Die Parameter sollten alle in den SOAP-Nachrichten enthalten sein oder genauer alle In- und Inout-Parameter im Request und alle Out- und Inout-Parameter im Response. Der SOAP-Standard empfiehlt, dass Parameter, die die Ressource beschreiben, auf die sich die Operation bezieht, auch in der URI enthalten sein sollten, an die der SOAP-Request geschickt wird. Das verbietet jedoch nicht, dass diese Parameter auch in der SOAP-Nachricht enthalten sind, wie auch im Tutorial zum SOAP-Standard [Mitra 02] festgestellt wird. Der SOAP-Sender sollte das tun, weil es die Verarbeitung im SOAP-Empfänger vereinfacht. Dieser kann dann alle benötigten Daten aus der SOAP-Nachricht entnehmen und muss nicht zusätzlich Daten aus der URI extrahieren.

Der Präzisierungsstandard sollte Konventionen für Operationsaufrufe enthalten, die wie oben skizziert, unabhängig von einer Kodierung formuliert sind. Das ist trotz des Vorhandenseins der RPC-Repräsentation im SOAP-Standard erforderlich, weil ein Präzisierungsstandard den breiten Standard nur präzisieren aber nicht ändern darf. Damit darf er die RPC-Repräsentation nicht ändern, sondern muss sie erneut unabhängig von Kodierungen enthalten. Die Verwendung des Elementes `rpc:result` sollte der Präzisierungsstandard verbieten. Außerdem sollte er vorschreiben, dass alle Parameter der Operation im Bodyeintrag der SOAP-Nachricht enthalten sein müssen.

5.2.5 Protokollbindungen

Die zum Aufruf der Operationen benötigten SOAP-Nachrichten werden mit Request-/Response-Semantik ausgetauscht. Im Kapitel 5.2.1 wurde festgelegt, dass der Präzisierungsstandard die Verwendung des Request-/Response-Kommunikationsmusters vorschreiben sollte, das Request-/Response-Semantik bietet. Ein Protokoll, das zum Transport der SOAP-Nachrichten verwendet wird, muss dieses Kommunikationsmuster unterstützen. Der SOAP-Standard erlaubt hierzu beliebige Protokolle.

Webkomponenten können nur miteinander kommunizieren, wenn sie das gleiche Protokoll auf die gleiche Art verwenden, um die SOAP-Nachrichten zu übertragen. Eine Protokollbindung legt beides fest. Daher ist es erforderlich, dass miteinander kommunizierende Webkomponenten die gleiche Protokollbindung verwenden. Ein Präzisierungsstandard sollte eine bestimmte Protokollbindung für alle Webkomponenten zwingend vorschreiben, so dass die Interoperabilität nicht durch unterschiedliche Protokollbindungen gefährdet wird.

Die einzige im SOAP-Standard definierte Protokollbindung ist die für HTTP. Andere Bindungen können in anderen Standards oder auch anwendungsspezifisch definiert werden. Jedoch ist es am ehesten zu erwarten, dass diese im SOAP-Standard definierte Protokollbindung in vielen Implementierungen enthalten sein wird.

Dafür spricht neben der Tatsache, dass die HTTP-Bindung im Standard enthalten ist, das allgemeine Interesse an HTTP im Webumfeld. HTTP ist das Protokoll mit dem Webbrowser auf Websites und Webanwendungen zugreifen. Daher ist es sehr weit verbreitet und gut verstanden. Es wird heute auch für viele andere Anwendungen verwendet, wie z. B. zur Übertragung von Streaming Media (z. B. für Audio- und Videodatenströme) und zur Übertragung von Dateien. Somit entwickelt es sich zu einer Art Universalprotokoll, was die Verwendung im Kontext von SOAP nahe legt. Für SOAP 1.1 hat sich gezeigt, dass es in der Praxis das am häufigsten für die Übertragung von SOAP-Nachrichten eingesetzt wird.

Die HTTP-Bindung unterstützt sowohl das Request-/Response- als auch SOAP-Response-Kommunikationsmuster. Damit erlaubt seine Verwendung das in Kapitel 5.2.1, geforderte Request-/Response-Kommunikationsmuster. Die Zuordnung von Request und Response ist bei der HTTP-Bindung bereits durch HTTP möglich und muss daher nicht mit Hilfe von Headereinträgen geschehen.

Wie die SOAP-Nachricht serialisiert wird, legt die HTTP-Bindung nicht fest. Wie in Kapitel 3.10 ausgeführt wurde, muss jede Implementierung jedoch die Serialisierung nach XML 1.0 gemäß [Bray 00] unterstützen. Ein SOAP-Empfänger kann eine SOAP-Nachricht nur dann versteht, wenn er ihre Serialisierung kennt. Daher ist es für die Interoperabilität ein Vorteil, wenn festgelegt wird, welche Serialisierungen erlaubt sind. Da ohnehin jede SOAP-Implementierung die Serialisierung nach XML 1.0 unterstützen muss, sollte festgelegt werden, dass diese immer verwendet werden muss.

Leider bedeutet die Festlegung auf die Serialisierung von XML 1.0 nicht, dass ein SOAP-Empfänger eine SOAP-Nachricht in jedem Fall deserialisieren kann. Der Grund ist, dass bei der Serialisierung nach XML 1.0 unterschiedliche Zeichensätze verwendet werden können. Erlaubte Zeichensätze sollten festgelegt werden. Der Standard von XML 1.0 [Bray 00] legt fest, dass jeder XML-Prozessor die Zeichensätze UTF-8 und UTF-16 unterstützen muss. Er kann optional weitere Zeichensätze unterstützen. Daher liegt es nahe, für SOAP-Nachrichten nur UTF-8 und UTF-16 zu erlauben.

Zusammenfassend sei festgestellt, dass ein Präzisierungsstandard vorschreiben sollte, dass nur das Protokoll HTTP zum Austausch von SOAP-Nachrichten verwendet werden darf, wobei die HTTP-Bindung verwendet werden muss. Als Serialisierung sollte XML 1.0 nach [Bray 00] und die Zeichensatze UTF-8 oder UTF-16 vorgeschrieben werden.

5.2.6 SOAP-Intermediaries

Nachdem in den vorherigen Kapiteln 5.2.1 bis 5.2.5 Optionen des SOAP-Standards betrachtet wurden, soll hier diskutiert werden, ob der Präzisierungsstandard SOAP-Intermediaries und die mit ihnen verbundenen Teile verbieten sollte. Im SOAP-Standard sind SOAP-Intermediaries jedoch keine Option. Zwar kann SOAP auch ohne SOAP-Intermediaries verwendet werden. Auf den SOAP-Standard haben sie jedoch einen wichtigen Einfluss z. B. auf das SOAP-Processing-Model. In der Charter der XML Protocol Working Group [Fallside 02] werden sie als essentiell angesehen, um verteilte Systeme zu erstellen, die im Internet skalieren.

Auf der anderen Seite ist es vor allem das in [Ewald 01] beschriebene Workflow Camp, das SOAP-Intermediaries als gemeinsames Routingmodell benötigt. Für die einfacheren Szenarien im des Interop Camps sind SOAP-Intermediaries dagegen nicht notwendig. In dieser Arbeit werden die in Kapitel 2.4 beschriebenen Webkomponenten betrachtet, die ihre Dienste zur Verfügung stellen, so wie heute Webanwendungen ihre Dienste Benutzern zur Verfügung stellen. Dann ist davon auszugehen, dass eine direkte HTTP-Verbindung zwischen den Webkomponenten aufgebaut werden kann. Damit sind SOAP-Intermediaries für Nachrichtenaustausch nicht notwendig.

Wird auf SOAP-Intermediaries verzichtet, vereinfacht das Modell von SOAP. Der SOAP-Message-Path wird zu einer direkten Verbindung zwischen dem ursprünglichen Sender und dem endgültigen Empfänger. Dann macht es keinen Sinn mehr, über Rollen von SOAP-Knoten zu sprechen. Jeder SOAP-Empfänger ist immer der endgültige Empfänger. Jeder Headereintrag ist für ihn bestimmt. Daher kann das Role-Attribut von Headereinträgen und das Element `env:Role` in Fehlernachrichten entfallen. Ebenso entfällt in Fehlernachrichten auch das Element `env:Node`. Außerdem vereinfacht sich das SOAP-Processing-Model. Ein SOAP-Empfänger muss nicht mehr prüfen, welche Rollen er hat und nicht mehr untersuchen, welche Headereinträge für ihn bestimmt sind.

Trotzdem sind Headereinträge weiter sinnvoll. Sie erlauben es z. B., dass ein SOAP-Sender einer SOAP-Nachricht Daten zufügen kann, die nicht in der Beschreibung des Webservices spezifiziert sind und die der SOAP-Empfänger wenn erforderlich ignorieren kann. Besonders wichtig sind Headereinträge auch zusammen mit der RPC-Reprä-

sentation bzw. den Konventionen für Operationen, wie sie in Kapitel 5.2.4 beschrieben wurden. Sie erlauben es einer SOAP-Nachricht Daten zuzufügen, die nicht Parameter der Operation sind.

Da SOAP-Intermediaries einerseits für einfache Szenarien unnötig sind und andererseits die Komplexität von SOAP erhöhen, sollte ein Präzisierungsstandard sie verbieten. Damit kommunizieren stets der ursprüngliche SOAP-Sender und endgültige SOAP-Empfänger miteinander über eine direkte HTTP-Verbindung.

5.3 Präzisierung von WSDL

Für WSDL erscheint ein Präzisierungsstandard noch wichtiger als für SOAP. Die Möglichkeit beliebige Typsysteme zu verwenden macht es Werkzeugen unmöglich, WSDL-Dokumente zu interpretieren, die unbekannte Typsysteme enthalten. Auch dass es möglich ist, dieselbe Schnittstelle auf unterschiedliche Arten zu repräsentieren, erfordert offensichtlich eine Präzisierung von WSDL.

5.3.1 Verwendung von SOAP

In WSDL können Webservices beschrieben werden, die beliebige Protokolle verwenden. Für jedes Protokoll wird hierzu die als Bindung bezeichnete Erweiterung von WSDL benötigt. Drei Bindungen sind im zweiten Teil des WSDL-Standards beschrieben, vor allem die SOAP-Bindung.

Es ist offensichtlich, dass interoperable Webkomponenten das gleiche Protokoll verwenden müssen, was auch das Format ausgetauschter Nachrichten mit einschließt. In dieser Arbeit wurde im Kapitel 2 dargelegt, dass SOAP hierzu das geeignete, konsensfähige Protokoll ist. Es sollte daher in seiner aktuellen Version 1.2 von Webkomponenten verwendet werden, wobei zusätzlich die Anforderungen zur Präzisierung von SOAP aus dem Kapitel 5.2 berücksichtigt werden sollten.

Tatsächlich bilden die Anforderungen in diesem Kapitel zusammen mit denen aus Kapitel 5.2 Anforderungen für einen gemeinsamen Präzisierungsstandard für SOAP und WSDL. Der Präzisierungsstandard beschreibt also, wie SOAP zusammen mit WSDL für interoperable Webkomponenten verwendet werden sollte. Für die Anforderungen in diesem Kapitel bedeutet das umgekehrt, dass WSDL so weit präzisiert werden darf, dass nur die Verwendung von SOAP mit der Präzisierung aus Kapitel 5.2 beschrieben werden kann.

Damit findet die Kommunikation stets nach dem Request-/Response-Kommunikationsmuster statt (siehe Kapitel 5.2.1), so dass in WSDL nur Input-Output-Operationen (siehe Kapitel 4.3.5) beschrieben werden müssen. Dann hat in jeder abstrakten Schnittstelle jedes Element `wsdl:portType/wsdl:operation` als erstes Kindelement das Element `wsdl:input` und als zweites `wsdl:output`. Das gleiche gilt in gebundenen Schnittstellen für die Elemente `wsdl:binding/wsdl:operation`.

Das Request-/Response-Kommunikationsmuster muss verwendet werden, um die Operationen nach den Konventionen aus Kapitel 5.2.4 zu realisieren. Insbesondere muss sichergestellt werden, dass es in jeder SOAP-Nachricht genau einen Bodyeintrag gibt, der im Request den Namen der Operation hat und im Response die Konkatenation aus dem Namen der Operation und der Zeichenkette „`Response`". Der Bodyeintrag muss alle übertragenen Parameter enthalten, jedes als ein Kindelement.

Der Präzisierungsstandard sollte also klarstellen, dass WSDL nur im Zusammenhang mit SOAP und seiner Präzisierung aus dem Kapitel 5.2 verwendet werden darf. Insbesondere schränkt das WSDL auf die SOAP-Bindung mit der Serialisierung von XML 1.0 und der Protokollbindung für HTTP ein, sowie auf die Verwendung des Request-/Response-Kommunikationsmusters zusammen mit den Konventionen für Operationen aus Kapitel 5.2.4.

5.3.2 Typsystem

Wird SOAP verwendet, müssen die anwendungsspezifischen Teile der Nachrichten in WSDL beschrieben werden. Da WSDL grundsätzlich beliebige Datenformate für Nachrichten zulässt, muss es ermöglichen, Struktur und Datentypen beliebiger Datenformate zu beschreiben. Die dazu erlaubten beliebigen Typsysteme sind ein Grund, warum es unmöglich ist, dass ein Programm oder auch ein Entwickler jedes WSDL-Dokument interpretieren kann.

Daher wurde bereits in Kapitel 4.1 festgestellt, dass die Verwendung von XML-Schema für die Interoperabilität zweckmäßig ist. Es ist das einzige Typsystem für das im WSDL-Standard vorgestellt wird, wie es in WSDL integriert wird. In abstrakten Nachrichten können mit Attributen `wsdl:message/wsdl:part/@type` bzw. `wsdl:message/wsdl:part/@element` auf in XML-Schema definierte Typen oder deklarierte Elemente verwiesen werden. Für andere Typsysteme müssen weitere Attribute deklariert werden, im WSDL-Standard sind sie nicht beschrieben.

Auch die im vorherigen Kapitel 5.3.1 aufgestellte Anforderung, dass SOAP verwendet werden muss, spricht für die Verwendung von XML-Schema. SOAP-Nachrichten sind XML-Dokumente, die zu beschreibenden anwendungsspezifischen Teile XML-Fragmente. XML-Schema ist die Sprache, mit der typischerweise das Vokabular von XML-Fragmenten definiert wird.

XML-Schema bietet auch elementare Datentypen, die bei der Deklaration von Nachrichten verwendet werden können. Nur diese sollten verwendet werden. Der WSDL-Standard schreibt das jedoch nicht vor. Ebenso können Typen verwendet werden, die außerhalb von WSDL oder XML-Schema definiert worden sind, wobei auch eine rein informelle Beschreibung des Typs erlaubt ist. In WSDL werden sie durch ihren qualifizierten Namen identifiziert. Ebenso wie die Verwendung anderer Typsysteme macht es auch die Verwendung solcher Typen unmöglich, ein WSDL-Dokument vollständig zu interpretieren, wenn nur ein solcher Typ unbekannt ist. Daher sollten nur elementare Typen von XML-Schema verwendet werden sowie Typen, die mit Hilfe von XML-Schema definiert wurden.

Ein Präzisierungsstandard sollte daher also XML-Schema als einziges Typsystem in WSDL vorschreiben. Im WSDL-Dokument dürfen keine Typen verwendet werden, die nicht in XML-Schema definiert oder elementare Typen von XML-Schema sind. Die betreffenden Schemadokumente müssen im WSDL-Dokument enthalten sein oder importiert werden.

5.3.3 encoded versus literal

Auch bei Verwendung von XML-Schema als Typsystem gibt es in WSDL zwei grundsätzlich unterschiedliche Techniken, die Struktur und die Typen anwendungsspezifischer Teile ausgetauschter Nachrichten zu beschreiben. Welche Technik verwendet wird, lässt sich für einzelne Nachrichten in der gebundenen Schnittstelle mit dem Use-Attribut (siehe Kapitel 4.3.7.2) steuern. Die erlaubten Werte sind `encoded` und `literal`.

Ein Präzisierungsstandard sollte nur eine der beiden Techniken erlauben. Ansonsten könnte es Implementierungen geben, die nur eine unterstützen, was für die Interoperabilität ein Problem ist. Zumindest würde es Implementierungen vereinfachen, wenn nur eine Technik implementiert werden muss. Daher soll im Folgenden diskutiert werden, welche von beiden für die Interoperabilität zweckmäßiger ist.

Beide Techniken wurden bereits in Kapitel 4.3.7.2 vorgestellt und sollen hier für die nachfolgende Diskussion kurz wiederholt werden. Hat das Use-Attribut für den SOAP-Body, SOAP-Header oder eine Fehlernachricht den Wert `encoded`, wird damit festgelegt, dass der Aufbau dieses Teils mit Hilfe einer Kodierung definiert wird. Ausgegangen wird von einer Datenstruktur der Implementierungssprache (oder dem SOAP-Datenmodell). Diese wird in WSDL mit Hilfe von XML-Schema oder einem anderen Typsystem beschrieben. Die Transformationsregeln der Kodierung beschreiben wie die Datenstruktur in ein XML-Fragment transformiert wird. WSDL erlaubt beliebige Kodierungen, die mit URIs identifiziert werden. Die URI wird im Encodingstyle-Attribut im Element `soap12:body`, `soap12:header` oder `soap12:fault` angegeben.

Hat das Use-Attribut dagegen den Wert `literal`, wird der Inhalt von SOAP-Body, SOAP-Header oder dem Detail-Element der Fehlernachricht direkt in XML-Schema beschrieben. Es wird also auf den Umweg über die Beschreibung einer Datenstruktur der Implementierungssprache bzw. des SOAP-Datenmodells verzichtet und statt dessen direkt beschrieben, welche Struktur und Typen das betreffende XML-Fragment hat.

Schon die kürzere Darstellung macht offensichtlich, dass die Verwendung von `literal` die einfachere, direktere Technik ist, Nachrichten zu beschreiben. Die aufwendigere Technik von `encoded`, erst Datenstrukturen in der Implementierungssprache zu beschreiben und danach in ein XML-Fragment zu transformieren, hat den Vorteil, dass Beschreibungen der Datenstrukturen in der Implementierungssprache oft vorliegen.

Die Verwendung von `encoded` lässt jedoch mehr Raum für unterschiedliche Interpretationen der WSDL-Beschreibung, die zu Interoperabilitätsproblemen führen können. Ursachen für unterschiedliche Interpretationen bei `encoded` ist vor allem auch die Tatsache, dass Datenstrukturen in den Implementierungssprachen zumindest im Detail stark unterschiedlich ausgedrückt werden, wie es für objektorientierte Modelle in Kapitel 2.3.2 ausgeführt wurde.

Hinzu kommt, dass die Transformationsregeln einer Kodierung von der Art der Datenstrukturen ausgehen müssen. Daher sind prinzipiell für unterschiedliche Implementierungssprachen auch unterschiedliche Kodierungen erforderlich. Sie zu verwenden ist in WSDL erlaubt. Ein WSDL-Dokument kann jedoch nur interpretiert werden, wenn alle verwenden Kodierungen bekannt sind. Daher ist es für die Interoperabilität notwendig, nur eine oder wenige Kodierungen zuzulassen. Der WSDL-Standard standardisiert keine. Der SOAP-Standard enthält das in Kapitel 3.7 vorgestellte SOAP-Encoding.

Damit das SOAP-Encoding von den Möglichkeiten zur Datenstrukturierung von Implementierungssprachen unabhängig wird, beschreibt es seine Transformationsregeln ausgehend vom abstrakten SOAP-Datenmodell. Das ermöglicht eine breite Verwendung des SOAP-Encodings. Gleichzeitig hat das den Nachteil, dass Datenstrukturen der Implementierungssprache zunächst auf das SOAP-Datenmodell abgebildet werden müssen, was nicht standardisiert ist.

Für die Beschreibung der XML-Fragmente spielt dieser Nachteil letztlich keine Rolle, weil es für die Kenntnis seiner Struktur und Typen unwichtig ist, wie sie in der Implementierungssprache kodiert waren. Es erscheint jedoch unnötig aufwendig, ein

XML-Fragment zu beschreiben, indem zunächst mit XML-Schema als Typsystem eine Datenstruktur im SOAP-Datenmodell beschrieben wird, die dann mit Hilfe des SOAP-Encodings in ein XML-Fragment transformiert wird. Das wäre jedoch bei der Verwendung von `encoded` der Fall. Viel einfacher ist die direkte Beschreibung des XML-Fragmentes mit XML-Schema bei Verwendung von `literal`.

Unabhängig von der Frage, wie aufwendig die Beschreibung mit beiden Technik ist, lässt sich auch grundsätzlicher fragen, ob Schnittstellen auf Basis von Datenstrukturen aus Implementierungssprachen beschrieben werden sollten oder ob eine direkte Beschreibung der ausgetauschten Nachrichten nicht besser wäre. In der Einleitung des SOAP-Standards [Gudgin 02a] wird festgestellt, dass SOAP unabhängig von einem bestimmten Programmiermodell oder anderer implementierungsspezifischen Semantik entwickelt wurde. Das sollte entsprechend auch für WSDL gelten. Ein Ausgehen von Datenstrukturen aus Implementierungssprachen ist jedoch eine solche implementierungsspezifische Semantik.

Tim Ewald vertrat in seinem Vortrag zu [Ewald 01] darüber hinaus die These, dass bei der Beschreibung von Webservices nur das berücksichtigt werden sollte, was „auf der Leitung“ zu beobachtet ist. Alles andere hält für Fiktion. „Auf der Leitung“ können die Nachrichten beobachtet werden, also mit den Annahmen aus Kapitel 5.3.1 SOAP-Nachrichten. Die These von Tim Ewald würde bedeuten, dass die SOAP-Nachrichten direkt beschrieben werden müssen, unabhängig von „auf der Leitung“ nicht beobachtbaren Datenstrukturen der Implementierungssprache oder des SOAP-Datenmodells.

Liest man den WSDL-Standard genau, legt auch er nahe, SOAP-Nachrichten direkt mit XML-Schema zu beschreiben. Dort wird festgestellt: „*Also wird es empfohlen, dass wenn immer möglich das XML-Schema-Typsystem verwendet wird, um zwischen Webservices ausgetauschte Nachrichten zu beschreiben.*“[27] In diesem Satz wird also nicht nur nahegelegt, in WSDL XML-Schema als Typsystem zu verwenden. Darüber hinaus sollen die ausgetauschten Nachrichten selbst mit XML-Schema beschrieben werden.

Alle diese Argumente sprechen für die Verwendung von `literal`, dass also anwendungsspezifische Teile von Nachrichten direkt in XML-Schema deklariert werden sollten. Die in Kapitel 5.2.4 geforderten Konventionen zur Repräsentation von Operationen müssen dabei berücksichtigt werden. Das schließt z. B. Mechanismen ein, mit denen Datenelemente (Knoten), nur einmal im XML-Fragment gespeichert werden, wie es in Kapitel 3.7 vorgestellt wurde. Möglich ist das mit den Typen `xsd:ID` und `xsd:IDREF` von XML-Schema. Tim Ewald weist jedoch in [Ewald 01] für SOAP 1.1 und WSDL 1.1 darauf hin, dass eine exakte Definition des SOAP-Encodings von SOAP 1.1 in XML-Schema nicht vollständig möglich ist. Das gilt ebenso für das SOAP-Encoding von SOAP 1.2, worauf noch in Kapitel 6.2.2 eingegangen wird. Trotzdem lassen sich XML-Nachrichten weitgehend äquivalent in XML-Schema beschreiben.

Das Encodingstyle-Attribut, in dem bei `encoded` die URI der verwendeten Kodierung angegeben wird, darf auch bei `literal` verwendet werden. Dann verliert es jedoch seine wichtige Rolle und ist nur noch ein Hinweis, wie die in XML-Schema beschriebene Struktur zustande gekommen ist. Da das Attribut nicht immer vorhanden sein muss, darf ein Programm, das ein WSDL-Dokument liest, jedoch nicht von Existenz und Wert des Encodingstyle-Attributes abhängen.

---

[27] Übersetzung aus [Chinnici 02], Kapitel 2.1. Englischer Originaltext: „*Thus, it is RECOMMENDED that the XML Schema type system be used to describe messages exchanged between Web services whenever possible.*“

Es lässt sich zusammenfassen, dass ein Präzisierungsstandard fordern sollte, dass das Use-Attribut stets den Wert `literal` haben muss, so dass die anwendungsspezifischen XML-Fragmente direkt in XML-Schema beschrieben werden. Die Angabe des Encodingstyle-Attributes in WSDL als Hinweis sollte erlaubt werden. Programme, die WSDL-Dokumente lesen, dürfen jedoch nicht von seiner Existenz abhängig sein.

5.3.4 rpc versus document

Eine weitere Option bei der Beschreibung eines Webservices in WSDL ist die Wahl des Style-Attributes. Es wurde bereits in Kapitel 4.3.7.1 vorgestellt und soll hier für die nachfolgende Diskussion kurz wiederholt werden. Das Style-Attribut kann einen der beiden Werte `rpc` und `document` annehmen. In der SOAP-Nachricht bestimmt es, ob ein zusätzliches Element erzeugt wird, das den Namen der Operation hat und in dem die einzelnen Teile der Nachricht als Kindelemente enthalten sind. Bei `rpc` ist das der Fall. Für den SOAP-Body handelt es sich bei dem zu erzeugenden Element um den Bodyeintrag. Bei `document` wird ein solches Element jedoch nicht erzeugt und die Kindelemente direkt im übergeordneten Element (z. B. dem SOAP-Body) gespeichert.

Die Frage, ob ein Element wie der Bodyeintrag bei `rpc` automatisch erzeugt wird oder bei `document` nicht, sollte nicht allzu wichtig sein. Zwar wird nach den Konventionen für Operationen aus Kapitel 5.2.4 ein solcher Bodyeintrag benötigt. Wird `document` verwendet, kann er jedoch explizit in XML-Schema deklariert werden. Ein Präzisierungsstandard sollte der Einfachheit halber nur eine der beiden Optionen erlauben. Welche für die Interoperabilität zweckmäßiger ist, soll nachfolgend diskutiert werden.

Auf ersten Blick erscheint die Wahl von `rpc` naheliegend. Schon der Name legt sie nahe, sollen doch Operationen repräsentiert werden, die mit den Konventionen aus Kapitel 5.2.4 geeignet sind RPCs zu realisieren. Auch wird bei dieser Wahl der dazu benötigte Bodyeintrag automatisch erzeugt, ohne dass er explizit deklariert werden müsste.

Tatsächlich ist eine Wahl von `rpc` jedoch nicht in jedem Fall möglich. Der WSDL-Standard stellt sowohl für Headereinträge, also auch für Detaileinträge von Fehlernachrichten fest, dass für sie immer `document` angenommen wird. Bei der Wahl von `rpc` würde also der Bodyeintrag automatisch erzeugt werden, während Header- und Detaileinträge in jedem Fall explizit deklariert werden müssten.

Ein noch wichtigeres Argument liefert Tim Ewald in seinem Vortrag zu [Ewald 01]. Er stellt fest, dass es nur zwei sinnvolle Kombinationen von Use- und Style-Attribut gibt. Der Wert `encoded` sollte nur zusammen mit `rpc` verwendet werden. Der Wert `literal` nur zusammen mit `document`. Zum Zeitpunkt seines Vortrages unterstützten die meisten Programme, die WSDL erzeugen oder lesen die Kombination `encoded` zusammen mit `rpc`. Mittlerweile wird jedoch vermehrt auch die Kombination `literal` zusammen mit `document` unterstützt. Wichtige Vertreter sind die meisten mit Microsoft's .NET implementierten Webservices.

Die enge Verwandtschaft zwischen `encoded` mit `rpc` bzw. `literal` mit `document` hat dazu geführt, dass viele Beobachter die Diskussionen über das Use- und das Style-Attribut als eine einzige Diskussion betrachten. Sie bezeichnen die Diskussion mit `rpc` versus `document`. So werden z. B. in [McCarthy 02] die Vorteile des „document style" vorgestellt, die jedoch primär auf der Tatsache beruhen, dass mit `literal` XML-Fragmente direkt in XML-Schema beschrieben werden. Das wäre jedoch prinzipiell auch mit `rpc` möglich. Analog dazu wird auf der Website von XMethods [Hong 02], die

vor allem eine Liste von Webservices enthält, für jeden Webservice nur sein „Style“ mit den möglichen Werten „RPC“ und „DOC“ (entspricht `document`) angegeben.

Trotz der Verwandtschaft von Use- und Style-Attribut wurden in dieser Arbeit beide zunächst getrennt betrachtet, weil der WSDL-Standard prinzipiell beliebige Kombinationen zulässt. Da Programme die WSDL interpretieren aber häufig nur die beiden Kombinationen `encoded` mit `rpc` oder `literal` mit `document` unterstützen, sollten für die Interoperabilität auch nur diese beiden Kombinationen verwendet werden. Mit der Wahl des Wertes `literal` für das Use-Attribut in Kapitel 5.3.3 folgt, dass für das Style-Attribut der Wert `document` gewählt werden muss. Diese Entscheidung passt auch zum vorgenannten Argument, dass nur mit der Wahl von `document` die Body-, Header- und Detaileinträge auf die gleich Art deklariert werden können. Daher sollte für das Style-Attribut stets dieser Wert gewählt werden.

Für die korrekte Wahl der Namen von Bodyeinträgen muss beachtet werden, dass sie qualifiziert sein müssen. Der Name der Operation kann nur der lokale Name des Elementes sein. Dieser muss sich jedoch, wie in Kapitel 3.2 beschrieben, in einem Namensraum befinden. Dessen URI wird bei Verwendung von `rpc` explizit im Namespace-Attribut angegeben. Bei `document` muss dagegen der Namensraum nicht explizit angegeben werden. Der Bodyeintrag (ebenso wie Header- und Detaileinträge) wird in XML-Schema deklariert. Dort wird auch sein Namensraum festgelegt.

Der WSDL Standard verbietet auch bei `document` nicht, dass das Namespace-Attribut angegeben wird. In [Long 01] wird jedoch darauf hingewiesen, dass beim Leser eines WSDL-Dokumentes Unklarheit darüber aufkommen kann, ob der mit XML-Schema definierte Namensraum oder der im Namespace-Attribut angegebene verwendet werden soll. Daher wird dort vorgeschlagen, dass entweder das Namespace-Attribut die URI des Namensraumes enthalten muss, den der Bodyeintrag hat oder das Namespace-Attribut entfallen soll, wenn der Bodyeintrag ohnehin explizit deklariert ist. Da letzteres bei `document` stets der Fall ist, sollte der Einfachheit halber immer auf die Verwendung des Namespace-Attributes verzichtet werden.

Die Konvention, dass Bodyeinträge den Namen der Operation haben, löst ein in SOAP vorhandenes Problem, auf das Tim Ewald in [Ewald 01] für SOAP 1.1 hinweist. Das Problem besteht auch in SOAP 1.2. Für einen Webservice gibt es keinen Standardweg aus dem Request abzuleiten, welche Operation aufgerufen wird. Tim Ewald sieht unterschiedliche Wege das Problem zu lösen. Mit der RPC-Repräsentation ist die offensichtliche Lösung, aus dem Namen des Bodyeintrages die Operation abzuleiten.

Das setzt jedoch voraus, dass der qualifizierte Name des Bodyeintrages eindeutig eine einzige Operation bezeichnet. Somit muss das Überladen von Operationen verboten werden, bei dem mehrere Operationen den gleichen Namen haben und durch ihre unterschiedlichen Parameter unterschieden werden. Der aktuelle Arbeitsentwurf des WSDL-Standards [Chinnici 02] verbietet das Überladen von Operationen. Es wird jedoch in einer „Editorial Note“ angegeben, dass hierüber noch kein Konsens besteht. Sollte der endgültige WSDL-Standard das Überladen von Operationen zulassen, muss es aus den oben genannten Gründen im Präzisierungsstandard verboten werden.

Der Präzisierungsstandard sollte also für das Style-Attribut festlegen, dass es stets den Wert `document` haben muss. Das Namespace-Attribut wird dann nicht mehr benötigt und darf nicht mehr angegeben werden. Das Überladen von Operationen muss verboten werden, um die Abbildung des Namens des Bodyeintrages im Request auf die Operation auf einfache Art zu ermöglichen.

5.3.5 Type- versus Element-Attribut

In Kapitel 5.3.2 wurde festgelegt, dass XML-Schema als Typsystem verwendet werden muss. In Deklarationen von Nachrichten kann mit den Attributen `wsdl:message/wsdl:part/@type` (Type-Attribut) und `wsdl:message/wsdl:part/@element` (Element-Attribut) auf XML-Schema verwiesen werden. Beide Attribute unterscheiden sich darin, worauf verwiesen wird. Mit einem Type-Attribut wird auf einen in XML-Schema definierten Typen verwiesen, während mit dem Element-Attribut auf ein in XML-Schema deklariertes Element verwiesen wird.

Nachfolgend soll angenommen werden, dass für das Use-Attribut der Wert `literal` und für das Style-Attribut der Wert `document` verwendet wird, wie in den Kapiteln 5.3.3 und 5.3.4 festgelegt wurde. Dann wird der Typ, auf den mit dem Type-Attribut verwiesen wird, in der SOAP-Nachricht zum Typ des Body-, Header- bzw. Detail-Elementes einer Fehlernachricht. Ein Element-Attribut verweist dagegen auf die Deklaration des Elementes, das zum Kindelement der eben genannten Elemente wird. Es werden also direkt Body-, Header- oder Detaileinträge deklariert. Mit Hilfe mehrerer Parts in einer Nachrichtendeklaration können mehrere solcher Kindelemente zugefügt werden.

In der Regel ist es möglich die gleiche SOAP-Nachricht mit dem Type- oder Element-Attribut zu spezifizieren. Es handelt sich also um alternative Option, was nach Kapitel 5.3.3 zu Interoperabilitätsproblemen führen kann. So wird auch im Arbeitsentwurf des WSDL-Standards [Chinnici 02] die Frage aufgeworfen, ob es nicht ausreicht, nur das Element- oder das Type-Attribut zu unterstützen. Ein Präzisierungsstandard sollte diese Entscheidung treffen. Daher werden diese Attribute im Folgenden für SOAP-Body, SOAP-Header und das Detail-Element von Fehlernachrichten diskutiert.

Mit den in Kapitel 5.2.4 geforderten Konventionen für Operationen enthält der SOAP-Body stets genau einen Bodyeintrag, der in WSDL beschrieben werden muss. Mit dem Element-Attribut kann direkt auf eine in XML-Schema formulierte Beschreibung des Bodyeintrages verwiesen werden. Mit dem Type-Attribut müsste dagegen in XML-Schema ein komplexer Typ definiert werden, der als einziges Kindelement den Bodyeintrag hat. Letzteres ist aufwendiger und birgt daher eher die Gefahr, dass der Inhalt des SOAP-Bodys nicht nach den Konventionen aus Kapitel 5.2.4 spezifiziert wird. So ist für die Deklaration des SOAP-Bodys die Verwendung des Element-Attributes vorzuziehen.

Die Beschreibung des SOAP-Headers unterscheidet sich von der des SOAP-Bodys darin, dass er mehrere Kindelemente (Headereinträge) haben darf. Außerdem legt der WSDL-Standard ausdrücklich fest, dass es nicht notwendig ist, alle Headereinträge zu spezifizieren. Diese Festlegung ist notwendig, weil ein SOAP-Sender ohne vorherige Absprachen einer SOAP-Nachricht nicht spezifizierte Headereinträge zufügen darf. Der SOAP-Empfänger darf einen Headereintrag ignorieren, es sei denn er ist explizit als zwingender Headereintrag gekennzeichnet. (Siehe Kapitel 3.3.) Diese Überlegungen zeigen auch, dass Headereinträge voneinander eine relative Unabhängigkeit haben und nicht zusammen deklariert werden sollten.

Würde zur Deklaration von Headereinträgen nun das Type-Attribut verwendet, dann würde damit der Typ des SOAP-Headers angegeben werden. Dieser Typ müsste alle Deklarationen von erlaubten Headereinträgen enthalten. Mit Hilfe von `xsd:any` müsste spezifiziert werden, dass weitere Headereinträge erlaubt sind. Die gemeinsame Deklaration der Headereinträge widerspricht der ihrer Unabhängigkeit. Außerdem ist es so

möglich, in der Deklaration des SOAP-Headers auf die Verwendung von `xsd:any` zu verzichten, so dass der SOAP-Sender keine weiteren Headereinträge zufügen darf, was dem SOAP-Standard widerspricht.

Wird hingegen das Element-Attribut verwendet, wird jeder Headereintrag einzeln als ein Part der Nachricht und in XML-Schema als ein Element spezifiziert. Damit wird die Unabhängigkeit der Headereinträge viel klarer. Ebenso widerspricht es nicht der Tatsache, dass weitere Headereinträge zugefügt werden dürfen. Daher ist auch für die Deklaration des SOAP-Headers die Verwendung das Element-Attributes vorzuziehen.

Noch anders verhält es sich bei der Beschreibung des Detail-Elementes, das in Fehlernachrichten die anwendungsspezifischen Teile enthält. Hier ist der Entwickler einer Anwendung frei, den Inhalt des Detail-Elementes nach seinen Vorstellungen zu verwenden. Mit dem Type-Attribut könnte dem Detail-Element ein in XML-Schema definierter Typ zugeordnet werden, was dem Entwickler die maximale Flexibilität geben würde.

Mit dem Element-Attribut würden die Detail-Einträge deklariert werden, wobei für jeden Detail-Eintrag ein Part in der Nachrichtendeklaration benötigt wird. Die Verwendung des Element-Attributes schränkt den Entwickler also stärker darin ein, wie er die Struktur der Nachfahren des Detail-Elementes spezifizieren muss. Trotzdem bleibt es aber möglich, den Fehler im Detail-Element anwendungsspezifisch zu beschreiben.

Für die Beschreibung des Detail-Elementes wäre also die Verwendung des Type-Attributes attraktiver. Dem gegenüber stehen die Vorteile des Element-Attributes bei der Beschreibung von SOAP-Body und SOAP-Header. Letztere Vorteile überwiegen, schon weil auch mit der Verwendung des Element-Attributes genügend Flexibilität bleibt, den Inhalt von Detail-Elementen zu beschreiben. Daher sollte ein Präzisierungsstandard die Verwendung des Element-Attributes vorschreiben und die des Type-Attributes verbieten.

5.3.6 Verwendung von wsdl:import

Auch `wsdl:import` sollte im Präzisierungsstandard betrachtet werden. Wie bereits in Kapitel 4.3.2 beschrieben, erlaubt es `wsdl:import`, außerhalb des WSDL-Dokumentes beschriebene Namensräume zu importieren, so dass enthaltene Namen im WSDL-Dokument verwendet werden können. Dazu hat `wsdl:import` die Attribute `namespace` und `location`. Mit ersterem wird die URI des zu importierenden Namensraumes angegeben; in letzterem, das auch als Location-Attribut bezeichnet wird, der Ort an dem sich eine Beschreibung des Namensraumes befindet.

Nicht beschrieben wird im WSDL-Standard jedoch, in welcher Form diese Informationen beschrieben sein müssen. Offensichtlich ist, dass WSDL-Dokumente hierzu erlaubt sein müssen, weil das Ziel von `wsdl:import` ist, WSDL-Beschreibungen auf mehrere WSDL-Dokumente aufteilen zu können. Der WSDL-Standard verbietet es jedoch nicht, auf beliebige andere Arten von Beschreibungsdokumenten zu verweisen, mit denen Namensräume definiert werden können. Hierzu gehören zumindest XML-Schemadokumente.

Für die Interoperabilität ist es wichtig, dass nicht auf beliebige Arten von Beschreibungsdokumenten verwiesen werden kann. Kein Programm wäre sonst in der Lage, alle Arten zu interpretieren. Damit wird auch die Interpretation des WSDL-Dokumentes unmöglich, das einen solchen Import enthält. Ein Präzisierungsstandard sollte daher die erlaubten Arten von Beschreibungsdokumenten festlegen.

Dass die Beschreibung von `wsdl:import` im Standard von WSDL 1.1 [Christensen 01], präzisiert werden muss, wird auch in den Anforderungen zu WSDL 1.2 [Schlimmer 02]

festgestellt. Im Standard von WSDL 1.1 wird allerdings der Import von beliebigen Arten von Beschreibungsdokumenten explizit erlaubt. Der aktuelle Arbeitsentwurf von WSDL 1.2 beschreibt diesen Aspekt noch ungenauer, indem auf Arten von Beschreibungsdokumenten gar nicht eingegangen wird. Die Anforderung R006 weist auch darauf hin, dass die Beziehung zum `xsd:import` von XML-Schema geklärt werden muss.

In XML-Schema wird `xsd:import` für den gleichen Zweck verwendet, wie `wsdl:import` in WSDL. Mit ihm werden Namensräume in ein XML-Schemadokument importiert, so dass die im Namensraum enthaltenen Namen verwendet werden können. Das Element `xsd:import` enthält auch das Location-Attribut, das dort den Namen `schemaLocation` hat und optional ist. Anders als bei `wsdl:import` wird für das Location-Attribut von `xsd:import` im Standard von XML-Schema [Thompson 01] ausdrücklich festgelegt, dass es auf eine Serialisierung eines XML-Schemadokumentes verweisen muss.

Auch innerhalb von WDSL kann `xsd:import` innerhalb von eingebetteten XML-Schemadokumenten verwendet werden. Zum Import von XML-Schemadokumenten würden also zwei Optionen zur Verfügung stehen, wenn erlaubt werden würde, sie auch mit `wsdl:import` zu importieren.

Beide Optionen unterscheiden sich jedoch darin, in welchem Teil des WSDL-Dokumentes der Import sichtbar ist. Mit `xsd:import` ist der Import nur innerhalb der Typbeschreibung sichtbar. Zumindest sagt der WSDL-Standard nichts darüber aus, dass ein solcher Import außerhalb der Typbeschreibung im übrigen WSDL-Dokument sichtbar ist. Aus Interoperabilitätsgründen sollte es daher auch nicht angenommen werden. Umgekehrt ist es bei `wsdl:import`. Hier gilt der Import für das gesamte WSDL-Dokument. Die Typbeschreibung in WSDL ist jedoch eine in XML-Schema beschriebene, abgeschlossene Einheit. Daher ist nicht klar, ob der Import auch innerhalb der Typbeschreibung gültig ist. Damit ist also ein Import mit `wsdl:import` nur außerhalb der Typbeschreibung und ein Import mit `xsd:import` nur innerhalb der Typbeschreibung gültig.

Für die Frage, ob eine der beiden Optionen unnötig ist und daher im Präzisierungsstandard verboten werden sollte, ist es also wichtig zu wissen, wo ein importiertes XML-Schemadokument benötigt wird. Der Import von XML-Schemadokumenten wird dann notwendig, wenn Typen, Elemente oder Attribute zunächst bei der Anwendungsentwicklung unabhängig von SOAP und WSDL in eigenen XML-Schemadokumenten spezifiziert und in SOAP-Nachrichten verwendet werden sollen. Dann werden diese XML-Schemadokumente importiert und ihre Typen, Elemente oder Attribute in WSDL verwendet.

Die Typen, Elemente oder Attribute können in der Typbeschreibung innerhalb von WSDL verwendet werden, um neue Typen, Elemente oder Attribute zu definieren. Wird z. B. der Typ eines Parameters einer Operation in einem XML-Schemadokument beschrieben, würde er bei der Deklaration des Bodyeintrages verwendet.

Außerhalb der Typbeschreibung könnten importierte Typen oder Elemente direkt bei der Nachrichtendeklaration im Type- oder Element-Attribut angegeben werden. Nach Kapitel 5.3.5 ist jedoch nur das Element-Attribut erlaubt. Mit der Annahme aus Kapitel 5.3.4, dass für das Style-Attribut der Wert `document` verwendet wird, folgt dann, dass im XML-Schemadokument ein Body-, Header- oder Detail-Eintrag deklariert werden muss. Solche sind jedoch SOAP-spezifisch, so dass es untypisch wäre, diese in anwendungsspezifischen XML-Schemadokumenten bereits vorzufinden.

Somit ist es nicht notwendig, importierte XML-Schemadokumente direkt bei der Nachrichtendeklaration und damit außerhalb von Typbeschreibungen zu verwenden. Da der Import von XML-Schemadokumenten innerhalb der Typbeschreibung mit `xsd:import` erfolgen kann, gibt es keine Notwendigkeit den Import von XML-Schemadokumenten mit `wsdl:import` zu erlauben.

Der Präzisierungsstandard sollte daher festlegen, dass `wsdl:import` nur zum Importieren von WSDL-Dokumenten verwendet werden darf. Dokumente in XML-Schema können mit `xsd:import` innerhalb der in XML-Schema formulierten Typbeschreibung in WSDL importiert werden. So importierte Namensräume dürfen jedoch nur innerhalb der importierenden Typbeschreibung verwendet werden.

5.4 Zusammenfassung

Die in diesem Kapitel 5 genannten Anforderungen sollen als Basis für einen Präzisierungsstandard für SOAP zusammen mit WSDL dienen. Sie werden noch einmal in Tabelle 2 und Tabelle 3 zusammengefasst. Mit ihnen lassen sich die breiten Standards von SOAP und WSDL so weit präzisieren, dass die Interoperabilität zwischen Webkomponenten sichergestellt werden kann, wenn sich die zu ihrer Implementierung verwendeten Werkzeuge an die Präzisierung halten. Diese Präzisierung von SOAP und WSDL soll auch für nachfolgende Untersuchungen in dieser Arbeit angenommen werden.

Beschrieben in Kapitel	Anforderung
5.2.1	Es muss das Request-/Response-Kommunikationsmuster verwendet werden.
5.2.2	Der SOAP-Empfänger darf nicht auf das Vorhandensein des Encodingstyle-Attributes in der SOAP-Nachricht oder dessen Wert angewiesen sein. Er sollte es möglichst ignorieren.
5.2.3	Das Attribut `xsi:type` darf in SOAP-Nachrichten nicht verwendet werden. Statt dessen werden Typen von Elementen in der Schnittstellenbeschreibung in WSDL spezifiziert.
5.2.4	Operationen müssen nach den Konventionen aus Kapitel *5.2.4* realisiert werden. Insbesondere enthält der SOAP-Body stets genau einen Bodyeintrag, dessen Name sich aus dem Name der Operation ergibt. Die Parameter der Operation werden jeweils mit ihren Nachfahren zu einem Kindelement des Bodyeintrages.
5.2.4	Alle (In- und Inout- bzw. Out- und Inout-) Parameter der Operation müssen in der SOAP-Nachricht im Bodyeintrag enthalten sein.
5.2.4	Das Element `rpc:result` darf nicht verwendet werden.
5.2.5	Es muss die HTTP-Bindung und damit HTTP zum Austausch von SOAP-Nachrichten verwendet werden.
5.2.5	Es muss die Serialisierung XML 1.0 verwendet werden sowie einer der beiden Zeichensätze UTF-8 oder UTF-16.
5.2.6	SOAP-Intermediaries dürfen nicht verwendet werden. Daher entfallen das Role-Attribut in Headereinträgen sowie die Elemente `env:Role` und `env:Node` in Fehlernachrichten.

Tabelle 2: Anforderungen für einen Präzisierungsstandard für SOAP

Beschrieben in Kapitel	Anforderung
5.3.1	WSDL muss zur Beschreibung von Webservices verwendet werden, die über SOAP kommunizieren, wobei die in dieser Tabelle zusammengefassten Anforderungen beachtet werden.
5.3.2	Als Typsystem muss XML-Schema verwendet werden. Es dürfen keine Typen verwendet werden, die nicht in XML-Schema definiert oder elementare Typen von XML-Schema sind.
5.3.2	Verwendete Schemadokumente müssen im WSDL-Dokument enthalten sein oder in dieses importiert werden.
5.3.3	Das Use-Attribut muss stets den Wert `literal` haben.
5.3.3	Das Encodingstyle-Attribut in WDSL darf nur als Hinweis verwendet werden. Programme, die WSDL-Dokumente lesen, dürfen nicht von ihm abhängen.
5.3.4	Das Style-Attribut muss stets den Wert `document` haben.
5.3.4	Das Namespace-Attribut darf nicht angegeben werden.
5.3.4	Operationen dürfen nicht überladen werden.
5.3.5	Das Type-Attribut darf nicht verwendet werden.
5.3.6	Mit `wsdl:import` dürfen nur WSDL-Dokumente importiert werden.
5.3.6	In XML-Schema formulierte Typbeschreibungen können mit `xsd:import` innerhalb einer Typbeschreibung importiert werden. Der Import ist dann nur innerhalb der importierenden Typbeschreibung gültig.

Tabelle 3: Anforderungen für einen Präzisierungsstandard für WSDL

6 Ausdrucksfähigkeit von WSDL

Der in Kapitel 5 vorgeschlagene Präzisierungsstandard ermöglicht, dass Entwickler WSDL auf die gleiche Art verwenden, weil unnötige Optionen verboten wurden. Das erlaubt es Werkzeugen ebenso wie Entwicklern, die WSDL-Dokumente anderer Entwickler zu verstehen.

Interoperabilitätsprobleme können sich jedoch nicht nur aus einem falschen Verständnis von WSDL-Dokumenten ergeben. Auch wenn die enthaltene Spezifikation unpräzise ist, kann es geschehen, dass Webclient und Webservice nicht miteinander interoperabel sind, obwohl sie die Spezifikation der Schnittstellen im WSDL-Dokument korrekt implementieren.

Daher soll in diesem Kapitel untersucht werden, wie präzise sich ein Webservice in WSDL beschreiben lässt. Zunächst wird kurz zusammengefasst, was in WSDL ausgedrückt werden kann. Kapitel 6.1 zeigt danach unterschiedliche Anforderungen, die sich WSDL nicht ausdrücken lassen. Schließlich werden in Kapitel 6.2 Grenzfälle analysiert, um die Grenzen der Ausdrucksfähigkeit von WSDL auszuloten.

Mit WSDL und dem in Kapitel 5 vorgeschlagenen Präzisierungsstandard kann einerseits ausgedrückt werden, unter welchen Netzwerkadressen ein Webservice erreicht werden kann und andererseits über welche Schnittstellen. Die Schnittstellen sind Mengen von Operationen, zwischen denen keine Beziehungen festgelegt werden können. Eine Operation wird im Wesentlichen durch zwei Nachrichtentypen beschrieben, wobei der grundlegende Aufbau der SOAP-Nachricht implizit festliegt und enthaltene anwendungsspezifische XML-Fragmente (Body-, Header- oder Detaileinträge) in XML-Schema spezifiziert sind.

Mit XML-Schema kann die Struktur dieser anwendungsspezifischen XML-Fragmente sowie die möglichen Werte von Attributen und Blattelementen festgelegt werden. Für letzteres gibt es flexible Verfahren zur Definition von einfachen Typen, mit denen sogar die erlaubten Werte mit Hilfe regulärer Ausdrücke angegeben werden können.

6.1 Was kann nicht in WSDL ausgedrückt werden?

Was WSDL jedoch nicht erlaubt, ist über diese Möglichkeiten hinausgehende Anforderungen zu formulieren. So ist es z. B. nicht möglich Beziehungen zwischen Werten in verschiedenen Attributen oder Blattelementen festzulegen. Nur für einzelne Werte kann ein Typ angegeben werden. Es können weder Werte innerhalb eines Body-, Header- oder Detaileintrages miteinander in Beziehung gesetzt werden, noch zwischen solchen innerhalb einer SOAP-Nachricht.

Noch weniger ist es in WSDL möglich, explizit Beziehungen zwischen dem Request und Response eines Operationsaufrufes oder zwischen SOAP-Nachrichten unterschiedlicher Operationsaufrufe auszudrücken. Für unterschiedliche Operationen kann lediglich festgelegt werden, welche Operationen zusammen eine Schnittstelle bilden. Nicht festgelegt werden kann z. B., in welcher Reihenfolge Operationen aufgerufen werden müssen. Auch lassen sich keine Beziehungen ausdrücken, die zwischen Parametern von Operationen gelten müssen.

Anforderungen, wie die eben angesprochenen, können jedoch für die Entwicklung von Webclients wichtig sein. Einige der Anforderungen muss der Webclient selbst erfüllen. Z. B. muss er Reihenfolgen einhalten, in der Operationen aufgerufen werden müssen. Auch bei Anforderungen, die der Webservice erfüllen muss, ist es wichtig, dass der Webclient mit dessen Verhalten rechnet.

Damit sind solche Anforderungen wichtiger Teil des Vertrages zwischen Webclient und Webservice und müssten auch als Teil der Schnittstelle beschrieben werden können. Mit WSDL ist das jedoch nicht möglich. Implementieren zwei Webservices nach der WSDL-Beschreibung die gleichen Schnittstellen, kann das dazu führen, dass ein Webclient nur mit einem der beiden Webservices interoperabel ist, wie in Kapitel 2.7 festgestellt. Eine präzisere Beschreibung der Schnittstelle könnte das verhindern.

Nachfolgend sollen nicht in WSDL beschreibbare Anforderungen an unterschiedlichen Beispielen erläutert werden. Damit wird offensichtlich, wie leicht nichttriviale Schnittstellen solche Anforderungen enthalten können. Die Beispiele stammen aus der Schnittstelle der Webkomponente für eine Internetzeitung, die in Kapitel 2.6 vorgestellt wurde. Begonnen werden soll mit einfachen Anforderungen, die sich auf einzelne SOAP-Nachrichten beziehen. Dann werden die Anforderungen komplexer und beziehen sich zunächst auf einzelne Request-/Response-Paare und dann auf mehrere Operationsaufrufe gemeinsam. Weitere Beispiele für Anforderungen werden auch in Kapitel 6.2 gegeben. Damit soll dort ausgelotet werden, wo die Grenzen der Ausdrucksfähigkeit von WSDL liegen.

6.1.1 Anforderungen an einzelne SOAP-Nachrichten

Anforderungen an einzelne SOAP-Nachrichten können sich auf die SOAP-Nachricht als ganzes oder sogar nur auf einzelne Body-, Header- oder Detail-Einträge beziehen, die vollständig in XML-Schema beschrieben sind. Die Anforderung im ersten Beispiel bezieht sich nur auf einen Bodyeintrag. Es ist der Bodyeintrag des Responses der Operation `Categories` der Webkomponente für die Internetzeitung, die die Rubriken der Internetzeitung zurückliefert. Abbildung 51 zeigt einen solchen Response.

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope"
        xmlns:nwst="http://example.org/NewsService020917/Types">
  <env:Body>
    <nwst:CategoriesResponse>
      <nwst:NoCategories>6</nwst:NoCategories>
      <nwst:CategoriesSequence>
        <nwst:Category>Business</nwst:Category>
        <nwst:Category>Computer</nwst:Category>
        <nwst:Category>Culture</nwst:Category>
        <nwst:Category>Politics</nwst:Category>
        <nwst:Category>Sports</nwst:Category>
        <nwst:Category>University</nwst:Category>
      </nwst:CategoriesSequence>
    </nwst:CategoriesResponse>
  </env:Body>
</env:Envelope>
```

Abbildung 51: Beispiel für Response der Operation `Categories`

In der WSDL-Beschreibung des Webservices, wird der Bodyeintrag in XML-Schema als Element deklariert. Dabei werden seine Struktur und die Typen der Blattelemente festgelegt. In diesem Fall enthält das Blattelement `nwst:NoCategories` den numeri-

schen Wert 6. Er beschreibt, wie viele Rubriken die Internetzeitung hat und damit auch die Anzahl der Kindelemente des Elementes `nwst:CategoriesSequence`.

Die Tatsache, dass die Anzahl der Elemente `nwst:CategoriesSequence` stets der Wert von `nwst:NoCategories` sein muss, lässt sich in WSDL bzw. XML-Schema nicht ausdrücken. Jeder Webservice, der die Schnittstelle implementiert, muss die Anforderung jedoch befolgen. Auch Webclients können diese Tatsache nutzen, um z. B. Arrays mit genügend Platz für alle Rubriken zu allozieren. Daher sollte eine präzise Beschreibung der Schnittstelle auch eine solche Anforderung enthalten.

In solchen einfachen Fällen, in denen sich eine Anforderung nur auf ein einzelnes XML-Fragment bezieht, das vollständig in XML-Schema beschrieben wird, würde zur Beschreibung ein erweitertes XML-Schema ausreichen. Die Erweiterung müsste es erlaubten, Abhängigkeiten zwischen Werten von Blattelementen oder Attributen zu beschreiben. WSDL müsste hierzu nicht erweitert werden.

Das ist im nachfolgend gezeigten Beispiel anders. Dort muss eine Beziehung zwischen dem Header- und Bodyeintrag einer SOAP-Nachricht ausgedrückt werden, die nicht gemeinsam in XML-Schema beschrieben sind. Wie die XML-Fragmente zusammenhängen wird erst in WSDL selbst beschrieben. Daher erfordert die Beschreibung solcher Beziehungen auch Erweiterungen von WSDL.

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope"
    xmlns:nwst="http://example.org/NewsService020917/Types"
    xmlns:nwsext="http://example.org/NewsService020917/Extension">
  <env:Header>
    <nwsext:AvailableMessages>2</nwsext:AvailableMessages>
  </env:Header>
  <env:Body>
    <nwst:ListResponse>
      <nwst:MessageID>80AF91EA-32EB-4691-9ED5-D0BF047B9424</nwst:MessageID>
      <nwst:MessageID>AD6F371A-ABE9-41f5-8362-AAE7640F1DDD</nwst:MessageID>
    </nwst:ListResponse>
  </env:Body>
</env:Envelope>
```

Abbildung 52: Beispiel für Response der Operation `List` mit Headereintrag

Abbildung 52 zeigt eine solche SOAP-Nachricht. Es handelt sich um einen Response der Operation `List`. Dieser liefert die IDs von Artikeln (Nachrichten), die in der Internetzeitung enthalten sind, jeweils als Wert eines Blattelementes `nwst:MessageID`. Im Beispiel wird nun angenommen, dass eine bestimmte Implementierung des Webservices dem Response einen Headereintrag `nwsext:AvailableMessages` zugefügt hat, der angibt, wie viele Artikel geliefert werden. Analog zum Beispiel in Abbildung 51 muss dann der Wert von `nwsext:AvailableMessages` stets gleich der Anzahl der Elemente `nwst:MessageID` sein. Diese Anforderung lässt sich in WSDL nicht ausdrücken.

Ebenso wie Abhängigkeiten zwischen Header- und Bodyeinträgen, kann es in Fehlernachrichten Beziehungen zwischen Header- und Detaileinträgen geben. Auch diese lassen sich in WSDL nicht ausdrücken. Hinzu kommen auch Beziehungen zwischen Detaileinträgen und den verwendeten Fehlercodes. Solche sind wichtig, weil die Fehlercodes angeben, um welche Fehlersituation es sich handelt. Detaileinträge werden dagegen verwendet, um die Fehlersituation anwendungsspezifisch genauer zu beschreiben. Dabei auftretende Beziehungen sollten ausgedrückt werden können. Die Beschreibung von Fehlernachrichten wird noch genauer in Kapitel 6.2.4 betrachtet.

6.1.2 Anforderungen an einzelne Request-/Response-Paare

Ebenso wie Anforderungen an einzelne SOAP-Nachrichten, gibt es auch solche, die sich auf einzelne Request-/Response-Paare beziehen und damit auf einzelne Operationsaufrufe. Um die Anforderung zu erfüllen, muss der Response abhängig vom gesendeten Request geeignet gebildet werden. SOAP-Nachrichten zu einem einfachen Beispiel dieser Art werden in Abbildung 53 und Abbildung 54 gezeigt.

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope"
        xmlns:nwst="http://example.org/NewsService020917/Types">
  <env:Body>
    <nwst:Get>
      <nwst:SessionID>8CCF4AB6-FF9E-451d-8675-1B4F56B05296</nwst:SessionID>
      <nwst:MessageIDsSequence>
       <nwst:MessageID>801F91EF-32EB-4691-9ED5-4072047B9424</nwst:MessageID>
       <nwst:MessageID>AD6F371A-ABE9-41f5-8362-AAE7640F1DDD</nwst:MessageID>
      </nwst:MessageIDsSequence>
      <nwst:Lock>false</nwst:Lock>
    </nwst:Get>
  </env:Body>
</env:Envelope>
```

Abbildung 53: Beispiel für Request der Operation `Get`

Abbildung 53 zeigt einen Request der Operation `Get`. Die Operation erlaubt, für eine ID den mit ihr bezeichneten Artikel abzufragen. Mit einem Operationsaufruf kann das für eine Vielzahl von Artikeln geschehen. Für jeden enthält der Bodyeintrag des Requests ein Element `nwst:MessageIDsSequence/nwst:MessageID`, das die ID des Artikels enthält. Die Artikel werden im Response geliefert, wie in Abbildung 54 gezeigt. Jeder gelieferte Artikel befindet sich im Bodyeintrag in einem Kindelement `nwst:Message`.

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope"
        xmlns:nwst="http://example.org/NewsService020917/Types">
  <env:Body>
    <nwst:GetResponse>
      <nwst:Message>
        <nwst:Category>Culture</nwst:Category>
        <nwst:Title>Marcus V. in concert on Saturday!</nwst:Title>
        <nwst:Abstract>Hamburg. On Saturday is  … </nwst:Abstract>
        <nwst:Date>2002-03-19</nwst:Date>
       <nwst:ImageURL>http://www.example.org/img/hamburg.jpg</nwst:ImageURL>
        <nwst:Text>...</nwst:Text>
        <nwst:Public>false</nwst:Public>
      </nwst:Message>
      <nwst:Message>
        <nwst:Category>Computer</nwst:Category>
        <nwst:Title>New XML Webservice</nwst:Title>
        <nwst:Abstract>Hamburg. The TUHH shows … </nwst:Abstract>
        <nwst:Date>2002-03-18</nwst:Date>
        <nwst:Text>...</nwst:Text>
        <nwst:Public>true</nwst:Public>
      </nwst:Message>
    </nwst:GetResponse>
  </env:Body>
</env:Envelope>
```

Abbildung 54: Beispiel für Response der Operation `Get`

In WSDL kann für den Request nur ausgedrückt werden, dass in einem Element `nwst:MessageIDsSequence` mehrere Kindelemente `nwst:MessageID` erlaubt sind. Analog kann für den Bodyeintrag im Response festgelegt werden, dass Kindelemente `nwst:Message` erlaubt sind. Da jedoch stets für jedes Element `nwst:MessageID` genau ein Element `nwst:Message` geliefert wird, müssen die Anzahlen dieser beiden Elemente in Request bzw. Response gleich sein. Das lässt sich in WSDL nicht ausdrücken.

Die gezeigte Beziehung besteht zwischen den Bodyeinträgen des Requests und Responses. Ebenso kann es auch Beziehungen zwischen Body-, Header- und Detaileinträgen geben. So könnte es z. B. Beziehungen zwischen Headereinträgen in Request und Response geben oder zwischen einen Headereintrag im Request und einem Bodyeintrag im Response.

Besonders interessant sind auch Beziehungen zwischen Requests und Fehlernachrichten, die als Responses geliefert werden. Solche Beziehungen können ausdrücken, dass eine Fehlernachricht geliefert wird, falls der Request eine bestimmte Eigenschaft hat. Dabei kann z. B. festgelegt werden, welcher Fehlercode geliefert wird oder dass Detaileinträge bestimmte Werte enthalten. Somit können mit Anforderungen an Beziehungen zwischen Requests und Fehlernachrichten indirekt Aussagen über ungültige Requests gemacht werden.

6.1.3 Anforderungen an mehrere Operationsaufrufe

Viele Anforderungen beschreiben Beziehungen zwischen Requests und Responses unterschiedlicher Operationsaufrufe. Dabei können mehr als zwei SOAP-Nachrichten miteinander in Beziehung gesetzt werden. Es könnte z. B. Anforderungen geben, die alle Nachrichten eines bestimmten Typs miteinander in Beziehung setzen.

Anforderungen können auch vorschreiben, dass Operationsaufrufe nur in bestimmten Reihenfolgen erlaubt sind. Häufiger wird es jedoch der Fall sein, dass es Beziehungen zwischen Parametern von Operationsaufrufen gibt. Z. B. können erlaubte Werte von Parametern eines Operationsaufrufs von Parametern vorheriger Operationsaufrufe abhängen.

Wann ein Operationsaufruf erlaubt ist oder wie sich ein Webservice verhält, kann auch vom Zustand des Webservices abhängen. In vielen Fällen ist der Zustand des Webservices jedoch von vorherigen Operationsaufrufen abhängig und lässt sich durch diese beschreiben. Damit lassen sich Anforderungen an Zustände des Webservice auf Beziehungen zwischen Nachrichten zurückführen. Im Folgenden sollen sich Anforderungen daher nur auf Nachrichten beziehen.

Ein Anwendungsbereich, dessen Details durch eine Menge von Anforderungen beschrieben werden können, ist ein Mechanismus für Sessions. Eine Session fasst eine Reihe von Operationsaufrufen eines Webclients zusammen. Sie dient im Webservice zur Verwaltung von Zustandsinformationen für den Webclient. Mit einer Menge von Anforderungen kann festgelegt werden, welche Beziehungen zwischen Operationsaufrufen gelten müssen, damit der Mechanismus korrekt funktioniert.

Im Beispiel der Webkomponente für eine Internetzeitung ist die Authentifikation mit dem Mechanismus für Sessions kombiniert. Eine Session wird mit der Operation `Login` begonnen, die auch den Benutzer authentifiziert. Sie liefert eine Session-ID, die in nachfolgenden Operationen als Parameter angegeben wird, um die Session zu identifizieren. Beendet wird die Session mit der Operation `Logout`. Wurde diese Operation für die Session-ID aufgerufen, wird diese ungültig und darf nachfolgend nicht mehr bei Operationen angegeben werden.

Zur Illustration des Mechanismus für Sessions zeigt Abbildung 55 ein Beispiel für einen Response der Operation `Login`. Er enthält im Bodyeintrag das Kindelement `nwst:SessionID`, in dem die Session-ID geliefert wird. Im Beispiel des Requests der Operation `Get` in Abbildung 53 (Seite 100) wurde die Session-ID als Parameter verwendet.

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope"
        xmlns:nwst="http://example.org/NewsService020917/Types">
  <env:Body>
    <nwst:LoginResponse>
      <nwst:SessionID>8CCF4AB6-FF9E-451d-8675-1B4F56B05296</nwst:SessionID>
    </nwst:LoginResponse>
  </env:Body>
</env:Envelope>
```

Abbildung 55: Beispiel für Response der Operation `Login`

Ein Webclient, der die Webkomponente für die Internetzeitung verwendet, muss den Mechanismus für die Sessions korrekt verwenden, so wie oben informell beschrieben. Formaler lässt sich die Beschreibung als eine Menge von Anforderungen ausdrücken. So darf der Webclient in Requests von `Get`, `List`, `Logout`, `ChangeCategory`, u. s. w. nur eine Session-ID angeben, die vorher von der Operation `Login` geliefert wurde. Vorher darf für die Session-ID nicht die Operation `Logout` aufgerufen worden sein. Eine zweite Anforderung könnte sein, dass ein Webclient für jede Session-ID, die von einer Operation `Login` geliefert wurde, irgendwann die Operation `Logout` aufrufen muss.

Auch der Webservice muss den Mechanismus korrekt implementieren. Er muss zumindest die Anforderung beachten, dass jeder Aufruf von `Login` eine unterschiedliche `Session-ID` zu liefern hat. Außerdem sollte gefordert werden, welche Fehlernachricht er bei einer ungültigen `Session-ID` liefert. Sie ist ungültig, wenn sie nicht von der Operation `Login` stammt oder für die vorher die Operation `Logout` aufgerufen wurde. Für Fehlernachrichten sollten zumindest Fehlercode (`env:Fault/env:Code/env:Value`) und Subfehlercode (`env:Fault/env:Code/env:Subcode/env:Value`) festgelegt werden sowie gegebenenfalls der Inhalt von Detail-Einträgen.

Alle genannten anwendungsspezifischen Anforderung an dem Mechanismus für Sessions können in WSDL nicht ausgedrückt werden. Das gleiche gilt auch für generelle, nicht anwendungsspezifische Anforderungen, die man in einigen Fällen über Webservices fordern möchte. So könnte es einen Webservice geben, der empfangene Requests nach einer festgelegten Strategie bearbeitet, z. B. First-In-First-Out (FIFO)[28]. Auch das lässt sich in WSDL nicht ausdrücken.

Wichtig wäre es auch beschreiben zu können, wenn die Nebenläufigkeit, mit der Operationen aufgerufen werden dürfen, eingeschränkt ist. Der SOAP-Standard [Gudgin 02b] beschränkt sich bei der Beschreibung des Request-/Response-Kommunikationsmusters auf die Beschreibung einer Kommunikation von Request und Response. In der Regel können mehrere solcher Request-/Response-Kommunikationen gleichzeitig stattfinden. Gibt es hier Einschränkungen, ist das Teil der Schnittstelle und sollte beschrieben werden können, was in WSDL nicht möglich ist.

[28] In der Strategie FIFO werden Responses in genau der Reihenfolge gesendet werden, in der die zugehörigen Requests empfangen wurden.

Im einfachsten Fall könnte ein Webservice überhaupt keine Nebenläufigkeit zulassen. Dann müsste gefordert werden, dass alle Webclients erst dann wieder einen Request senden, wenn der Webservice mit einem Response auf den vorherigen Request geantwortet hat. Realistischer wäre die Anforderung, dass ein Webservice innerhalb einer Session keine Nebenläufigkeit zulässt. Dann müssten die obige Anforderung nur einzelne Webclients beachten.

Im Beispiel der Webkomponente für eine Internetzeitung ist die Nebenläufigkeit bezogen auf einzelne Artikel eingeschränkt. Mit der Operation `Get` kann eine Sperre (Schreibsperre) auf einen oder mehrere Artikel gesetzt werden. Danach darf nur innerhalb der Session auf die gesperrten Artikel zugegriffen werden. Die Sperre wird wieder freigegeben, wenn der Webclient die Operation `Unlock` aufruft oder die Session mit der Operation `Logout` beendet.

Nicht jede Operation `Get` setzt automatisch eine Sperre. Im Request aus Abbildung 53 auf Seite 100 hat das Element `nwst:Lock` den Wert `false`, so dass keine Sperre gesetzt wird. Das wäre auch der Fall, wenn das optionale Element `nwst:Lock` fehlen würde. Im Beispiel in Abbildung 56 ist das anders. Mit dem Wert `true` im Element `nwst:Lock` wird angezeigt, dass auf alle Artikel, die abgefragt werden, eine Sperre gesetzt werden soll.

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope"
              xmlns:nwst="http://example.org/NewsService020917/Types">
  <env:Body>
    <nwst:Get>
      <nwst:SessionID>8CCF4AB6-FF9E-451d-8675-1B4F56B05296</nwst:SessionID>
      <nwst:MessageIDsSequence>
       <nwst:MessageID>80AF91EA-32EB-4691-9ED5-D0BF047B9424</nwst:MessageID>
      </nwst:MessageIDsSequence>
      <nwst:Lock>true</nwst:Lock>
    </nwst:Get>
  </env:Body>
</env:Envelope>
```

Abbildung 56: Beispiel für Request der Operation `Get`, der Sperre setzt

Wie der Mechanismus für Sessions kann der auch dieser Mechanismus durch eine Menge von Anforderungen beschrieben werden, die sich auf Nachrichten beziehen. Zunächst muss gefordert werden, dass Operationen, mit denen auf Artikel zugegriffen wird (`Delete`, `Publish`, `ChangeCategory`, `Unlock` ...), eine Fehlernachricht liefern, wenn der betreffende Artikel von einer anderen Session gesperrt ist. Der Zustand, dass der Artikel von einer Session gesperrt ist, wird mit Hilfe von Nachrichten formuliert. Ein Artikel A ist von einer Session S gesperrt, wenn vorher in der Session S für den Artikel A die Operation `Get` aufgerufen wurde, wobei `nwst:Lock` den Wert `true` hatte und dabei nicht als Response eine Fehlernachricht gesendet wurde und danach nicht in der Session S die Operation `Logout` oder für den Artikel A die Operation `Unlock` aufgerufen wurde. Ist der Artikel in einer anderen Session gesperrt, muss gefordert werden, welcher Fehlercode (`env:Fault/env:Code`) in der Fehlernachricht geliefert wird.

Eine zweite Anforderung betrifft die Operation `Get`. Enthält ihr Request in einem Element `nwst:MessageID` den Wert eines Artikels, der bereits in einer anderen Session gesperrt wurde, so liefert die Operation als Response eine Fehlernachricht. Ein Webclient darf die Operation `Unlock` für einen Artikel nur dann aufrufen, wenn er vorher für

den Artikel in der gleichen Session mit der Operation `Get` erfolgreich eine Sperre gesetzt hat. Erfolgreich bedeutet dabei, dass als Response keine Fehlernachricht geliefert wurde.

Hätte der Mechanismus für Sperren mehr Möglichkeiten, wären auch mehr Anforderungen notwendig. Das wäre z. B. der Fall, wenn es zusätzlich zu den Schreibsperren auch Lesesperren gäbe. Ebenso müssten Anforderungen zugefügt werden, wenn mit Hilfe von Headereinträgen Operationensaufrufe Teil verteilter Transaktionen werden könnten, wie Abbildung 9 auf Seite 31 angedeutet. Wie alle vorher beschriebenen Anforderungen, lässt sich das in WSDL jedoch nicht ausdrücken.

6.2 Grenzfälle

Interessant ist nun zu untersuchen, wo die Grenzen der Beschreibungsfähigkeit von WSDL liegen. Was kann noch in WSDL beschrieben werden? Ab wann reicht die Beschreibungsfähigkeit von WSDL nicht mehr aus? Das soll nachfolgend mit Hilfe von Grenzfällen untersucht werden. Für verschiedene Beispiele wird aufgezeigt, was noch in WSDL beschrieben werden kann. Danach wird jeweils gezeigt, dass eine präzisere Beschreibung zweckmäßig wäre, aber nicht möglich ist.

6.2.1 Optionale Parameter

Ein einfacher Grenzfall bezieht sich auf optionale Parameter. Optionale Parameter einer Operation sind solche, die vorhanden sein können, aber nicht müssen. Häufig sind es Kindelemente des Bodyeintrages deren Vorhandensein optional ist. Das lässt sich in XML-Schema beschreiben. Es lässt sich aber nicht ausdrücken, in welchen Fällen ein optionales Kindelement vorhanden sein muss.

Die Webkomponente für eine Internetzeitung enthält verschiedene optionale Kindelemente. Z. B. ist bei der Operation `Login` das Passwort optional. Daher kann im Request das in Abbildung 57 gezeigte Element `nwst:Password` fehlen. Abbildung 58 zeigt die Spezifikation des Bodyeintragen in XML-Schema. In der Deklaration des Kindelementes `nwst:Password` wird das Attribut `minOccurs` angegeben. Es erhält den Wert 0 um anzuzeigen, dass das Kindelement optional ist.

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope"
        xmlns:nwst="http://example.org/NewsService020917/Types">
  <env:Body>
    <nwst:Login>
      <nwst:Name>Venzke</nwst:Name>
      <nwst:Password>igelgruen</nwst:Password>
    </nwst:Login>
  </env:Body>
</env:Envelope>
```

Abbildung 57: Beispiel für Request der Operation `Login` mit Passwort

```
<xsd:complexType name="loginRequest">
  <xsd:sequence>
    <xsd:element ref="nwst:Name" />
    <xsd:element ref="nwst:Password" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:element name="Login" type="nwst:loginRequest" />
```

Abbildung 58: Deklaration des Bodyeintrages der Operation `Login`

Somit ist es möglich, optionale Parameter auszudrücken. Häufig hängt die Tatsache, wann ein Parameter optional ist, jedoch von anderen Parametern bzw. Werten in der SOAP-Nachricht ab. Im Fall der Operation `Login` wird z. B. der Parameter für das Passwort nur genau dann nicht angegeben, wenn es sich beim Benutzer um einen Gast handelt. Das wird mit dem Benutzernamen „Gast" im Element `nwst:Name` angezeigt. Wird ein anderer Benutzername angegeben, muss auch das Passwort angegeben werden.

Dass das Vorhandensein des optionalen Passwortes vom Benutzernamen abhängig ist, lässt sich in WSDL nicht ausdrücken. Das gleiche gilt auch für andere optionale Elemente oder Attribute, bei denen sich das Vorhandensein aus anderen Teilen der SOAP-Nachricht oder sogar aus anderen SOAP-Nachrichten ergibt.

6.2.2 Alternative Struktur

Ein anderer Grenzfall sind Alternativen in der Struktur eines XML-Fragmentes, wenn also z. B. entweder einige Kindelemente oder aber ein Attribut vorhanden sein muss, aber nicht beides vorhanden sein darf. In gewisser Weise handelt es sich dann um zwei optionale Teile im XML-Fragment, wie sie im vorherigen Kapitel 6.2.1 beschrieben wurden, wobei das Vorhandensein des einen Teiles von der Nichtexistenz des anderen abhängt.

Der Grenzfall hat eine erhebliche Relevanz für SOAP-Nachrichten, weil gerade dieser Fall bei der Kodierung von mehrfach referenzierten Knoten durch das SOAP-Encoding von SOAP 1.2 auftritt, was in Kapitel 3.7 vorgestellt wurde. Daher wird der Grenzfall auch in [Ewald 01] für SOAP 1.1 betrachtet.

Beim SOAP-Encoding von SOAP 1.2 wird ein mehrfach referenzierter Knoten nur einmal vollständig als Element in ein XML-Fragment transformiert. Für alle weiteren Vorkommnisse des Knotens wird auf dieses Element verwiesen. Abbildung 59 zeigt ein XML-Fragment das ein Beispiel enthält[29]. Das Element `my:c/my:a` enthält den Knotens vollständig. Daten sind im Kindelement enthalten. Zusätzlich hat es das Attribut `id`, dessen Wert den Knoten identifiziert, so dass es im XML-Dokument referenziert werden kann. Das geschieht im zweiten Vorkommnis des Knotens im Element `my:c/my:b/my:a`. Statt des Kindelementes gibt es hier nur das Attribut `ref`. Es enthält den Wert, der den Knoten identifiziert.

```
<my:c xmlns:my="http://example.org/MultirefSample">
  <my:a id="r123">
    <my:str>Hallo</my:str>
  </my:a>
  <my:b>
    <my:a ref="r123" />
  </my:b>
</my:c>
```

Abbildung 59: XML-Fragment eines mehrfach referenzierten Knotens

Ein solches XML-Fragment kann in WSDL beschrieben werden, indem für das Use-Attribut der Wert `encoded` gewählt wird. (Siehe Kapitel 4.3.7.2.) Dabei muss explizit angegeben werden, dass das SOAP-Encoding verwendet wird. In Kapitel 5.3.3 wurde jedoch für das Use-Attribut gefordert, dass der Wert `literal` gewählt und das XML-

[29] Dieses Beispiel gehört im Gegensatz zu den anderen nicht zur Webkomponente für eine Internetzeitung. Dort werden keine mehrfach referenzierten Knoten verwendet.

Fragment in XML-Schema beschrieben werden muss. Dann ist es auch erforderlich, mehrfach referenzierte Knoten in XML-Schema zu spezifizieren. Abbildung 60 zeigt, wie dazu das Element `a` deklariert werden könnte.

```
<xsd:complexType name="aType">
  <xsd:sequence>
    <xsd:element ref="my:str" minOccurs="0" />
  </xsd:sequence>
  <xsd:attribute name="ref" type="xsd:IDREF" use="optional" />
  <xsd:attribute name="id" type="xsd:ID" use="optional" />
</xsd:complexType>

<xsd:element name="a" type="my:aType" />
```

Abbildung 60: Deklaration des Elementes für mehrfach referenzierten Knoten

Mit dem Attribut `minOccurs` wird angezeigt, dass das Kindelement `my:str`, das die eigentlichen Daten enthält, optional ist. Ebenso wird das Attribut `ref` als optional deklariert, wozu in XML-Schema das Attribut `xsd:attribute/@use` verwendet wird. Damit beschreibt diese Deklaration beide im XML-Fragment in Abbildung 59 verwendeten Elemente `a`. Leider beschreibt sie jedoch nicht, dass nur das Kindelement oder das Attribut `ref` vorhanden sein darf, aber nicht beides. Nach der Deklaration in Abbildung 60 wäre daher auch das Element `my:c/my:b/my:a` im XML-Dokument in Abbildung 61 gültig, das sowohl das Kindelement als auch das Attribut `ref` enthält. Ebenso wäre dann erlaubt, dass weder das Kindelement noch das Attribut `ref` vorhanden ist. Beides ist nach dem SOAP-Encoding jedoch nicht erlaubt und auch nicht sinnvoll.

```
<my:c xmlns:my="http://example.org/MultirefSample">
  <my:a id="r123">
    <my:str>Hallo</my:str>
  </my:a>
  <my:b>
    <my:a ref="r123">
      <my:str>Welt</my:str>
    </my:a>
  </my:b>
</my:c>
```

Abbildung 61: Nach SOAP-Encoding ungültiges XML-Fragment

Es bleibt die Frage, ob das XML-Fragment in XML-Schema nicht präziser beschrieben werden kann, so dass nur die Fälle gültig sind, die auch das SOAP-Encoding erlaubt. Die einzige Möglichkeit in XML-Schema Alternativen auszudrücken, ist die Verwendung von `xsd:choice`, die bereits in Kapitel 4.2.6.1 beschrieben wurde. Damit kann ausgedrückt werden, dass im XML-Fragment ein Element aus einer Menge von Elementen erlaubt ist. Die Elemente in der Menge müssen unterschiedliche Namen haben. Die Verwendung von `xsd:choice` bezieht sich jedoch nicht auf Attribute. Damit kann nicht ausgedrückt werden, dass entweder ein Attribut oder ein Kindelement vorhanden sein muss.

Somit ist es nicht möglich, mehrfach referenzierte Knoten in XML-Schema präzise zu spezifizieren. Die mögliche Spezifikation mit optionalen Attributen und Kindelementen erlaubt auch XML-Fragmente, die nicht gewünscht sind. Trotzdem ist es mit dem SOAP-Encoding und der Verwendung des Wertes `encoded` für das Use-Attribut natürlich möglich, mehrfach referenzierte Knoten in WSDL zu spezifizieren. Das wurde aber in Kapitel 5.3.3 verboten. Außerdem kann so nur dieser speziellen Art von Alternativen

spezifiziert werden. Andere Alternativen in der Struktur von XML-Fragmenten, die nicht in XML-Schema mit `xsd:choice` spezifiziert werden können, können also auch mit dem SOAP-Encoding nicht beschrieben werden.

6.2.3 Dynamische Aufzählungstypen

Der eben beschriebene Grenzfall bezieht sich ebenso wie der aus Kapitel 6.2.1 auf die Präzision, mit der einzelne XML-Fragmente innerhalb einer SOAP-Nachricht beschrieben werden können. Der Grenzfall, der nun vorgestellt werden soll, zeigt dagegen das Problem auf, dass zur Beschreibung einer SOAP-Nachricht „Wissen" über eine andere SOAP-Nachricht benötigt wird. Es geht um eine dynamische Form von Aufzählungstypen.

Einen dynamischen Aufzählungstypen benötigt auch die Webkomponente für die Internetzeitung. Die Internetzeitung hat verschiedene Rubriken. Artikel sind den Rubriken zugeordnet. Daher muss bei unterschiedlichen Operationen eine Rubrik als Parameter angegeben werden. Sie ist in den Requests der Operationen `ChangeCategory`, `Create` und `List` enthalten und im Response der Operation `Get`. Dabei wird jeweils ein Element `nwst:Category` verwendet, wie es im nachfolgenden Beispiel in Abbildung 62 mit einen Request der Operation `ChangeCategory` gezeigt wird.

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope"
              xmlns:nwst="http://example.org/NewsService020917/Types">
  <env:Body>
    <nwst:ChangeCategory>
      <nwst:SessionID>8CCF4AB6-FF9E-451d-8675-1B4F56B05296</nwst:SessionID>
      <nwst:MessageID>80AF91EA-32EB-4691-9ED5-D0BF047B9424</nwst:MessageID>
      <nwst:Category>Culture</nwst:Category>
    </nwst:ChangeCategory>
  </env:Body>
</env:Envelope>
```

Abbildung 62: Beispiel für Request der Operation `ChangeCategory`

Im Element `nwst:Category` dürfen nur Rubriken angegeben werden, die die Internetzeitung auch hat. Diese Anforderung ist Teil der Schnittstelle des Webservices und sollte daher auch in WSDL beschrieben werden können. Für eine feste Menge von Rubriken ist das möglich. Hierzu könnten in XML-Schema definierte Aufzählungstypen verwendet werden, die in Kapitel 4.2.5 vorgestellt wurden. Dazu wird ein bereits definierter einfacher Typ mit `xsd:restriction` auf die erlaubten Werte beschränkt. Für jeden erlaubten Wert hat `xsd:restriction` ein Kindelement `xsd:enumeration`, das den erlaubten Wert enthält.

```
<xsd:simpleType name="categories">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Business" />
    <xsd:enumeration value="Computer" />
    <xsd:enumeration value="Culture" />
    <xsd:enumeration value="Politics" />
    <xsd:enumeration value="Sports" />
    <xsd:enumeration value="University" />
  </xsd:restriction>
</xsd:simpleType>

<xsd:element name="Category"  type="nwst:categories" />
```

Abbildung 63: Deklaration des Elementes `nwst:Category` mit `xsd:enumeration`

Abbildung 63 zeigt ein Fragment eines Schemadokumentes, in dem auf diese Art das Element `nwst:Category` deklariert wird. Verkürzt wurde es bereits in Abbildung 29 gezeigt. Es wird zunächst der von XML-Schema vordefinierte einfache Typ `xsd:string` auf die Werte `Business`, `Computer`, `Culture`, `Politics`, `Sports` und `University` beschränkt und so ein neuer einfacher Aufzählungstyp für Rubriken definiert. Dieser würde bei der Deklaration des Elementes `nwst:Category` verwendet werden.

Nachteil dieser Vorgehensweise ist, dass dann nicht nur zur Schnittstelle gehört, dass nur erlaubte Rubriken angegeben werden dürfen, sondern auch welche Rubriken die Internetzeitung hat. Damit würde sich auch die Schnittstelle ändern, wenn die Internetzeitung ihre Rubriken ändert und gegen die Schnittstelle implementierte Webclients müssten angepasst werden. Zwar würden häufig Webclients trotz der Änderung des Aufzählungstyps weiter verwendbar sein. Da aber die Schnittstelle Vertrag zwischen Webclient und Webservice ist, kann das nicht garantiert werden. Der Webclient könnte in seiner Implementierung Annahmen über die erlaubten Werte des Elementes `nwst:Category` gemacht haben und mit anderen Werten nicht korrekt funktionieren.

Ebenso wäre es bei dieser Vorgehensweise nicht möglich, dass Webservices von zwei Internetzeitungen mit unterschiedlichen Rubriken dieselbe Schnittstelle haben. Das verhindert, dass die gleichen Webclients für beide Webservices verwendet werden können. Webclients müssten also speziell für eine Internetzeitung implementiert werden.

Aus diesen Gründen sind bei der Webkomponente für die Internetzeitung die Rubriken nicht Teil der Schnittstelle. Trotzdem sollen nur die erlaubten Rubriken als Parameter verwendet werden dürfen. Um das zu ermöglichen, können die Rubriken der Internetzeitung innerhalb der Webkomponente z. B. in einer Datenbank abgelegt werden. Mit Hilfe der Operation `Categories` kann ein Webclient abfragen, welche Werte für Rubriken erlaubt sind. Ein Beispiel für einen Response dieser Operation ist in Abbildung 64 gezeigt. Er enthält die gleichen Rubriken, wie der in Abbildung 63 definierte Aufzählungstyp.

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope"
        xmlns:nwst="http://example.org/NewsService020917/Types">
  <env:Body>
    <nwst:CategoriesResponse>
      <nwst:NoCategories>6</nwst:NoCategories>
      <nwst:CategoriesSequence>
        <nwst:Category>Business</nwst:Category>
        <nwst:Category>Computer</nwst:Category>
        <nwst:Category>Culture</nwst:Category>
        <nwst:Category>Politics</nwst:Category>
        <nwst:Category>Sports</nwst:Category>
        <nwst:Category>University</nwst:Category>
      </nwst:CategoriesSequence>
    </nwst:CategoriesResponse>
  </env:Body>
</env:Envelope>
```

Abbildung 64: Beispiel für einen Response der Operation `Categories`

Während es nun möglich ist, Aufzählungstypen in XML-Schema zu definieren, gibt es in WSDL keine Möglichkeit auszudrücken, dass nur Werte erlaubt sind, die von einer anderen Operation geliefert werden. Im Gegensatz zu den anderen Anforderungen, die in diesem Kapitel 6.2 und dem vorherigen Kapitel 6.1 vorgestellt wurden, kann diese

auch nicht als Beziehung zwischen SOAP-Nachrichten beschrieben werden. Ein Webclient muss nämlich nicht die Operation `Categories` verwenden, um die gültigen Werte abzufragen. Ein Wert ist auch dann gültig, wenn die Operation `Categories` ihn liefern würde, wenn sie denn aufgerufen würde. Es muss also nie eine SOAP-Nachricht ausgetauscht worden sein, die den gültigen Wert enthält. Es wäre wünschenswert, auch solche Anforderungen ausdrücken zu können, was ist WSDL jedoch nicht möglich ist.

6.2.4 Fehlernachrichten

Fehlernachrichten (siehe Kapitel 3.4) sind bei der Untersuchung der Grenzen von WSDL ein weiterer, interessanter Bereich. Teile von Fehlernachrichten sind bereits im SOAP-Standard festgelegt, weitere Teile können in WSDL spezifiziert werden. Viele wichtige Beziehungen und Werte können in WSDL jedoch nicht ausgedrückt werden.

Im SOAP-Standard festgelegte Teile von Fehlernachrichten legen vor allem deren Struktur und Typen von Werten fest. Zusätzlich zur ohnehin verwendeten Nachrichtenstruktur jeder SOAP-Nachricht, wird mit dem Element `env:Fault` und seinen Kindelementen die Struktur festgelegt, mit Fehler beschrieben werden. Das erlaubt jedem Webclient prinzipiell, Fehlernachrichten zu interpretieren. Im SOAP-Standard sind die qualifizierten Namen der in `env:Fault/env:Code/env:Value` erlaubten Fehlercodes festgelegt. Nur mit Detaileinträgen, also Kindelementen von `env:Fault/env:Detail`, kann die Struktur der Fehlernachricht anwendungsspezifisch erweitert werden.

Ein Beispiel für eine Fehlernachricht wird in Abbildung 65 gezeigt[30]. Sie wird als Response geliefert, wenn bei der Operation `Get` der Webkomponente für eine Internetzeitung im Request (siehe Abbildung 53 bzw. Abbildung 56) mindestens eine ungültige ID eines Artikels angegeben wurde. Zu erkennen sind die festgelegte Struktur sowie der Detaileintrag `nwst:MessageIDsSequence`. Letzterer enthält die anwendungsspezifische Erweiterung, mit der der Webclient darüber informiert wird, welche der im Request angegebenen IDs ungültig waren.

```
<?xml version="1.0" encoding="UTF-8" ?>
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope"
        xmlns:nwst="http://example.org/NewsService020917/Types"
        xmlns:nwse="http://example.org/NewsService020917/Errors">
  <env:Body>
    <env:Fault>
      <env:Code>
        <env:Value>env:Sender</env:Value>
        <env:Subcode>
          <env:Value>nwse:InvalidMessageID</env:Value>
        </env:Subcode>
      </env:Code>
      <env:Reason>Provided MessageID(s) invalid.</env:Reason>
      <env:Detail>
        <nwst:MessageIDsSequence>
          <nwst:MessageID>80AF91EA-32EB-4691-9ED5-D0BF047B9424</nwst:MessageID>
          <nwst:MessageID>AD6F371A-ABE9-41f5-8362-AAE7640F1DDD</nwst:MessageID>
        </nwst:MessageIDsSequence>
       </env:Detail>
    </env:Fault>
  </env:Body>
</env:Envelope>
```

Abbildung 65: Beispiel für eine SOAP-Fehlernachricht

[30] Die SOAP-Fehlernachricht aus Abbildung 65 wurde bereits in Kapitel 3.4 in Abbildung 10 gezeigt.

Neben den genannten Festlegungen aus dem SOAP-Standard kann die Fehlernachricht auch in WSDL beschrieben werden. Hier wird für eine Operation festgelegt, welche Fehlernachrichten sie als Response zurückliefern kann. Ebenso kann spezifiziert werden, welche Detaileinträge es in jeder Fehlernachricht gibt und wie sie in XML-Schema deklariert werden.

In WSDL befindet sich die Deklaration von Fehlernachrichten sowohl in der Beschreibung der abstrakten, wie auch der gebundenen Schnittstelle. In der abstrakten Schnittstelle wird festgelegt, welche Fehlernachrichten eine Operation hat. In der gebundenen Schnittstelle wird dagegen nur angegeben, dass das Use-Attribut den Wert `literal` hat. Das ist mit den Festlegungen aus Kapitel 5 bei jeder Operation in jeder gebundenen Schnittstelle der Fall.

Daher ist in Abbildung 66 als Beispiel nur ein Ausschnitt der Definition einer abstrakten Schnittstelle gezeigt[31]. Es ist die Definition der Operation `Get`. Mit jedem Element `wsdl:portType/wsdl:operation/wsdl:fault` wird eine mögliche Fehlernachricht deklariert. Da in der Definition der Operation `Get` drei solche Elemente enthalten sind, können als Responses der Operation `Get` drei verschiedene Fehlernachrichten geliefert werden.

```
<wsdl:message name="FaultDetailsMessageIDs">
  <wsdl:part name="FaultDetailsMessageIDs"
             element="nwst:MessageIDsSequence" />
</wsdl:message>

<wsdl:portType name="NewsServicePortType">
  <!-- ... -->

  <wsdl:operation name="Get">
    <wsdl:input  message="nwsi:GetRequest" />
    <wsdl:output message="nwsi:GetResponse" />
    <wsdl:fault  name="InvalidSessionID"
                 message="nwsi:FaultDetailsEmpty" />
    <wsdl:fault  name="InvalidMessageID"
                 message="nwsi:FaultDetailsMessageIDs" />
    <wsdl:fault  name="MessageLocked"
                 message="nwsi:FaultDetailsMessageIDs" />
  </wsdl:operation>

  <!-- ... -->
</wsdl:portType>
```

Abbildung 66: Definition der Operation `Get` in abstrakter Schnittstelle

Die in Abbildung 65 gezeigte Fehlernachricht wird mit Hilfe des zweiten Elementes `wsdl:fault` beschrieben. Für jede Fehlernachricht ist im Attribut `wsdl:fault/@name` ein Name angegeben, der sie innerhalb der Operation im Kontext des WSDL-Dokumentes eindeutig identifiziert. Für die Fehlernachricht in Abbildung 65 wird der Name `InvalidMessageID` angegeben. Dieser ist „zufällig“ identisch mit dem in der SOAP-Nachricht verwendeten Subfehlercode. Der WSDL-Standard legt einen solchen Zusammenhang aber nicht fest.

Mit dem Attribut `wsdl:fault/@message` wird in der Deklaration der Fehlernachricht auf eine in WSDL deklarierte Nachricht verwiesen. In deren Deklaration wird auf ein

[31] Abbildung 67 auf Seite 113 zeigt einen entsprechenden Ausschnitt der gebundenen Schnittstelle mit einer Erweiterung.

oder mehrere in XML-Schema deklarierten Elemente verwiesen, die die Detaileinträge der SOAP-Nachricht bilden. Zur Deklaration der Fehlernachricht aus Abbildung 65 wird daher mit dem Attribut `wsdl:fault[2]/@message` auf die Nachricht mit dem Namen `nwsi:FaultDetailsMessageIDs` verwiesen, deren Deklaration in Abbildung 66 mit enthalten ist. Die Deklaration der Nachricht verweist ihrerseits auf das in XML-Schema deklarierte Element `nwst:MessageIDsSequence`, das in Abbildung 65 als Detaileintrag zu erkennen ist.

In WSDL kann also für jede Operation festgelegt werden, welche Fehlernachrichten sie liefern kann und welche Detailelemente diese haben. Was in WSDL jedoch nicht ausgedrückt werden kann, sind diverse Beziehungen zwischen Werten innerhalb der Fehlernachricht und auch bestimmte Werte selbst, vor allem Fehlercodes und Subfehlercodes.

Erlaubte Fehlercodes sind im SOAP-Standard festgelegt. In WSDL kann aber nicht spezifiziert werden, welche Fehlercodes in welcher Fehlernachricht verwendet werden. Da in den gleichen Fehlernachricht einer Operation in der Regel der gleiche Fehlercode verwendet wird, sollte das möglich sein.

Ähnlich verhält es sich auch mit den Subfehlercodes in den Elementen `env:Subcode/env:Value`. Gibt es für eine Fehlernachricht nur einen möglichen Subfehlercode, wie es bei der Webkomponente für die Internetzeitung stets der Fall ist, sollte dieser spezifiziert werden können, was in WSDL nicht möglich ist. In der Fehlernachricht in Abbildung 65 wird z. B. stets der Subfehlercode `nwse:InvalidMessageID` verwendet. Im Gegensatz zur im SOAP-Standard grundsätzlich festgelegten Menge von Fehlercodes kommt bei Subfehlercodes hinzu, dass für sie beliebige qualifizierte Namen erlaubt sind. Sind daher für eine Fehlernachricht mehrere Subfehlercodes erlaubt, ist es noch wichtiger, dass spezifiziert werden kann, welche erlaubt sind.

Wie bei Fehlercodes und Subfehlercodes, ist es nicht möglich, erlaubte Werte für andere Kindelemente von `env:Fault` anzugeben. Es kann nicht spezifiziert werden, welche für Menschen lesbare Beschreibung im Element `env:Fault/env:Reason` enthalten ist. Auch die Werte der in Kapitel 5.2.6 verbotenen Kindelemente `env:Fault/env:Node` und `env:Fault/env:Role` können nicht spezifiziert werden.

Neben den erlaubten Werten selbst, wäre es auch wichtig, Beziehungen zwischen ihnen zu spezifizieren. So hat der Subfehlercode die Aufgabe, einen bestimmten Fehlercode zu präzisieren. In WSDL gibt es jedoch keine Möglichkeit festzulegen, welchen Fehlercode ein Subfehlercode präzisiert. Ebenso wäre es naheliegend für Fehlercodes oder Subfehlercodes die für Menschen lesbaren Beschreibungen festlegen zu wollen, was auch nicht möglich ist.

Wie bei Body- und Headereinträgen gibt es auch innerhalb von Detaileinträgen wichtige Beziehungen, die sich nicht in WSDL ausdrücken lassen. Das wurde bereits in den Kapiteln 6.1.1, 6.2.1, 6.2.2 und 6.2.3 diskutiert. Zusätzlich kann es in Fehlernachrichten Beziehungen zwischen Detaileinträgen und Fehlercodes bzw. Subfehlercodes geben. Auch diese können nicht in WSDL spezifiziert werden. Das wäre jedoch wichtig, weil Detaileinträge die durch Fehlercode und Subfehlercode beschriebene Fehlersituation genauer beschreiben, wie bereits in Kapitel 6.1.1 festgestellt.

Beziehungen kann es auch innerhalb der Fehlernachricht zwischen Headereinträgen und Fehlernachrichten geben. Der SOAP-Standard legt z. B. fest, dass ein Webservice mit einer Fehlernachricht mit Fehlercode `env:MustUnderstand` antworten muss, wenn er einen Request mit einem zwingenden Headereintrag enthält, den er nicht versteht oder nicht nach dessen Spezifikation verarbeiten kann. Dann muss er der Fehlernachricht

einen Headereintrag `flt:Misunderstood` zufügen, der den qualifizierten Namen des nicht verstandenen Headereintrages enthält. Das wurde bereits im Kapitel 3.4 beschrieben. Zwar ist diese Beziehung zwischen dem Fehlercode `env:MustUnderstand` und dem Headereintrag `flt:Misunderstood` im SOAP-Standard festgelegt. Ähnliche Abhängigkeiten in Fehlernachrichten sollten aber in WSDL ausgedrückt werden können, was nicht möglich ist.

Neben den bisher beschriebenen Beziehungen innerhalb einer Fehlernachricht, gibt es auch solche zwischen Fehlernachrichten und anderen SOAP-Nachrichten, insbesondere zum Request des gleichen Operationsaufrufes. Wie bereits in Kapitel 6.1.2 angesprochen, kann mit Anforderungen an solche Beziehungen festgelegt werden, in welchem Fall Fehlernachrichten geschickt werden müssen und damit, wann eine Fehlersituation vorliegt. Die Anforderung legt dann fest, dass eine bestimmte Nachricht oder eine Menge von Nachrichten zu einer Fehlersituation führt. Sie kann ausdrücken, welcher Fehlercode bzw. Subfehlercode in der Fehlernachricht geliefert wird und auch festlegen, was für Detaileinträge gelten muss.

Anforderungen an Beziehungen zwischen Request und Fehlernachricht eines Operationsaufrufes erlauben es Fehlersituationen anzuzeigen, die sich nur auf den Operationsaufruf beziehen. In Kapitel 6.2.1 auf Seite 104 wurde z. B. die Operation `Login` der Webkomponente für eine Internetzeitung vorgestellt. Das Passwort musste dort angegeben werden, wenn ein anderer Benutzername als „Gast“ angegeben wird. Hält sich der Webclient nicht an diese Anforderung, muss der Webservice mit einer Fehlernachricht antworten. Diese Tatsache kann in WSDL nicht ausgedrückt werden.

Das gilt auch für Fehlersituationen, die erst durch den Zusammenhang mehrerer SOAP-Nachrichten unterschiedlicher Operationsaufrufe ausgedrückt werden können. Z. B. ist es bei der Fehlersituation der Fall, die zur Fehlernachricht in Abbildung 65 führt, wenn also im Request der Operation `Get` eine oder mehrere ungültige IDs von Artikeln angegeben werden.

Die Tatsache, dass eine solche ID ungültig ist, kann nur mit Hilfe vieler SOAP-Nachrichten beschrieben werden. Mit der Operation `Create` wird ein Artikel angelegt, im Response wird seine ID geliefert. Die Operation `Delete` zerstört dagegen Artikel und macht seine ID, die im Request übertragen wurde, ungültig. Daher sind alle vor einem Aufruf der Operation `Get` ausgetauschten Responses der Operation `Create` und Requests der Operation `Delete` erforderlich, um zu entscheiden, ob eine ID gültig ist.

Es lässt sich zusammenfassen, dass sich mit WSDL über Fehlernachrichten nur spezifizieren lässt, welche Fehlernachrichten eine Operation liefern kann und welche Detaileinträge die Fehlernachrichten haben. Detaileinträge können in XML-Schema spezifiziert werden. WSDL erlaubt es jedoch nicht, Fehlercodes, Subfehlercodes und ähnliche Werte anzugeben. WSDL erlaubt es auch nicht, Beziehungen innerhalb der Fehlernachricht oder mit anderen SOAP-Nachrichten zu spezifizieren. Beides wäre jedoch erforderlich, um Fehlernachrichten und Fehlersituationen vollständig beschreiben zu können.

6.2.5 Headereinträge

Grenzen der Beschreibungsfähigkeit hat WSDL auch in Bezug auf Headereinträge. Allerdings ist es hier gar nicht gewünscht, vollständig zu beschreiben, welche Headereinträge es geben kann. Ein SOAP-Sender ist frei, zusätzliche Headereinträge einer

SOAP-Nachricht zuzufügen. Die Semantik von Headereinträgen und deren optionales Attribut `env:mustUnderstand` erlauben zu steuern, ob ein SOAP-Empfänger einen Headereintrag ignorieren darf oder nicht. Das wurde bereits in Kapitel 3.3 vorgestellt.

Damit ist grundsätzlich freigestellt, ob ein Headereintrag in WSDL spezifiziert wird oder nicht. Sind bei der Erstellung der WSDL-Beschreibung eines Webservices Headereinträge bekannt, sollten sie spezifiziert werden, damit Webclients über diese Details der Schnittstelle informiert werden.

Der Ausschnitt aus einer WSDL-Beschreibung in Abbildung 67 zeigt ein Beispiel, in dem ein Headereintrag deklariert wird. Es handelt sich um eine Erweiterung der Webkomponente für eine Internetzeitung. Die Operation `Get` wurde so erweitert, dass ihr Request stets einen Headereintrag `nwst:TipURL` haben muss. Abbildung 68 zeigt ein Beispiel für einen solchen Request.

Abbildung 67 zeigt den Ausschnitt der Definition der gebundenen Schnittstelle der erweiterten Operation `Get`. In `wsdl:operation/wsdl:input` wird der Request der Operation beschrieben. Neben der Beschreibung des SOAP-Bodys mit dem Kindelement `soap12:body` gibt es zusätzlich das Kindelement `soap12:header`, das den Headereintrag beschreibt. Mit seinen Attributen `message` und `part` wird auf einen Part in einer Nachrichtendeklaration verwiesen. Auch die Nachrichtendeklaration ist in Abbildung 67 dargestellt. Sie verweist auf das in XML-Schema deklarierte Element `nwst:TipURL`, das den Headereintrag bildet.

```
<wsdl:message name="TipURLHeader">
  <wsdl:part name="TipURLHeader" element="nwst:TipURL" />
</wsdl:message>

<wsdl:binding name="NewsServiceBinding"
              type="nwsi:NewsServicePortType" >
  <!-- ... -->

  <wsdl:operation name="Get" >
    <soap12:operation />
    <wsdl:input>
      <soap12:body use="literal" />
      <soap12:header use="literal"
                     message="nwsi:TipURLHeader"
                     part="TipURLHeader"  />
    </wsdl:input>
    <wsdl:output><soap12:body use="literal" /></wsdl:output>
    <wsdl:fault name="InvalidSessionID">
      <soap12:fault name="InvalidSessionID" use="literal" />
    </wsdl:fault>
    <wsdl:fault name="InvalidMessageID">
      <soap12:fault name="InvalidMessageID" use="literal" />
    </wsdl:fault>
    <wsdl:fault name="MessageLocked">
      <soap12:fault name="MessageLocked" use="literal" />
    </wsdl:fault>
  </wsdl:operation>

  <!-- ... -->
</wsdl:binding>
```

Abbildung 67: Definition der Operation `Get` in gebundener Schnittstelle mit Headereintrag

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope"
        xmlns:nwst="http://example.org/NewsService020917/Types">
  <env:Header>
    <nwst:TipURL env:mustUnderstand="true">
      tip://doolin/?OleTx-0369c725-b1f0-4ee3-8df2-47c8a983d285
    </nwst:TipURL>
  </env:Header>
  <env:Body>
    <nwst:Get>
      <nwst:SessionID>8CCF4AB6-FF9E-451d-8675-1B4F56B05296</nwst:SessionID>
      <nwst:MessageIDsSequence>
        <nwst:MessageID>80AF91EA-32EB-4691-9ED5-D0BF047B9424</nwst:MessageID>
        <nwst:MessageID>AD6F371A-ABE9-41f5-8362-AAE7640F1DDD</nwst:MessageID>
      </nwst:MessageIDsSequence>
    </nwst:Get>
  </env:Body>
</env:Envelope>
```

Abbildung 68: Beispiel für Request der Operation `Get` mit Headereintrag `nwst:TipURL`

Auf diese Art kann festgelegt werden, dass ein Headereintrag Teil einer SOAP-Nachricht sein muss. Ein Webclient benötigt diese Information, damit er den betreffenden Headereintrag im Request mitsenden kann oder im Falle eines Responses den Headereintrag erwartet. Im Falle des Responses benötigt der Webclient zusätzlich die Information, ob er den Headereintrag ignorieren darf. Das wird ihm im Response mit dem Attribut `env:mustUnderstand` angezeigt. Da das Teil der Schnittstelle ist, sollte es auch in WSDL beschrieben werden.

Das ist möglich, da in XML-Schema spezifiziert werden kann, dass das Attribut `env:mustUnderstand` stets einen bestimmten Wert haben oder immer fehlen muss. Das Fragment eines Schemadokumentes in Abbildung 69 zeigt, wie das für den Headereintrag `nwst:TipURL` aus Abbildung 68 geschieht. Bei diesem Headereintrag muss das Attribut `env:mustUnderstand` stets vorhanden sein und den Wert `true` haben.

Hierzu wird in Abbildung 69 dem Headereintrag das Attribut `env:mustUnderstand` explizit zugefügt. In der Deklaration wird mit dem Attribut `fixed` festgelegt, dass der Wert stets `true` sein muss. Durch Zuweisung des Wertes `required` an das Attribut `use` wird das Vorhandensein des Attributes `env:mustUnderstand` vorgeschrieben. Ebenso könnte festgelegt werden, dass das Attribut `env:mustUnderstand` den Wert `false` haben muss. Wird dem Attribut `use` dabei der Wert `optional` gegeben, darf das Attribut `env:mustUnderstand` auch fehlen. Wird dem Attribut `use` der Wert `prohibited` zugewiesen, kann sogar das Vorhandensein des Attributes `env:mustUnderstand` verboten werden.

```
<xsd:complexType name="tipURL">
  <xsd:simpleContent>
     <xsd:extension base="xsd:anyURI">
      <xsd:attribute ref="env:mustUnderstand"
                     fixed="true" use="required" />
     </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>

<xsd:element name="TipURL" type="nwst:tipURL" />
```

Abbildung 69: Deklaration des Headereintrages `nwst:TipURL`

Aus der Tatsache, dass ein Headereintrag im Response stets das Attribut `env:mustUnderstand` mit dem Wert `true` hat, kann der Webclient schließen, dass er diesen Headereintrag interpretieren können muss, damit er die Operation erfolgreich verwenden kann. Umgekehrt wäre es für ihn eine wichtige Information, welche Headereinträge in Requests der Webservice nicht interpretieren kann. Solchen Headereinträgen darf er das Attribut `env:mustUnderstand` mit dem Wert `true` nicht zufügen. Für einzelne Headereinträge ist es möglich das zu deklarieren. Wie eben vorgestellt, müssen sie hierzu so deklariert werden, dass das Attribut `env:mustUnderstand` fehlen oder den Wert `false` haben muss.

Leider ist die Menge der möglichen Headereinträge jedoch unbegrenzt. Daher kann nicht jeder mögliche Headereintrag, den der Webservice nicht kennt, bei der Spezifikation eines Webservices angegeben werden. Möglich wäre es nur, die Headereinträge anzugeben, die dem Webservice bekannt sind. Für alle anderen müsste spezifiziert werden können, dass ein Webclient sie nur senden darf, wenn sie nicht das Attribut `env:mustUnderstand` mit dem Wert `true` enthalten, so dass sie der Webservice ignorieren kann. Das kann in WSDL jedoch nicht ausgedrückt werden.

Neben dieser Tatsache gibt es auch bei Headereinträgen unterschiedliche Beziehungen zwischen Werten in der gleichen oder zu anderen SOAP-Nachrichten, die nicht in WSDL spezifiziert werden können. So könnte z. B. der in Abbildung 68 gezeigte Headereintrag `nwst:TipURL` optional sein. Dann könnte gefordert werden, dass entweder dieser Headereintrag verwendet werden darf oder Sperren explizit im Bodyeintrag mit dem Element `nwst:Lock` gesetzt werden können, wie in Kapitel 6.1.3 vorgestellt, aber nicht beides gleichzeitig. WSDL erlaubt es jedoch nicht, eine solche Beziehung zwischen der Existenz eines Headereintrages und der eines optionalen Elementes im Bodyeintrag zu spezifizieren.

Weitere nicht in WSDL spezifizierbare Beziehungen mit Headereinträgen wurden bereits in dieser Arbeit angesprochen. Besonders wurde in Kapitel 6.2.4 darauf hingewiesen, dass in Fehlernachrichten keine Beziehungen zwischen Headereinträgen und Fehlercodes, Subfehlercodes oder Werten in Detaileinträgen spezifiziert werden können. Außerdem wurde in den Kapiteln 6.1.2 und 6.1.3 darauf hingewiesen, welche Begrenzungen es bei der Spezifikation von Beziehungen zwischen mehreren SOAP-Nachrichten gibt. Diese gelten selbstverständlich bei auch Beziehungen zwischen Werten in Headereinträgen.

In WSDL kann somit für einzelne Headereinträge spezifiziert werden, welche Struktur und welchen Typ sie haben. Das schließt mit ein, dass festgelegt werden kann, dass das Attribut `env:mustUnderstand` vorhanden sein und einen bestimmten Wert haben muss. Somit kann einem Webclient mitgeteilt werden, dass er bestimmte Headereinträge im Response interpretieren können muss. Ebenso lässt sich ihm mitteilen, dass er bestimmten Headereinträgen im Request nicht das Attribut `env:mustUnderstand` mit dem Wert `true` zufügen darf. Obwohl das für einzelne Headereinträge spezifiziert werden kann, ist es nicht möglich, gleiches für beliebige Headereinträge zu spezifizieren, insbesondere nicht für solche, die bei der Erstellung des WSDL-Dokumentes nicht bekannt waren. Neben dieser Tatsache lassen sich auch wieder viele Beziehungen zwischen Werten innerhalb der SOAP-Nachricht und mit Werten in anderen SOAP-Nachrichten nicht spezifizieren.

6.3 Fazit

Die Untersuchung in diesem Kapitel hat gezeigt, dass die Ausdrucksfähigkeit von WSDL begrenzt ist. Welche Teile einer Schnittstelle mit der Präzisierung aus Kapitel 5 mit WSDL beschreibbar sind, wurde am Anfang dieses Kapitels kurz zusammengefasst. Eine Schnittstelle wird als eine Menge von Operationen spezifiziert, von denen jede hauptsächlich durch zwei Nachrichtentypen beschrieben wird. Für einen Nachrichtentyp werden ein oder mehrere XML-Fragmente mit Hilfe von XML-Schema spezifiziert. So können dessen Struktur und die Typen der Blattelemente und Attribute festgelegt werden.

Über diese Beschreibung hinaus kann wenig spezifiziert werden. Insbesondere ist es nicht möglich, Beziehung zwischen Werten von Blattelementen und Attributen festzulegen. Die Kapitel 6.1 und 6.2 haben viele Beispiele gezeigt, bei denen Anforderungen an solche Beziehungen jedoch wichtige Bestandteile des Vertrages zwischen Webclient und Webservice sind. Daher sind diese Anforderungen auch Teil der Schnittstelle, können aber nicht formal in WSDL spezifiziert werden. Die fehlende Präzision von WSDL kann dazu führen, dass Webclient und Webservice zueinander nicht interoperabel sind, obwohl sie demselben WSDL-Dokument entsprechen.

7 Spezifikation mit XQuery-Ausdrücken über Traces

Um die in Kapitel 6 vorgestellten Anforderungen an Schnittstellen von Webservices formulieren zu können, für die das in WSDL nicht möglich ist, soll in diesem Kapitel ein geeignetes Spezifikationsverfahren entwickelt werden. Sein Name sei SXQT, was für „Specifications using XQuery expressions on Traces“ steht.

SXQT soll eine tragfähige Basis für die in Kapitel 2.7 grob umrissene automatische Validation sein. Bei der automatischen Validation wird für zwischen Webkomponenten ausgetauschte SOAP-Nachrichten geprüft, ob sie der Spezifikation entsprechen. SXQT-Spezifikationen sollten diese Prüfung möglichst einfach gestalten. Gleichzeitig muss es möglich sein, Schnittstellen von Webkomponenten so genau zu beschreiben, dass sichergestellt ist, dass Webkomponenten zueinander interoperabel sind, wenn sie die SXQT-Spezifikation erfüllen. Nur dann führt die automatische Validation beobachteter SOAP-Nachrichten gegen die SXQT-Spezifikation zu aussagefähigen Ergebnissen.

Ein weiteres Entwurfsziel ist, dass SXQT an Webkomponenten angepasst sein muss. Da Webkomponenten stark mit XML verbunden sind, sollten für SXQT zu XML gehörige Technologien berücksichtigt werden. Das hat zwei Seiten. SXQT muss erlauben, den Austausch von SOAP-Nachrichten, also XML-Dokumenten, zu beschreiben. Für den Umgang mit XML-Dokumenten sind also geeignete Mechanismen vorzusehen. Auf der anderen Seite sollte SXQT selbst Beschreibungsmittel verwenden, die im Bereich von XML üblich sind. Es muss beachtet werden, dass nicht jeder Entwickler von Webservices in der Verwendung formaler Beschreibungsmittel geschult ist. XML und dazugehörige Technologien sind jedoch in der Regel bekannt. Werden diese Technologien für SXQT verwendet, vereinfacht das den Entwicklern Spezifikationen zu erstellen und zu verstehen.

SXQT sollte nicht eine bestimmte Implementierung einer Webkomponente nahe legen. Davon unabhängig sollen nur die Schnittstellen beschrieben werden. Wie in Kapitel 2.5 festgestellt, sollte sich die Beschreibung möglichst direkt daran orientieren, wie die Kommunikation über das Internet stattfindet. Das ist von der Implementierung unabhängig.

7.1 Spezifikation von Entitäten

Als Fundament für die Entwicklung des Spezifikationsverfahrens SXQT soll von einem Spezifikationsverfahren von C.A.R. Hoare ausgegangen werden, dass in [Hoare 85] beschrieben wird. Es erlaubt das Verhalten von Entitäten im Sinne von beobachtbaren Ereignissen zu beschreiben. Solche Beschreibungen sind völlig unabhängig davon, wie die Entität realisiert ist. So können Webkomponenten unabhängig von ihrer Implementierung beschrieben werden.

In seinem Buch [Hoare 85] beschreibt Hoare primär die Sprache „Communicating Sequential Processes“ (CSP), die es mit Hilfe von Prozesstermen erlaubt, beliebige Entitäten zu beschreiben. Solche Modelle beschreiben zustandsbasiert, wie eine Entität implementiert werden könnte. Daher soll CSP nicht die Basis für Spezifikationsverfahren SXQT verwendet werden. Zusätzlich zu CSP beschreibt Hoare in [Hoare 85] jedoch auch ein Spezifikationsverfahren, das auf logischen Aussagen beruht. Es wird vor allem verwendet, um den Zusammenhang zu Prozesstermen zu zeigen, kann jedoch auch unabhängig von diesen angewendet werden. Dieses soll in diesem Kapitel zum Spezifikationsverfahren SXQT fortentwickelt werden. Dazu relevante Teile von [Hoare 85] werden im Folgenden zusammengefasst.

7.1.1 Entitäten und Ereignisse

Um seine Formalismen zu definieren, betrachtet Hoare Ereignisse, die sich auf Entitäten beziehen. Das Verhalten der Entitäten, die er als Objekte (objects) bezeichnet[32], soll beschrieben werden. In Beispielen verwendet Hoare häufig Entitäten, die reale Gegenstände sind. So wird häufig das Beispiel von Warenautomaten verwendet, u. a. eine einfache Variante, die nur eine Art von Schokolade verkauft. Dieses soll auch hier zur Illustration dienen.

Ereignisse (events) sind Aktionen, die sich auf eine Entität beziehen. Für die Modellierung der Entität werden nur die im Modell relevanten Ereignisse betrachtet. So gibt es für einen Warenautomaten in der Realität eine Vielzahl von Ereignissen. Im Modell könnten trotzdem nur zwei Arten von Ereignissen betrachtet werden, wie das Einwerfen einer Münze und das Ausgeben einer Tafel Schokolade.

Durch das Stattfinden von Ereignissen kommunizieren Entitäten miteinander. Der Warenautomat kommuniziert z. B. mit seiner Umgebung, die durch Käufer gebildet wird. Die Umgebung kann selbst als Entität betrachtet werden. Sie nimmt an den Ereignissen des Wartenautomaten Teil und kommuniziert so mit ihm.

Ähnliche Arten von Ereignissen fasst Hoare zu Ereignisklassen (event classes) zusammen. Ereignisklassen werden durch ihre Namen, die Ereignisnamen (event names), identifiziert. In gewisser Weise sind Ereignisklassen Mengen von Ereignissen. Hoare legt fest, dass alle Ereignisse einer Ereignismenge voneinander als ununterscheidbar angesehen werden. Um das Auftreten eines Ereignisses in Formalismen auszudrücken, reicht es daher den Namen der Ereignisklasse anzugeben. So kann vereinfachend der Ereignisname auch als Name von Ereignissen verwendet werden. Zur Modellierung des Warenautomatens könnte z. B. jedes Einwerfen einer Münze mit dem Ereignisnamen `coin` und jedes Ausgeben einer Tafel Schokolade mit dem Ereignisnamen `choc` bezeichnet werden.

Hoare legt fest, dass eine Entität im Modell eine festgelegte Menge von Ereignisklassen hat. Sie enthält die Ereignisklassen der Ereignisse, die für die Entität als relevant erachtet wurden. Die Menge ihrer Ereignisnamen bilden das Alphabet der Entität, das eine wichtige Eigenschaft von ihr ist. Der oben beschriebene Warenautomat mit den beiden Ereignisklassen `coin` und `choc` hätte z. B. das Alphabet `{coin,choc}`.

---

[32] Da der Begriff „Objekt“ im Kontext von objektorientierten Modellen bereits sehr präzise festgelegt ist, wird in dieser Arbeit statt dessen der Begriff „Entität“ verwendet.

Hoare verwendet eine Reihe von Annahmen über Ereignisse. Sie können mit den folgenden Stichworten zusammengefasst werden:

- Atomar
- Keine zeitliche Dauer
- Keine Gleichzeitigkeit
- Keine Betrachtung absoluter Zeiten
- Keine Kausalitäten

Ereignisse sind atomar. Sie finden also entweder vollständig statt oder überhaupt nicht. Sie geschehen augenblicklich, ohne eine zeitliche Dauer. Trotzdem können Aktionen beschrieben werden, die nicht Atomar sind oder eine zeitliche Ausdehnung haben, indem sie durch mehrere Ereignisse beschrieben werden. Z. B. kann der Anfang der Aktion und ihr Ende durch ein Ereignis beschrieben werden.

Obwohl Ereignisse keine zeitliche Dauer haben, könnten sie prinzipiell gleichzeitig stattfinden. Hoare verbietet das jedoch. Ist es wichtig, dass zwei Ereignisse gleichzeitig stattfinden, kann das dargestellt werden, indem sie zu einem einzelnen Ereignis verschmolzen werden. Ansonsten muss erlaubt werden, dass beide Ereignisse hintereinander in beiden Reihenfolgen auftreten.

Hoare verzichtet auf die Beschreibung absoluter Zeiten. Er begründet das mit Vereinfachungen beim Entwurf von Modellen und bei Schlussfolgerungen. Außerdem wird so ermöglicht, Modelle auf Systeme unterschiedlicher Geschwindigkeit anzuwenden. Somit spielt für das Auftreten von Ereignissen nur ihre Reihenfolge eine Rolle.

Eine weitere Vereinfachung von Hoare ist der Verzicht auf die Betrachtung von Kausalitäten, also die Frage, welche Entität ein Ereignis auslöst. So wird im Beispiel des Warenautomaten die Ausgabe der Schokolade vom Warenautomaten selbst herbeigeführt, während der Einwurf der Münze eher auf eine Handlung des Kunden zurückgeht. Hoare stellt fest, dass der Verzicht der Beschreibung von Kausalitäten zu einer Vereinfachung von Theorie und Anwendung führt.

7.1.2 Prozessterme

Prozessterme und die auf ihnen beruhende Prozessalgebra bilden den Hauptteil von [Hoare 85]. Zwar werden sie nicht für das in Spezifikationsverfahren SXQT benötigt. Ein Grundverständnis ist aber für nachfolgende Untersuchungen zweckmäßig.

Basis für Prozessterme, die Hoare als Prozesse bezeichnet[33], ist das in Kapitel 7.1.1 vorgestellte Modell von Ereignissen und Entitäten. Prozessterme sind ein zustandsbasierter Formalismus, um das Verhalten von Entitäten zu beschreiben. Dabei wird ihr Verhalten im Sinne erlaubter Reihenfolgen von Ereignissen beschrieben, die sich auf die Entität beziehen.

Um das Verhalten zu beschreiben, wird vom Alphabet der Entität ausgegangen, das auch das Alphabet des Prozessterms ist. Der Prozessterm beschreibt die Entität in einem bestimmen Zustand. Er legt fest, welche Ereignisse des Alphabetes in seinem Zustand stattfinden dürfen. Findet dann eines dieser Ereignisse statt, ändert das in der Regel den

[33] Unter einem „Prozess“ wird typischerweise ein konkreter Vorgang verstanden. Hoare versteht darunter jedoch ein Muster für einen Vorgang. Daher wird wie auch in [Valk 01] in dieser Arbeit statt von Prozessen von Prozesstermen gesprochen.

Zustand der Entität. Sie wird durch einen neuen Prozessterm beschrieben, der seinerseits festlegt, welche Ereignisse nun stattfinden dürfen. Durch die erlaubten Ereignisse und die mit ihnen verbundenen Wechsel zu anderen Prozesstermen werden die erlaubten Reihenfolgen der Ereignisse und damit das Verhalten der Entität beschrieben.

7.1.3 Traces

Anders als Prozessterme sind Traces eine wichtige Basis für das in dieser Arbeit entwickelte Spezifikationsverfahren SXQT. Sie stellen eine Art Beispielfolge von Ereignissen dar, die in einer Entität stattgefunden haben. Die Idee ist, dass ein Beobachter (z. B. ein Mensch) eine Entität beobachtet und die auftretenden Ereignisse notiert[34]. Beobachtete Ereignisse stammen stets aus dem Alphabet der Entität. Notiert werden ihre Ereignisnamen. Die Aufzeichnungen des Beobachters sind der Trace. Beginnt ein Beobachter die Entität zu beobachten, ist der Trace zunächst leer. Mit jedem beobachteten Ereignis wächst der Trace. Trotzdem bleibt er zu jedem Zeitpunkt endlich.

Der Beobachter benötigt weder ein Verständnis von Zeit, noch muss er verstehen, was die einzelnen Ereignisse bedeuten oder dass bestimmte Ereignisse logisch zusammengehören. Er notiert lediglich ihre Reihenfolge. Dabei wird Gleichzeitigkeit von Ereignissen dadurch ausgeschlossen, dass der Beobachter nur in der Lage ist, die Ereignisse hintereinander zu notieren. Gegebenenfalls muss er eine der Reihenfolgen wählen.

Formal versteht Hoare unter einem Trace eine endliche Sequenz von Ereignisnamen, mit der die Ereignisse aufgezeichnet werden, die bis zu einem bestimmten Zeitpunkt mit einer Entität stattgefunden haben[35].

Hoare notiert Traces als kommaseparierte Sequenzen der Ereignisnamen, die in spitzen Klammern umschlossen werden. Der leere Trace wird als `<>` notiert. Beispiele für Traces des in Kapitel 7.1.1 angesprochenen Warenautomatens sind neben dem leeren Trace die Traces `<coin>`, `<coin,choc>` sowie `<coin,choc,coin,choc,coin>`.

Um in mathematischen Formalismen mit Traces umgehen zu können, definiert Hoare eine Reihe von Operationen auf Traces, die als Traceoperationen bezeichnet werden sollen. Sie werden auch für Spezifikationen benötigt. Da Traces Sequenzen (von Ereignisnamen) sind, handelt es sich bei den meisten Traceoperationen einfach um Operationen für Sequenzen. Sie könnten auch unabhängig von Traces verwendet werden. Der folgende Absatz zeigt kurz einige Beispiele. Detaillierter werden die Traceoperationen von Hoare noch in Kapitel 7.6.3 behandelt.

Die Operation `Concatenation`[36] `s^t` bezeichnet einen Trace, der erst die Ereignisnamen aus dem Trace `s` und danach die aus dem Trace `t` enthält. Mit der Restriktion `t↑A` wird ein Trace erzeugt, der nur die Ereignisnamen aus einem Trace `t` in gleicher Reihenfolge enthält, die im Alphabet `A` enthalten sind. Zerlegen lässt sich ein Trace `t` mit

---

[34] Hoare verwendet in [Hoare 85] das Bild, dass statt Entitäten Prozessterme beobachtet werden. Ein Beobachter kann jedoch nicht dieses algebraische Modell beobachten, sondern lediglich die Entität selbst. Daher wird hier von der Beobachtung von Entitäten gesprochen.

[35] Hoare definiert Traces in [Hoare 85] auf Seite 41 wie folgt: „*A trace of the behavior of a process is a finite sequence of symbols recording the events in which the process has engaged up to some moment in time.*" Die Definition im Text ist der von Hoare ähnlich, sie ist jedoch gemäß Fußnote 34 angepasst und der allgemeine Begriff „Symbols" durch den spezielleren „Ereignisnamen" ersetzt.

[36] Hoare bezeichnet diese Operation mit dem Namen `Catenation`. Da er jedoch eine weitere Operation dieses Namens einführt (siehe Kapitel 7.6.3.7), wird in dieser Arbeit der Name `Concatenation` verwendet.

den Operationen `Head` t_0 und `Tail t'`, wobei t_0 den ersten Ereignisnamen in `t` bezeichnet und `t'` den Trace `t` ohne das erste Element. `Head` ist ein Beispiel für eine Traceoperation, die keinen Trace liefert. Sie liefert einen Ereignisnamen. Andere Operationen liefern auch Zahlen oder sind Prädikate. Z. B. liefert `#t` die Anzahl der Ereignisnamen im Trace `t`. Das Prädikat `s≤t` prüft, ob der Trace `s` der Anfang des Traces `t` ist.

7.1.4 Spezifikationen

Traces und die Möglichkeit, über sie Prädikate zu formulieren, sind die Basis für Spezifikationen über Entitäten, die Hoare vorstellt. Unter einer Spezifikation eines Produktes versteht Hoare sehr allgemein eine Beschreibung wie sein Verhalten beabsichtigt war. Diese Absicht wird als Prädikat, also als logische Aussage formuliert, die für jeden beobachtbaren Aspekt des Produktes eine freie Variable enthält. Es wird erwartet, dass für jede beobachtete Belegung der freien Variablen die logische Aussage zutrifft.

Als allgemeines Beispiel verwendet Hoare die Spezifiktion eines elektronischen Verstärkers mit einer Spannungsverstärkung von 10, bei dem bei einer Eingangsspannung `v` zwischen 0 und 1 Volt die Toleranz der Ausgangsspannung `v'` weniger als 1 Volt beträgt[37]. Hoare verzichtet in seiner Spezifikation auf Einheiten. Die Spezifikation ist durch den logischen Ausdruck `(0≤v≤1 ⇒|v'-10*v| ≤ 1)` beschrieben.

Sollen auf diese Art Entitäten spezifiziert werden, bei denen die Reihenfolge ihrer Ereignisse relevant sind, und die durch Prozessterme beschrieben werden können, dann ist die offensichtlichste relevante Beobachtung der Trace der Ereignisse, die bis zu einem bestimmten Zeitpunkt stattgefunden haben. Eine Spezifikation, die sie beschreibt, muss somit eine freie Variable für den Trace enthalten. Daneben kann es für andere beobachtbare Größen weitere freie Variablen geben. Um die logischen Aussagen für solche Spezifikationen zu schreiben, verwendet Hoare eine Prädikatenlogik, wobei Traceoperationen verwendet werden.

Diese Vorgehensweise soll anhand von Beispielen veranschaulicht werden, die sich erneut auf Warenautomaten mit dem Alphabet `{coin,choc}` beziehen und aus [Hoare 85] stammen. Für Warenautomaten könnte spezifiziert werden, dass sie nie mehr Tafeln Schokolade ausgeben dürfen als Münzen eingeworfen wurden. Damit darf die Anzahl der Ereignisse `choc` in einem beobachteten Trace `tr` nie größer sein als die Anzahl der Ereignisse `coin`. Mit den Traceoperationen aus Kapitel 7.1.3 lässt sich das als logischer Ausdruck `(#(tr↑{choc}) ≤ #(tr↑{coin}))` formulieren. Darin bezeichnet `(tr↑{coin})` einen Trace, der aus `tr` nur die Ereignisse `coin` enthält. Daher ist `#(tr↑{coin})` die Anzahl dieser Ereignisse in `tr`. Analog bezeichnet `#(tr↑{choc})` die Ereignisse `choc` im Trace. Diese muss stets kleiner oder gleich der Anzahl der Ereignisse `coin` sein.

Eine weitere Anforderung könnte sein, dass in den Warenautomaten keine Münzen eingeworfen werden können, wenn bereits für eine Tafel Schokolade bezahlt, diese aber noch nicht ausgegeben wurde. Dann darf sich das Ereignis `coin` nur einmal häufiger im Trace befinden als das Ereignis `choc`. Als logischer Ausdruck lässt sich das schreiben: `(#(tr↑{coin}) ≤ #(tr↑{choc}) + 1)`.

Um zu spezifizieren, dass mehrere Spezifikationen gelten sollen, müssen sie zu einer komplexeren Spezifikation zusammengefasst werden, indem sie mit der Konjunktion

[37] Dieses Beispiel befindet sich in [Hoare 85] auf Seite 58.

(logisches Und) verknüpft werden. Die beiden eben beschriebenen Anforderungen können zusammengefasst formuliert werden zu:
`((#(tr↑{choc}) ≤ #(tr↑{coin})) ∧ (#(tr↑{coin}) ≤ #(tr↑{choc}) + 1))`.

Hoare notiert die Tatsache, dass ein Produkt `P` eine Spezifikation `S` erfüllt, formal als `P satisfies S` oder verkürzt als `P sat S`. Das bedeutet, dass für jede Beobachtung von `P` das Prädikat `S` gilt. Ist das Produkt durch einen Prozessterm `P′` beschrieben, bedeutet das, dass für jeden beobachteten Trace `tr`, die Spezifikation `S(tr)` zutreffen muss. Da `traces(P′)` für die Menge aller Traces von `P′` bezeichnet, muss also gelten: `∀tr: (tr ∈ traces(P′) ⇒ S(tr))`. Hoare stellt vor, wie sich das für Prozessterme nachweisen lässt.

Mit Hilfe logischer Aussagen, deren Ausdrücke eine freie Variable `tr` für beobachtete Traces haben, lassen sich also Entitäten spezifizieren, die mit Prozesstermen beschrieben werden. Hoare stellt jedoch fest, dass das nur für deterministische Entitäten gilt. Das sind Entitäten, für die jede getroffene Entscheidung über ihr Verhalten als Ereignis beobachtbar ist. Für solche Entitäten gilt, dass ihr Zustand eindeutig durch ihren Anfangszustand und den beobachteten Trace beschrieben wird.

Anders ist es bei nichtdeterministischen Entitäten. Auch nach Beobachtung des gleichen Trace können sie unterschiedliche Zustände angenommen haben, wobei einem Beobachter die Entscheidung verborgen bleibt. Das kann dazu führen, dass die gleiche Entität, nachdem der gleiche Trace beobachtet wurde, in einigen Fällen ein Ereignis zulässt, in anderen jedoch nicht. Bietet die Umgebung nach Beobachtung des gleichen Traces stets die gleichen Ereignisse an, kann das in einigen Fällen zu einem Deadlock führen, in anderen nicht. Das lässt sich mit Spezifikationen, die nur auf Traces beruhen nicht ausdrücken.

Daher führt Hoare in Spezifikationen für nichtdeterministische Prozessterme eine zweite freie Variable `ref` für sogenannte Refusals ein. Ein Refusal ist eine Menge von Ereignisklassen, die einer Entität in ihrem Zustand von der Umgebung angeboten wird, so dass die Entität sofort in einen Deadlock gerät, also nie ein Ereignis stattfinden kann. In Spezifikationen `S(tr,ref)` ist `ref` die zweite freie Variable. Sie bezeichnet eine beobachtete Menge von Ereignisklassen, nach der die Entität angeboten wird, nachdem der Trace `tr` beobachtet wurde, so dass die Entität sofort in einen Deadlock gerät.

Hoare stellt auch für Spezifikationen mit Refusals vor, wie bewiesen werden kann, dass Prozessterme sie erfüllen. Im Spezifikationsverfahren SXQT, das in dieser Arbeit entwickelt wird, spielen Refusals jedoch keine Rolle. Der Grund ist, dass Refusals ohne eine Modellvorstellung der Entität und ohne ein Verständnis absoluter Zeit nicht beobachtet werden können.

Ein Beobachter kann nur ein stattfindendes Ereignis beobachten und notieren. Es ist aber nicht möglich zu beobachten, dass in der Zukunft kein Ereignis stattfinden wird. Damit kann er auch kein Deadlock beobachtet werden, weil nie auszuschließen ist, dass später noch ein Ereignis stattfindet. Etwas Vergleichbares wäre nur mit einem Verständnis von absoluter Zeit möglich. Dann könnte spezifiziert werden, dass ein Ereignis nach einer bestimmten Zeit stattfinden muss und der Beobachter könnte nach Ablauf dieser Zeit feststellen, dass das Ereignis nicht stattgefunden hat. Wie in Kapitel 7.1.1 festgestellt, legt Hoare jedoch fest, dass absolute Zeiten nicht mitbetrachtet werden. Damit kann auch das Ablaufen einer bestimmten Zeit nicht beobachtet werden und somit auch keine Refusals. Aus diesem Grund soll für das Spezifikationsverfahren SXQT auf die Betrachtung von Refusals verzichtet werden.

7.2 Übertragung auf Webservices

Sollen Spezifikationen mit einer freien Variable für Traces verwendet werden, um Webservices zu spezifizieren, ist es zunächst erforderlich, die von Hoare eingeführten Begriffe auf Webkomponenten zu übertragen. Das soll in diesem Kapitel 7.2 geschehen, bevor in Kapitel 7.3 weiter auf Spezifikationen von Webservices eingegangen wird.

7.2.1 Entitäten, Ereignisse

Webkomponenten in ihren beiden Rollen als Webservice und Webclient sind die Entitäten, die spezifiziert und beobachtet werden. Dabei ist insbesondere die Rolle des Webservices wichtig, wenn wie bei WSDL aus dieser Sicht spezifiziert werden soll. Zusätzlich kann auch die Umgebung des Webservices als Entität aufgefasst werden. Das wären alle Webclients gemeinsam, die auf den Webservice zugreifen.

In Hoares Modell kommunizieren Entitäten miteinander mit Hilfe von Ereignissen, die sich beobachten lassen. Bei Webkomponenten kann vor allem beobacht werden, dass eine SOAP-Nachrichten gesendet oder empfangen wurde. Daher sind das die für Webkomponenten relevanten Ereignisse.

Wie in Kapitel 7.1.1 vorgestellt, legt Hoare für Ereignisse in seinem Modell fest, dass sie atomar sind, keine zeitliche Dauer haben, nicht gleichzeitig sein können und dass für sie weder absolute Zeiten noch Kausalitäten betrachtet werden sollen. Diese Eigenschaften können bis auf das Fehlen von Kausalitäten auch auf Beobachtungen von SOAP-Nachrichten angewendet werden.

Die Beobachtung einer SOAP-Nachricht ist sowohl atomar als auch ohne zeitliche Ausdehnung. Das ist jedoch nur eine Annahme für das Modell. Selbstverständlich benötigt ein Webkomponente einige gewisse Zeit, um eine SOAP-Nachricht zu senden oder zu empfangen. Ebenso wird auch eine gewisse Zeit zur Übertragung einer SOAP-Nachricht von SOAP-Sender zum SOAP-Empfänger benötigt. Außerdem sind auch beide Vorgänge auch nicht atomar. Z. B. könnte das Senden einer SOAP-Nachricht durch einen Fehler unterbrochen werden. Für eine Beobachtung der Nachricht können jedoch beide Eigenschaften angenommen werden. Dazu muss ein Beobachter die SOAP-Nachricht nur zu einem Zeitpunkt beachten und auch nur dann, wenn das Senden bzw. Empfangen erfolgreich war.

Auf diese Art kann auch die Gleichzeitigkeit verboten werden. Ein Beobachter muss den Austausch von Nachrichten stets hintereinander beachten. Eine gleichzeitige Wahrnehmung ist nicht erlaubt. Anders als in Hoares Modell ist es hier jedoch nicht möglich, die Gleichzeitigkeit von Ereignissen durch Verschmelzung zu einem auszurücken. Das würde bedeuten, dass mehrere SOAP-Nachrichten zu einer verschmolzen werden müssten, was z. B. dem verwendeten Request-/Response-Kommunikationsmuster widersprechen könnte. Diese Tatsache ist jedoch keine große Einschränkung, weil der gleichzeitige Austausch von SOAP-Nachrichten über eine Netzwerkverbindung in vielen Fällen ohnehin technisch nicht möglich ist.

Um das Modell einfach zu halten, wird auch von Hoares Modell übernommen, dass keine absoluten Zeiten betrachtet werden. Es wird also nicht aufgezeichnet, wann eine SOAP-Nachricht beobachtet wurde. Nur die Reihenfolge der beobachten SOAP-Nachrichten hat eine Relevanz.

Hoares Annahme der fehlenden Kausalität erscheint im Kontext von Webkomponenten wenig sinnvoll, weil die Ereignisse die Beobachtung ausgetauschter SOAP-Nachrichten sind. Hier gibt es stets einen SOAP-Sender und einen SOAP-Empfänger. Diese beiden Rollen legen es nahe, dem SOAP-Sender als Auslöser des Ereignisses anzusehen. Der

SOAP-Empfänger kann sich nicht dagegen „wehren“, dass er eine SOAP-Nachricht empfängt. In Hoares Modell ist diese Möglichkeit für eine Entität stets gegeben. Daher soll diese spezielle Kausalität in dieser Arbeit betrachtet werden.

Ereignisse fasst Hoare zu Ereignisklassen zusammen und legt dabei fest, dass alle Ereignisse einer Ereignisklasse voneinander ununterscheidbar sind. Mit diesen Annahmen beschreibt Hoare auch die Kommunikation mit Hilfe von Nachrichten über Kanäle. Wegen der Ununterscheidbarkeit muss Hoare für jede mögliche Nachricht, die unterschieden werden soll, eine eigene Ereignisklasse verwenden. Er definiert geeignete Notationen, um mit den vielen Ereignisklassen überhaupt umgehen zu können.

Um Schnittstellen von Webkomponenten zu spezifizieren, müssen über sie ausgetauschte SOAP-Nachrichten beschrieben werden. Die von Hoare vorgestellten Nachrichten sind im Vergleich zu SOAP-Nachrichten jedoch sehr einfach. SOAP-Nachrichten haben eine komplexe Struktur. Innerhalb der Struktur unterscheiden sich zwei SOAP-Nachrichten häufig nur durch enthaltene Werte. Mit der Vorgehensweise von Hoare würde eine Vielzahl von Ereignisklassen benötigt sowie weitere Formalismen, um mit ihnen umzugehen.

Daher soll in dieser Arbeit anders vorgegangen werden. Beobachtungen ähnlicher SOAP-Nachrichten werden zu einer Ereignisklasse zusammengefasst. Ihre Ereignisse sind unterscheidbar. Für jede Operation gibt es vor allem zwei Ereignisklassen, eine für ihre Requests und eine für ihre Responses. Zusätzlich kann es Fehlernachrichten geben. Dann unterscheiden sich die Ereignisse einer Ereignisklasse vor allem in den Werten von Attributen und Blattelementen.

Interessant ist, diese Definition der Ereignisklassen mit der Definition von Schnittstellen in WSDL zu vergleichen. In WSDL werden in der Beschreibung abstrakter sowie gebundener Schnittstellen jeweils Nachrichtentypen beschrieben, die diesen Ereignisklassen entsprechen. Abbildung 67 auf Seite 113 zeigt ein Beispiel für eine gebundene Schnittstelle. Mit dem Element `wsdl:binding/wsdl:operation/wsdl:input` werden dort die Requests einer Operation beschrieben. Das definiert die Ereignisklasse der Requests dieser Operation. Analog wird die Ereignisklasse ihrer Responses mit dem Element `wsdl:binding/wsdl:operation/wsdl:output` beschrieben, Fehlernachrichten außerdem mit `wsdl:binding/wsdl:operation/wsdl:fault`. Auch bei der Definition abstrakter Schnittstellen findet sich diese Dreiteilung.

Um Ereignisklassen zu bezeichnen, verwendet Hoare Ereignisnamen. Diese werden in Formalismen angegeben, um die Ereignisklasse zu bezeichnen. Da in Hoares Modell Ereignisse einer Ereignisklasse ununterscheidbar sind, kann auch das Auftreten eines Ereignisses mit Ereignisnamen notiert werden. In diese Arbeit sind Ereignisse einer Ereignisklasse jedoch unterscheidbar. Daher ist letzteres nicht möglich. Trotzdem bleibt der Ereignisname wichtig, um die Ereignisklasse zu bezeichnen. Doch wie sollte er gewählt werden?

Ereignisnamen sollten einen Zusammenhang mit der SOAP-Nachricht oder der Beschreibung des Webservices in WSDL haben. Der Zusammenhang mit der SOAP-Nachricht ist vorzuziehen. Einerseits sind Ereignisnamen dann auch ohne die Verwendung von WSDL sinnvoll. Andererseits könnte so der Ereignisname und damit auch die Ereignisklasse direkt aus dem Inhalt der SOAP-Nachricht abgeleitet werden.

Bleibt die Frage, was in der SOAP-Nachricht als Ereignisname geeignet wäre. Dieser Teil müsste für jede SOAP-Nachricht einer Ereignisklasse gleich sein. Für einen Präzisierungsstandard für SOAP und diese Arbeit wurde in Kapitel 5.2.4 gefordert, dass der Name des Bodyeintrages im Request dem der Operation sein muss und im Response die

Konkatenation aus dem Namen der Operation mit der Zeichenkette „`Response`". Außerdem wurde in Kapitel 5.3.4 das Überladen von Operationen verboten. Damit beschreibt der Name des Bodyeintrages eindeutig die Operation und durch den Suffix „`Response`" können die Ereignisklassen für Request und Response unterschieden werden[38].

Das legt nahe, den Namen des Bodyeintrages als Ereignisnamen zu verwenden. Zu beachten ist, dass solche Namen qualifizierte Namen sind, bei denen also ein Präfix einen Namensraum bezeichnet. Diese Tatsache hat bei SOAP den Vorteil, dass die Namen unabhängig voneinander, weltweit eindeutig definiert werden können, was mit eindeutigen Bezeichnern (URIs) für Namenräumen erreicht wird. Durch die Verwendung von qualifizierten Namen für Ereignisnamen ist dieser Vorteil auch für Ereignisnamen gegeben.

Die Verwendung des Namens des Bodyeintrages als Ereignisname führt bei Fehlernachrichten dazu, dass alle Fehlernachrichten den gleichen Ereignisnamen haben, nämlich `env:Fault`. Das liegt daran, dass Fehlernachrichten im SOAP-Body stets eine festgelegt Struktur haben, wie in Kapitel 3.4 beschrieben. Da Ereignisnamen Ereignisklassen eindeutig identifizieren, befinden sich somit alle Fehlernachrichten in einer Ereignisklasse.

Die Menge der Ereignisklassen einer Entität bezeichnet Hoare als ihr Alphabet. Formal schreibt er es als Menge von Ereignisnamen. Da Ereignisse einer Ereignisklasse in seinem Modell ununterscheidbar sind, beschreibt er damit die Menge der unterscheidbaren Ereignissen. Um die Menge von unterscheidbaren Ereignissen von Webkomponenten zu beschreiben, reicht die Menge der Ereignisklassen bzw. der Ereignisnamen nicht aus. Hier muss das Alphabet als die Menge der SOAP-Nachrichten definiert werden, die von der Webkomponenten ausgetauscht werden können. Es wird mit WSDL unter Verwendung von XML-Schema festgelegt, wie in Kapitel 4 beschrieben.

7.2.2 Traces

Ebenso wie beim Alphabet reicht es auch beim Trace nicht aus, im Kontext von Webservices nur die Ereignisnamen zu notierten. In Hoares Modell reicht es, weil Ereignisse einer Ereignisklasse ununterscheidbar sind. Der Beobachter einer Entität kann dort alle beobachteten Ereignisse als Ereignisnamen notieren, wobei der Trace formal die Sequenz dieser Ereignisnamen ist. Damit ist die gesamte Beobachtung notiert.

Da bei Webservices die Ereignisse einer Ereignisklasse unterscheidbar sind, muss mehr als nur der Ereignisname notiert werden. Zur Beobachtung gehört zumindest der Inhalt der beobachteten SOAP-Nachricht. Zusätzlich können weitere Informationen ergänzt werden, die mit dem Ereignis beobachtbar sind. Beobachtbar ist zumindest die Richtung, in der die SOAP-Nachricht ausgetauscht wurde und welche SOAP-Nachrichten hintereinander auf der gleichen Netzwerkverbindung ausgetauscht wurden und damit zu einer Operation gehören können.

Es bleibt die Frage, in welcher Form solche Ereignisse notiert werden sollen. Bei Ereignisnamen, die letztlich Zeichenketten sind, ist das offensichtlich. Ebenso liegt es nahe, SOAP-Nachrichten direkt als XML-Dokumente zu notieren, was es auch erlaubt, sie als

[38] Damit diese gilt, muss genaugenommen verboten werden, dass der Name einer Operation mit der Zeichenkette „`Response`" endet. Sonst könnte es z. B. eine Operation mit Namen `A` geben und eine zweite mit Namen `AResponse`. Dann hätte der Bodyeintrag im Response der ersten Operation den gleichen Namen wie der Request der zweiten. Diese Feinheit soll hier jedoch ignoriert werden.

XML-Fragmente in andere XML-Dokumente einzubetten. Damit könnte die gesamte Beobachtung des Ereignisses als XML-Fragment gespeichert werden, das die SOAP-Nachricht und weitere beobachtete Informationen enthält.

Zu diesem Zweck soll in Traces für jede Beobachtung ein Element `tra:Message` (Message-Element) verwendet werden, wobei das Präfix `tra` den Namensraum bezeichnet, der Definitionen für Traces enthält[39]. Das Message-Element hat die SOAP-Nachricht als Kindelement. Weitere Informationen zur Beobachtung könnten ihm entweder als weitere Kindelemente oder als Attribute zugefügt werden. Nachfolgend wurden Attribute verwendet. Abbildung 70 zeigt ein XML-Fragment, das auf diese Art ein Ereignis beschreibt. Es handelt sich um die Beobachtung eines Responses, der bereits in Abbildung 55 auf Seite 102 gezeigt wurde.

```
<tra:Message xmlns:env="http://www.w3.org/2002/06/soap-envelope"
       xmlns:tra="http://ti5.tu-harburg.de/venzke/20021015/traces"
       xmlns:nwst="http://example.org/NewsService020917/Types"
       to="Client" operation="1">
  <env:Envelope>
    <env:Body>
      <nwst:LoginResponse>
        <nwst:SessionID>8CCF4AB6-FF9E-451d-8675-1B4F56B05296</nwst:SessionID>
      </nwst:LoginResponse>
    </env:Body>
  </env:Envelope>
</tra:Message>
```

Abbildung 70: Beispiel für XML-Fragment, das die Beobachtung eines Ereignisses beschreibt

In Abbildung 70 ist zu erkennen, dass dem Message-Element die Attribute `to` und `operation` zugefügt wurden. Das Attribut `operation` dient dazu, Request und Response eines Operationsaufrufes einander zuzuordnen. Diese Zuordnung ist wichtig, weil sich erst aus ihr die Bedeutung der beiden SOAP-Nachrichten ergibt. Daher wird auch in den Anforderungen zu SOAP 1.2 in [Apparao 02] festgestellt, dass es für Fälle, bei denen die Protokollbindung von SOAP diese Zuordnung nicht ermöglicht, in SOAP 1.2 Konventionen geben muss, die das ermöglichen. Die Zuordnung kann z. B. durch Verwendung von Headereinträgen geschehen, die innerhalb der SOAP-Nachricht den Operationsaufruf identifizieren.

In Kapitel 5.2.5 wurde für diese Arbeit jedoch festgelegt, dass nur das Protokoll HTTP zum Austausch von SOAP-Nachrichten verwendet werden darf. Dieses Protokoll erlaubt eine Zuordnung von Request und Response, indem es beide nacheinander auf der gleichen Netzwerkverbindung überträgt. (Siehe Kapitel 3.10.) Damit ist es nicht erforderlich, eine Zuordnung von Request zu Response in der SOAP-Nachricht selbst auszudrücken.

Als Beobachtung könnte der Beobachter daher notieren, welche SOAP-Nachrichten über die gleiche Netzwerkverbindung ausgetauscht wurden. HTTP 1.1 erlaubt es jedoch, die gleiche Netzwerkverbindung nacheinander für mehrere Request-/Response-Kommunikationen zu verwenden. Diese entsprechen unterschiedlichen Operationsaufrufen. Notiert nun der Beobachter, auf welchen Netzwerkverbindungen die SOAP-Nachricht ausgetauscht wurde, erschwert das, den Request und Response eines Operationsaufrufes zu erkennen. Um das zu vereinfachen, soll daher der Beobachter bereits notieren, zu welchem Operationsaufruf eine Nachricht gehört. Hierzu wählt er für jeden

[39] Das Präfix `tra` stehe für den Namensraum `http://ti5.tu-harburg.de/staff/venzke/20021015/traces`.

Operationsaufruf einen beliebigen Bezeichner und fügt ihn dem Message-Element als Attribut `operation` (`tra:Message/@operation`) zu. Erlaubt sind alle Werte des Datentyps `xsd:string`.

In Kapitel 5.2.1 wurde die Verwendung des Request-/Response-Kommunikationsmusters für diese Arbeit vorgeschrieben. Die beschriebene Vorgehensweise, Operationsaufrufe zu notieren, könnte jedoch auch für andere Kommunikationsmuster verwendet werden. Dem Operationsaufruf entspricht dann eine Kommunikation, also eine (kleine) Menge von SOAP-Nachrichten, die durch das Kommunikationsmuster beschrieben wird. Der eindeutige Bezeichner wird dann statt für den Operationsaufruf für die Kommunikation gewählt. Ihre Nachrichten können dann am gleichen Bezeichner im Attribut `operation` erkannt werden.

Welche SOAP-Nachricht der Request und welche der Response ist, kann der Beobachter daran erkennen, welche SOAP-Nachricht zuerst beobachtet wurde. Zusätzlich kann das auch an der Richtung erkannt werden, in der die SOAP-Nachrichten ausgetauscht wurden. Beim Request ist der Webservice stets der SOAP-Empfänger, während er beim Response der SOAP-Sender ist.

Auch diese Beobachtung sollte der Beobachter dem XML-Fragment zufügen, weil sie das Ereignis beschreibt. Hierzu soll in dieser Arbeit das Attribut `to` (`tra:Message/@to`) verwendet werden, für das die beiden Werte `Service` und `Client` erlaubt sind. Wird `Service` angegeben, handelt es um eine Beobachtung eines Requests, der SOAP-Empfänger ist also der Webservice. Ist der Wert dagegen `Client`, wurde ein Response beobachtet und damit ist der SOAP-Empfänger der Client.

Auch dieses Attribut könnte auf andere Kommunikationsmuster angewendet werden. Dann gibt es nicht unbedingt einen Request und einen Response. Die einzelnen SOAP-Nachrichten der Kommunikation können dann andere Bedeutungen haben. Trotzdem ist die Richtung, in der jede SOAP-Nachricht ausgetauscht wurde, definiert.

Um zu beschreiben, dass mehrere Webkomponenten miteinander kommunizieren, z. B. mehrere Webclients mit einem Webservice, könnten statt der Richtung die kommunizierenden Webkomponenten selbst angegeben werden. Da, wie in [Gudgin 02b] angegeben, Ressourcen im Internet einschließlich Webservices durch URIs bezeichnet werden, sollten solche auch für diesen Zweck verwendet werden. Im allgemeinen Fall wird es dabei erforderlich, in der Beobachtung nicht nur den Empfänger sondern auch den Sender zu notieren. Dazu würden zwei Attribute benötigt.

Sollte eine beobachtete SOAP-Nachricht so notiert werden, müsste der Beobachter stets ein Verständnis von den URIs der beteiligten Webkomponenten haben. Zum Verständnis des so notierten Ereignisses wäre es außerdem erforderlich zu wissen, welche URIs Webservices bezeichnen und welche Webclients. Daher soll der Einfachheit halber in dieser Arbeit nur das Attribut `to` mit seinen beiden Werten `Client` und `Service` verwendet werden.

Ein beobachtetes Ereignis wird also als Message-Element notiert, dass die beobachtete SOAP-Nachricht als Kindelement enthält. Das Attribut `operation` enthält einen Bezeichner für den Operationsaufruf, das Attribut `to` die Richtung, in der SOAP-Nachricht ausgetauscht wurde.

Innerhalb von Traces wird das verwendet. In Hoares Modell ist ein Trace eine Sequenz von beobachteten Ereignisse, die er als Ereignisname notiert. Auch in dieser Arbeit soll ein Trace eine Sequenz von beobachteten Ereignissen sein, wobei aber jede Beobachtung als Message-Element notiert wird. Damit ist ein Trace eine Sequenz von XML-

Fragmenten. Um eine solche Sequenz besser handhaben zu können, kann es von Vorteil sein, sie als vollständiges XML-Dokument zu notieren. Dazu kann die Sequenz von einem Element „umschlossen“ werden, so das die XML-Fragmente die Kindelemente bilden. In dieser Arbeit wird für Traces zu diesem Zweck ein Element mit Namen `tra:Trace` verwendet.

Das ist in Abbildung 71 angedeutet. Sie zeigt beispielhaft einen Trace, der vier beobachtete Ereignisse von zwei Operationsaufrufen enthält. Um das XML-Dokument in Abbildung 71 zu verkürzen, wurden dabei die Kindelemente des Elementes `env:Envelope` jeweils durch drei Punkte „...“ ersetzt. Das Element `tra:Trace` enthält für jedes beobachtete Ereignis ein Message-Element als Kindelement. Die Message-Elemente befinden sich im XML-Dokument in der Reihenfolge, in der die Ereignisse beobachtet wurden.

```
<?xml version="1.0" encoding="UTF-8"?>
<tra:Trace xmlns:env="http://www.w3.org/2002/06/soap-envelope"
    xmlns:tra="http://ti5.tu-harburg.de/venzke/20021015/traces">
  <tra:Message to="Service" operation="1">
    <env:Envelope> ... </env:Envelope>
  </tra:Message>
  <tra:Message to="Client" operation="1">
    <env:Envelope> ... </env:Envelope>
  </tra:Message>
  <tra:Message to="Service" operation="2">
    <env:Envelope> ... </env:Envelope>
  </tra:Message>
  <tra:Message to="Client" operation="2">
    <env:Envelope> ... </env:Envelope>
  </tra:Message>
</tra:Trace>
```

Abbildung 71: Beispiel für Trace, der vier Beobachtungen von Ereignissen enthält

7.2.3 Beschreibung von Traces mit XML-Schema

Naheliegend ist es, nach einer Beschreibung der Menge der beobachtbaren Traces mit XML-Schema zu fragen, weil Traces in dieser Arbeit als Sequenzen von XML-Fragmenten dargestellt werden und unter Verwendung des Elementes `tra:Trace` als XML-Dokumente notiert werden können. Eine solche Beschreibung ist zum größten Teil vorhanden. Ein fehlender Teil kann leicht angegeben werden. Das soll im Folgenden diskutiert werden.

Der fehlende Teil der Beschreibung in XML-Schema bezieht sich auf die Elemente `tra:Trace` und `tra:Message`, die die grundlegende Struktur des Traces bilden und im vorherigen Kapitel 7.2.2 vorgestellt wurden. Ihre Beschreibung in XML-Schema ist für alle Traces gleich und muss daher nur einmalig angegeben werden. Das soll in dieser Arbeit geschehen. Das Schemadokument ist in Abbildung 72 abgebildet.

Die Kindelemente des Elementes `tra:Trace` sind Message-Elemente. Diese enthalten als einziges Kindelement die beobachtete SOAP-Nachricht, also ein Element `env:Envelope`. Daher wird in Abbildung 72 in der Definition des Typs der Message-Elemente auf das Element `env:Envelope` verwiesen. Es gehört zu zum Namensraum `http://www.w3.org/2002/06/soap-envelope`, der im SOAP-Standard definiert wird. Dieser Namensraum enthält auch weitere Elemente und Attribute, die den Rahmen der SOAP-Nachricht bilden. Für ihn exsistiert ein XML-Schemadokument in [W3C 02b], das in das Schemadokument in Abbildung 72 importiert wird.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:env="http://www.w3.org/2002/06/soap-envelope"
    xmlns:tra="http://ti5.tu-harburg.de/venzke/20021015/traces"
    targetNamespace="http://ti5.tu-harburg.de/venzke/20021015/traces">

  <xsd:import namespace="http://www.w3.org/2002/06/soap-envelope"
      schemaLocation="http://www.w3.org/2002/06/soap-envelope/" />

  <xsd:simpleType name="toType">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="Client" />
      <xsd:enumeration value="Service" />
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:complexType name="MessageType">
    <xsd:sequence>
      <xsd:element ref="env:Envelope" />
    </xsd:sequence>
    <xsd:attribute name="to"
                   type="tra:toType" use="required" />
    <xsd:attribute name="operation"
                   type="xsd:string" use="required" />
  </xsd:complexType>

  <xsd:element name="Message" type="tra:MessageType" />

  <xsd:complexType name="TraceType">
    <xsd:sequence>
      <xsd:element ref="tra:Message"
                   minOccurs="0" maxOccurs="unbounded" />
    </xsd:sequence>
  </xsd:complexType>

  <xsd:element name="Trace" type="tra:TraceType" />
</xsd:schema>
```

Abbildung 72: XML-Schemadokument des Namensraumes für Traces

Der Namensraum `http://www.w3.org/2002/06/soap-envelope` lässt offen wie die anwendungsspezifischen Body-, Header- und Detaileinträge deklariert sind. Hierzu wird im XML-Schemadokument `xsd:any` (siehe Kapitel 4.2.6.2) verwendet. Mit dessen Hilfe wird bei der Definition der Typen für die Elemente `env:Body`, `env:Header` und `env:Fault/env:Detail` festgelegt, dass sie eine beliebige Zahl von beliebigen Kindelementen haben dürfen.

Obwohl dort Body-, Header- und Detaileinträge also nicht festgelegt sind, gibt es für sie doch Beschreibungen in XML-Schema, wenn diese gemäß den Konventionen für dem Präzisierungsstandard aus Kapitel 5.3 in WDSL beschrieben sind. Dann sind sie innerhalb der WSDL-Beschreibung in XML-Schema als Elemente deklariert. Somit sind auch diese anwendungsspezifischen Teile des Traces in XML-Schema beschrieben.

7.2.4 Sichten des Beobachters

Welche Ereignisse sich im Trace befinden, hängt davon ab, welche Entität ein Beobachter beobachtet. Im Falle von Webkomponenten sind das Webservices oder Webclients. Greifen mehrere Webclients auf den gleichen Webservice zu, wie in Abbildung 73 angedeutet, sieht ein Beobachter die SOAP-Nachrichten aller Webclients, wenn er den Webservice beobachtet. Beobachtet er dagegen einen Webclient, sieht er nur dessen SOAP-Nachrichten.

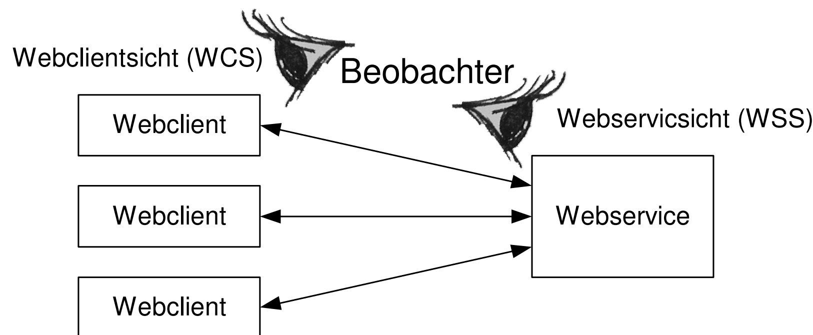

Abbildung 73: Beobachter mit verschiedenen Sichten

Welche Entität der Beobachter beobachtet, soll als seine Sicht bezeichnet werden. Von der Sicht hängt ab, welche Ereignisse der Beobachter wahrnimmt und sich damit im Trace befinden. Eine Sicht, in der der Beobachter alle Ereignisse des Webservices wahrnimmt, soll als Webservicesicht (WSS) bezeichnet werden. Analog wird eine Sicht auf die Ereignisse eines Webclients als Webclientsicht (WCS) bezeichnet.

Ziel in dieser Arbeit ist es, die Schnittstellen von Webservices zu beschreiben. Dazu können sowohl die WSS als auch die WCS sinnvoll sein. Da in der WSS alle SOAP-Nachrichten im Trace enthalten sind, die mit dem Webservice ausgetauscht wurden, kann in dieser Sicht das Verhalten des Webservices als ganzes verstanden werden. Z. B. können aus einem Trace Hinweise auf den Zustand des Webservices abgeleitet werden. Das ist in der WCS nicht möglich, weil durch die Beobachtung der SOAP-Nachrichten nur eines Webclients Informationen über Zustandsänderungen durch andere Webclients fehlen.

Die WCS ist trotzdem hilfreich, wenn es um die Beschreibung geht, wie ein Webclient den Webservice benutzen darf. Dazu sei für diese Arbeit stets angenommen, dass ein Webclient in der WCS nur mit einem Webservice kommuniziert. Dann handelt es sich bei den beobachteten SOAP-Nachrichten nur um solche, die zwischen einem Webclient und einem Webservice ausgetauscht werden. Die Konzentration auf nur diese SOAP-Nachrichten vereinfacht das Verständnis, wie ein einzelner Webclient den Webservice verwendet.

Im Kapitel 5.2.6 wurde für diese Arbeit festgelegt, dass SOAP-Intermediaries nicht betrachtet werden sollen. Wäre diese Festlegung nicht getroffen worden, könnte neben der WSS und der WCS noch eine Intermediarysicht eingeführt werden, bei der der Beobachter ein SOAP-Intermediary beobachtet. Mit dieser Sicht könnte der Beobachter vom SOAP-Intermediary empfangene oder gesendete SOAP-Nachrichten beobachten. Es wäre genauer festzulegen, ob es die empfangenen oder die gesendeten sind oder beides. In dieser Sicht könnte dann beobachtet werden, wie mehrerer Webclients mit mehreren Webservices kommunizieren. Das soll in dieser Arbeit jedoch nicht betrachtet werden.

7.3 Spezifikationen von Webservices

Nach der Übertragung von Hoares Begriffsbildung auf Webservices kann auch sein Spezifikationsverfahren auf Webservices übertragen werden. Webclients und Webservices lassen sich mit Hilfe der Traces als Entitäten beschreiben. Zur Spezifikation werden prädikatenlogische Ausdrücke verwendet, die eine freie Variable für den Trace enthalten, wobei der Trace eine Struktur hat, wie in Kapitel 7.2.2 beschrieben.

7.3.1 Logische Aussagen und WSDL

Mit prädikatenlogischen Ausdrücken ließen sich Schnittstellen von Webservices unabhängig von WSDL beschreiben. Im Spezifikationsverfahren SXQT soll jedoch die WSDL-Beschreibung weiterhin verwendet werden. Jede Anforderung, die über das in WDSL beschriebene hinausgehen, wird einzeln als prädikatenlogischer Ausdruck formuliert. Jeder Ausdruck führt zu einer genaueren Beschreibung der Schnittstelle, weil er das erlaubte Verhalten des Webservices weiter einschränkt. Aus Sicht des Spezifikationsverfahrens von Hoare wird in WSDL das Alphabet des Webservices beschrieben.

Eine SXQT-Spezifikation eines Webservices besteht damit aus seiner Beschreibung in WSDL sowie einer Menge von prädikatenlogischen Ausdrücken. Die Beschreibung in WSDL muss neben dem WSDL-Standard auch der in Kapitel 5.3 vorgestellten Präzisierung folgen.

Die Menge von prädikatenlogischen Ausdrücken ist einem einzelnen äquivalent, da der einzelne durch die Konjunktion der Ausdrücke der Menge gebildet werden kann und die Menge nur einen einzigen Ausdruck enthalten kann. In den Kapiteln 6.1 und 6.2 wurden beispielhaft eine Reihe von Anforderungen natürlichsprachlich beschrieben, die in WSDL nicht ausgedrückt werden können. Jede kann einzeln als logischer Ausdruck formuliert und der Beschreibung in WSDL zugefügt werden.

Weiterhin WSDL zu verwenden, erspart eine doppelte, redundante Beschreibung von Webservices. Webservices werden heute typischerweise ohnehin in WSDL beschrieben. WSDL wird durch das W3C standardisiert. Außerdem gibt es eine Vielzahl von Werkzeugen, die WSDL-Beschreibungen erzeugen oder interpretieren. Auch Entwickler haben heute z. T. Erfahrung im Umgang mit WSDL. Daher sollten Webservices in jedem Fall auch weiterhin in WSDL beschrieben sein. Würden Webservices zusätzlich vollständig mit einem neuen Spezifikationsverfahren unabhängig von WSDL spezifiziert, würden sie teilweise doppelt beschrieben, einerseits in WSDL und andererseits mit dem neuen Spezifikationsverfahren. Das würde zu einem höheren Spezifikationsaufwand führen. Außerdem bestünde dann die Gefahr, dass sich beide Spezifikationen widersprechen.

Vorteilhaft erscheint die Kombination von logischen Ausdrücken mit WSDL auch, weil in beidem unterschiedliche Aspekte des Webservices besonders gut beschrieben werden kann. So kann die Struktur der in der SOAP-Nachricht enthaltenen XML-Fragmente besonders klar und kompakt mit Hilfe von XML-Schema als Grammatik beschrieben werden. Eine Beschreibung wäre auch mit logischen Ausdrücken möglich. Die Ausdrücke wären jedoch sehr komplex, was die Spezifikation schlecht lesbar machen würde. Ähnlich verhält es sich auch mit Typen von Blattelementen und Attributen. Auch für ihre Beschreibung würde entweder eine große Zahl von Ausdrücken oder sehr komplexe benötigt.

Bestimmte Aspekte von Webservices lassen sich sogar überhaupt nicht mit logischen Aussagen formulieren. Wie sollte z. B. die Netzwerkadresse eines Webservices beschrieben werden? Wie sollten Operationen zu Schnittstellen zusammengefasst werden? In WSDL kann beides direkt deklariert werden.

Einzelne Anforderungen an Schnittstellen von Webservices, wie sie in den Kapiteln 6.1 und 6.2 vorgestellt wurden, lassen sich jedoch gut als logische Aussagen darstellen. Auf diesem Teil der Spezifikationen soll in dieser Arbeit der Schwerpunkt liegen.

Tatsächlich gibt es auch im Spezifikationsverfahren von Hoare die Trennung in eine grundsätzliche Beschreibung einer Entität und einer genaueren Beschreibung mit logischen Ausdrücken. Welche Ereignisse einer Entität als relevant erachtet werden, wird

durch ihr Alphabet festgelegt. Wann welches Ereignis stattfinden kann, wird danach mit einem logischen Ausdruck beschrieben. Nur diesen bezeichnet Hoare als Spezifikation. Er ist aber implizit vom Alphabet der Entität abhängig.

Ähnlich verhält es sich auch bei der Spezifikation von Webservices. WSDL beschreibt die Menge von SOAP-Nachrichten, die mit der Webkomponente ausgetauscht werden können, also ihr Alphabet. Allerdings handelt es sich anders als bei Hoare nicht einfach um eine Menge von Ereignisnamen. Wegen der unterscheidbaren, viel komplexeren Ereignisse (SOAP-Nachrichten) erfordert auch ihre Beschreibung mehr Informationen, wie im Kapitel 4 über WSDL beschrieben. Diese machen einen relevanten Teil der Beschreibung des Webservices aus.

Aus diesem Grund erscheint es nicht angemessen, dass WSDL nur ein impliziter Teil der Spezifikation ist. Daher wird er neben den logischen Ausdrücken als wesentlicher Teil einer SXQT-Spezifikation betrachtet.

Interessant ist, dass es durch die Verwendung von WSDL nicht mehr möglich ist, direkt Webclients zu spezifizieren. Eine WSDL-Beschreibung exsistiert nur für Webservices. Nur diese kann um logische Ausdrücke ergänzt werden. Da sowohl Webclients als auch Webservices im Sinne von Hoare Entitäten sind, mag das problematisch erscheinen. Tatsächlich sind die beschriebenen Schnittstellen von Webservices aber natürlich Schnittstellen zwischen Webservices und Webclients und beschreiben die zwischen ihnen erwartete Kommunikation. Indirekt wird also auch das Verhalten der Webclients mitbeschrieben.

7.3.2 Übertragen von Gesetzen für Prozessterme

Ein wesentlicher Unterschied zwischen dem Modell von Hoare und der Beschreibung von Webservices ist, dass die Entitäten nicht als Prozessterme beschrieben sind. Viele der Erkenntnisse in [Hoare 85] basieren jedoch auf Prozesstermen, die den Hauptfokus des Buches ausmachen. Trotzdem lassen sich einige seiner Gesetze auch unabhängig von Prozesstermen interpretieren und sich so Anforderungen an Spezifikationen ableiten.

Hierzu bekommt der Webservice bzw. Webclient selbst die Rolle des Prozessterms. Dann kann z. B. die Menge `traces(P)` als die Menge von Traces verstanden werden, die bei Beobachtung von `P` beobachtet werden kann. Diese Menge lässt sich definieren, auch wenn sie nicht mehr aus dem Prozessterm abgeleitet werden kann.

Mit diesem Verständnis von `traces(P)` können einige Gesetze von Hoare interpretiert werden. Die Tatsache, dass ein Trace nur Ereignisnamen aus dem Alphabet des Prozessterms enthalten darf, schreibt Hoare z. B. als `(traces(P) ⊆ (αP)*)`[40]. In diesem Ausdruck ist `αP` das Alphabet des Prozessterms `P` sowie `(αP)*` die Menge aller endlichen Traces, die aus `αP` gebildet werden können. Angewendet auf einen Webservice `P` bedeutet das, dass alle beobachteten Traces nur SOAP-Nachrichten aus seinem Alphabet enthalten dürfen, also solche die dem WSDL-Dokument entsprechen.

Ähnlich lässt sich Hoares Gesetz `(<> ∈ traces(P))`[41] für Webservices interpretierten. Es sagt aus, dass sich der leere Trace bei jeder Entität beobachten lässt. Das gilt auch für Webservices. Der leere Trace wird beobachtet, bevor überhaupt ein Ereignis stattgefunden hat. Die Aussage ist wichtig, weil eine SXQT-Spezifikation, die ihr wider-

---

[40] [Hoare 85], Seite 51, Gesetz L8.

[41] [Hoare 85], Seite 51, Gesetz L6.

spricht, nie erfüllt werden kann. Daher muss jeder Ausdruck `S(tr)` so konstruiert sein, dass er für den leeren Trace zutrifft, dass also gilt `(S(<>) = true)`.

Interessant ist auch Hoares Gesetz $(s\hat{}t \in traces(P) \Rightarrow s \in traces(P))$[42]. Darin ist `s^t` die Konkatenation der beiden Traces `s` und `t`. Das Gesetz sagt aus, dass stets der Trace `s` beobachtet werden kann, wenn es möglich ist, bei der gleichen Entität den Trace `s^t` zu beobachten. Das Gesetz muss gelten, weil es vor der Beobachtung von `s^t`, einen Zeitpunkt gegeben haben muss, an dem die Ereignisse in `t` noch nicht stattgefunden hatten. Bis zu diesem Zeitpunkt ist der Trace `s` beobachtet worden. Jeder logische Ausdruck muss so konstruiert sein, dass er auch diesem Gesetz nicht widerspricht.

7.3.3 Erfüllen einer Spezifikation

Auch der Begriff des Erfüllens einer Spezifikation kann auf Webservices sowie Webclients angewendet werden. Diesen Begriff definiert Hoare allgemein, indem er von Produkten redet. Ein Produkt `P` erfüllt die Spezifikation `S` (kurz: `P sat S`), falls für jede Beobachtung von `P` gilt, dass für sie die Spezifikation `S` zutrifft. Ist das Produkt durch einen Prozessterm beschrieben, kann das nachgewiesen werden. Bei Webkomponenten ist das jedoch nicht möglich.

Trotzdem kann für einen beobachteten Trace beurteilt werden, ob er der Spezifikation entspricht. Ist das für viele beobachtete Traces der Fall, beweist das nicht, dass die beteiligten Webkomponenten die Spezifikation erfüllen. Es kann lediglich das Vertrauen in diese Tatsache vergrößern. Umgekehrt reicht jedoch die Beobachtung eines einzelnen Traces, für den die Spezifikation nicht zutrifft, um nachzuweisen, dass die Spezifikation von einer der beteiligten Webkomponenten nicht erfüllt wird.

Da die Spezifikation gleichzeitig Webclient und Webservice beschreibt, stellt sich die Frage, welche dieser Entitäten gegen die Spezifikation verstoßen hat. Da das Modell von Hoare die Betrachtung von Kausalitäten ausschließt, verbietet sich dort diese Frage. Für diese Arbeit wurde dagegen in Kapitel 7.2.1 festgelegt, dass Kausalitäten betrachtet werden sollen. Der SOAP-Sender soll für von ihm gesendete SOAP-Nachrichten als verantwortlich angesehen.

In einem Trace, der nicht der Spezifikation entspricht, sind jedoch in der Regel viele SOAP-Nachrichten enthalten, die von einem Webservice und mehreren Webclients gesendet worden sein können. Welche der beobachteten SOAP-Nachrichten im Trace soll als die betrachtet werden, die gegen die Spezifikation verstößt?

Für diese Arbeit wird das im Folgenden festgelegt. Der Austausch der ersten SOAP-Nachricht im Trace, bei der die Spezifikation nicht mehr zutrifft, sei das Ereignis, das im Trace gegen die Spezifikation verstößt. Genauer soll wie folgt definiert werden. Sei `t` der Trace, für den bestimmt werden soll, welches Ereignis gegen den logischen Ausdruck `S(tr)` verstößt. Sei $n = \#t$ die Länge des Traces. Außerdem enthalten die Traces u_i gerade die `i` ersten Ereignisse aus `t` (mit $0 \le i \le n$). Sei `k` mit $1 \le k \le n$ die kleinste natürliche Zahl für die gilt $S(u_k) = false$. Dann ist das letzte Ereignis in u_k das Ereignis in `t` das gegen die Spezifikation verstößt.

Es sei klargestellt, dass es nicht in jedem Trace ein solches Ereignis geben muss. Vielmehr ist es gerade Ausdruck eines erwarteten Verhaltens, dass in einem Trace kein Ereignis gegen die Spezifikation verstößt. Andererseits gibt es aber in einem Trace `t` stets ein solches Ereignis, falls `S(t) = false`, da zumindest $S(u_n) = S(t) = false$ und `S(<>) = true` (siehe Kapitel 7.3.2).

[42] [Hoare 85], Seite 51, Gesetz L7.

Mit dieser Definition kann der Verstoß dem Webclient bzw. dem Webservice zugeordnet werden. Da der SOAP-Sender einer Nachricht für sie verantwortlich ist, hat der Webservice bzw. Webclient den Verstoß ausgelöst, der die SOAP-Nachricht gesendet hat, die gegen die Spezifikation verstößt. Dieser Webservice bzw. Webclient hat gegen die Spezifikation verstoßen.

7.3.4 Spezifikationen und Sichten

Bei der Erstellung einer SXQT-Spezifikation muss die in Kapitel 7.2.4 vorgestellte Sicht berücksichtigt werden. Durch die Sicht entscheidet sich, welche Entität beobachtet wird und damit, welche Ereignisse sich im Trace befinden. Logische Ausdrücke von Spezifikationen können nur SOAP-Nachrichten miteinander in Beziehung setzen, die sich im Trace befinden. Die verwendete Sicht muss erlauben sie zu beobachten.

Sollen z. B. SOAP-Nachrichten miteinander in Beziehung gesetzt werden, über die mehrere Webclients mit einem Webservice kommunizieren, muss die Webservicesicht (WSS) gewählt werden. Bei der Webclientsicht (WCS) würden nur SOAP-Nachrichten eines Webclients beobachtet. Sie könnten nicht mit SOAP-Nachrichten anderer Webclients in Beziehung gesetzt werden.

Umgekehrt müssen logische Ausdrücke auch für die WSS geeignet sein. Ist eine Spezifikation für die WCS erstellt, können weitere SOAP-Nachrichten von anderen Webclients im Trace dazu führen, dass der logische Ausdruck auch beim erwarteten Verhalten für einen beobachteten Trace `false` liefert oder bei nicht erwarteten Verhalten `true`.

Obwohl es auch Anforderungen gibt, die von der Sicht unabhängig als logischer Ausdruck formuliert werden können, ist das also allgemein nicht der Fall. Daher muss für einen logischen Ausdruck stets bekannt sein, in welcher Sicht er verwendet werden kann oder ob die Sicht für ihn keine Rolle spielt. Wenn möglich, sollten alle logischen Ausdrücke einer Spezifikation so formulieren werden, dass sie in der gleichen Sicht verwendbar sind. Dann ist nur ein Beobachter in einer Sicht erforderlich, um den Traces aufzuzeichnen.

7.3.5 Spezifikationen und Startzustand

Nicht nur die Sicht bestimmt, welche SOAP-Nachrichten sich im Trace befinden, sondern auch der Zeitpunkt, an dem die Beobachtung begonnen wurde. Von ihr abhängig können sich Webservice und Webclient in einem unterschiedlichen Startzustand befinden, so dass unterschiedliche Traces beobachtet werden können. Das muss für die Konstruktion logischer Ausdrücke von SXQT-Spezifikationen berücksichtigt werden.

Dieses Problem betrachtet Hoare in [Hoare 85] nicht. Es ist für ihn nicht relevant, weil bei ihm beobachtete Entitäten durch Prozessterme beschrieben werden, die auch den Zustand der Entität mit enthalten, wie in Kapitel 7.1.2 beschrieben. Es wird vorausgesetzt, dass die Beobachtung stets in diesem Startzustand begonnen wird. In Spezifikationen kann er daher vorausgesetzt werden.

Vorauszusetzen, dass die Beobachtung eines Webservices oder Webclients stets im gleichen, bekannten Startzustand begonnen wird, ist jedoch nicht hilfreich. Es gibt Fälle, in denen der Zustand eines Webservices nur direkt nach seiner Installation bekannt ist. Verwendet er z. B. eine Datenbank, um Daten abzulegen, bestimmt ihr Inhalt den Zustand des Webservices und damit auch sein Verhalten. Der Zustand ändert sich durch Operationsaufrufe. Einen bekannten Startzustand bei Beginn der Beobachtung vorauszusetzen, würde dann bedeuteten, dass die Beobachtung stets direkt nach der Installation des Webservices begonnen werden muss.

Auch wenn eine solche Art von langfristigem Zustand für die Spezifikation nicht von Interesse ist, erschwert das Voraussetzen eines bekannten Startzustandes zumindest den Zeitpunkt zu finden, in dem die Beobachtung begonnen werden muss. Zur Spezifikation der in Kapitel 6.1.3 beschriebenen Sperren, die die Webkomponente für eine Internetzeitung verwendet, könnte z. B. ein Startzustand gewünscht sein, in dem keine Sperren gesetzt sind. Wie soll der Beobachter ermitteln, wann das der Fall ist? Wie kann dem Beobachter formal der gewünschte Startzustand mitgeteilt werden?

Aus diesen Gründen müssen logische Ausdrücke von SXQT-Spezifikationen stets so formuliert werden, dass sie vom Startzustand unabhängig sind. Das erlaubt, die Beobachtung an jedem Zeitpunkt zu beginnen. Es erschwert aber, die logischen Ausdrücke zu formulieren. Nachfolgend soll diskutiert werden, wie dazu vorgegangen werden kann.

Zunächst sei festgestellt, dass viele logische Ausdrücke ohnehin vom Startzustand unabhängig sind. Das gilt vor allem für alle Ausdrücke, mit denen Anforderungen formuliert werden, die sich nur auf einzelne SOAP-Nachrichten beziehen. Beispiele wurden in den Kapiteln 6.1.1, 6.2.1 und 6.2.2 gegeben. Da sich solche Anforderungen nur auf eine einzelne SOAP-Nachrichten beziehen, kann jede beobachtete SOAP-Nachricht davon unabhängig geprüft werden, welche anderen SOAP-Nachrichten vorher beobachtet wurden.

Ergibt sich die Unabhängigkeit vom Startzustand für eine Anforderung nicht derart natürlich, müssen logische Ausdrücke so formuliert werden, dass sie SOAP-Nachrichten ignorieren, die sie nicht mit anderen in Beziehung setzten können, weil diese vor dem Beginn der Beobachtung ausgetauscht wurden. Logische Ausdrücke müssen den Trace in gewisser Weise filtern, so dass alle verbleibenden SOAP-Nachrichten geprüft werden können.

Ausdrücke für Anforderungen an einzelne Request-/Response-Paare können den Trace z. B. so filtern, dass alle Request-/Response-Paare ignoriert werden, für die der Request im Trace fehlt. So werden solche Ausdrücke vom dem Aspekt des Startzustandes unabhängig, dass schon vor Beginn der Beobachtung Requests ausgetauscht wurden, für die danach der Response beobachtet wird.

Kann an einer SOAP-Nachricht erkannt werden, dass ein geeigneter Startzustand erreicht ist, kann ein logischer Ausdruck so konstruiert werden, dass er alle vorherigen SOAP-Nachrichten ignorieren. Für die nachfolgenden SOAP-Nachrichten kann der Startzustand dann vorausgesetzt werden.

Gibt es für eine Webkomponente keine solchen SOAP-Nachrichten, kann sie für die Spezifikation modifiziert werden. Ihr kann eine Operation zugefügt werden, die sie in einen definierten Zustand bringt. Beim Aufruf der Operation werden SOAP-Nachrichten ausgetauscht und in den Trace aufgezeichnet. Damit kann für nachfolgende SOAP-Nachrichten ein definierter Startzustand angenommen werden. Vorherige SOAP-Nachrichten können von logischen Ausdrücken ignoriert werden.

Diese Vorgehensweise könnte z. B. für logische Ausdrücke verwendet werden, mit denen die in Kapitel 6.1.3 vorgestellten Sperren der Webkomponenten für eine Internetzeitung beschrieben werden. Der Webkomponente könnte eine Operation zugefügt werden, mit der alle Sperren freigegeben werden und sie so in einen bekannten Zustand bringt. Die logischen Ausdrücke können dann alle SOAP-Nachrichten ignorieren, die vor dem ersten Aufruf dieser Operation beobachtet wurden.

7.3.6 Notationen für logische Ausdrücke

Für logische Ausdrücke von SXQT-Spezifikationen muss eine geeignete Notation gewählt werden. Damit SXQT-Spezifikationen maschinell verarbeitet werden können, sollten sie maschinenverständlich abgelegt werden. Das ist z. B. für die in Kapitel 2.7 angedeutet automatische Validation erforderlich. Außerdem muss es möglich sein, Aussagen über XML-Fragmente zu formulieren, weil SOAP-Nachrichten im Trace XML-Fragmente sind.

Beides ist mit der von Hoare verwendeten Prädikatenlogik schwierig. Er verwendet die mathematische Notation für prädikatenlogische Ausdrücke, die Zeichen wie $\exists$, $\forall$ und $\subseteq$ enthält. Mit ihnen formuliert er logische Ausdrücke über Traces, die einfache Ereignisnamen enthalten, wie in Kapitel 7.1.4 vorgestellt. Ereignisnamen reichen in seinem Formalismus, weil Ereignisse einer Ereignisklasse ununterscheidbar sind und daher im Trace nur die Ereignisklasse durch den Ereignisnamen bezeichnet werden muss.

Lediglich zur Beschreibung von Ereignissen, die über Kanäle ausgetauscht werden, definiert Hoare im Kapitel 4 von [Hoare 85] eine Notation, die es erlaubt „etwas" komplexere Ereignisse zu beschreiben. Als Ereignisse betrachtet er dort Kommunikationen, also den Austausch einer Nachricht über einen Kanal, wobei unterschiedliche Nachrichten unterschieden werden können. Ereignisnamen solcher Ereignisse notiert Hoare in der Form c.v, wobei c der Name des Kanals und v die ausgetauschte Nachricht ist. Hoare definiert zwei Operationen, um aus einer Nachricht diese beiden Teile zu extrahieren.

Nachrichten sind bei Hoare stets einfache Werte. Es stellt kein allgemeines Verfahren zur Verfügung, um aus einfachen Werten komplexe zu konstruieren. Hoare definiert in [Hoare 85] auch keine Operationen, mit denen komplexe Werte in logischen Ausdrücken berücksichtigt werden können.

Für Spezifikationen von Webservices ist beides erforderlich. Ereignisse sind im Trace als XML-Fragmente enthalten. Daher werden Operationen benötigt, die es in logischen Ausdrücken erlauben mit XML-Fragmenten umzugehen. Einzelne Werte müssen aus dem XML-Fragment isoliert werden können, um sie miteinander in Beziehung setzen zu können. Auch muss es möglich sein, Aussagen über die Struktur formulieren zu können. Z. B. muss die Anzahl von Elementen im XML-Fragment mit bestimmten Eigenschaften bestimmt werden können. Das ist mit der Notation von Hoare nicht möglich.

Für die Beschreibung von Webservices ist es zweckmäßig, logische Ausdrücke in einer Form zu notieren, die sich gut eingeben und in Dateien speichern lässt. Wie bei WSDL sollte es auch möglich sein, diese innerhalb von XML-Dokumenten zu speichern. Das wird durch einige Zeichen erschwert, die in der Mathematik und von Hoare verwendet werden. Zeichen wie $\exists$, $\forall$ und $\subseteq$ sind in vielen Zeichensätzen und vor allem auf Tastaturen nicht vorhanden. In XML-Dokumenten muss für sie ein numerischer Zeichencode angegeben werden. Besser ist es daher, sie durch Zeichenketten aus einfacheren Zeichen zu ersetzen, z. B. das Zeichen $\forall$ durch die Zeichenkette „`every`".

Diese Erweiterungen und Änderungen an der von Hoare verwendeten Prädikatenlogik führen letztlich zu einer neuen Sprache, mit der die Aussagen formuliert werden. Sie soll weiterhin die Möglichkeit bieten, Aussagen als prädikatenlogische Ausdrücke zu notieren. Sie muss aber auch effizient maschinell les- und verwendbar sein. Für die Sprache sollte auch die Anforderung aus den „Web Services Architecture Requirements" in [Austin 02] berücksichtigt werden, dass für die Beschreibung von Nachrichten und Protokollen von Webservices XML-basierte Techniken verwenden werden sollten.

Vorteilhaft wäre, wenn es eine geeignete Sprache bereits gäbe, die zu den XML-basierten Techniken zählt. Wäre die Sprache Entwicklern, die Spezifikationen erstellen oder lesen sollen, bereits bekannt, müssten sie sie zu diesem Zweck nicht erst erlernen. Der Einarbeitungsaufwand würde reduziert. Würde die Sprache außerdem auch für andere Anwendungen verwendet, hat das den Vorteil, dass Entwickler über eine größere Erfahrung mit ihrem Umgang verfügen. Für die Verbreitung der Sprache wäre es außerdem vorteilhaft, wenn die Sprache standardisiert wäre, z. B. durch das W3C.

Da die logischen Ausdrücke der Sprache nicht nur von Menschen gelesen werden sollen, muss für sie eine Softwareunterstützung existieren. Zumindest sollte es diese erlauben, dass für einen beobachteten Trace geprüft werden kann, ob er einem logischen Ausdruck der Spezifikation entspricht. Hierzu muss die Software den logischen Ausdruck auswerten, also prüfen, ob er für den beobachteten Trace `true` oder `false` liefert. Vorteilhaft wäre es, wenn für die Sprache eine solche Softwareunterstützung nicht erst erstellt werden müsste, sondern sie bereits vorhanden wäre. Das ist nur möglich, wenn die Sprache nicht neu entwickelt wird. Bei Sprachen, die bereits standardisiert sind, ist die Verfügbarkeit der Softwareunterstützung zu erwarten.

7.3.7 Sprachen des W3C für Ausdrücke

Zwei Sprachen, die von W3C standardisiert sind und für die die Anforderungen aus dem vorherigen Kapitel 7.3.6 zutreffen, sind die XML Path Language (XPath) und die XML Query Language (XQuery). XPath wurde bereits in Kapitel 2.4.2 angesprochen. Anders als dort, soll hier jedoch die Version 2.0 betrachtet werden. Ihr aktueller Arbeitsentwurf ist [Berglund 02]. XQuery wird im Detail in Kapitel 7.4 vorgestellt. [Boag 02] ist der aktuelle Arbeitsentwurf.

Beide Sprachen werden beim W3C gemeinsam entwickelt. Sie teilen grundsätzlich die gleiche Syntax und Semantik. [Berglund 02] stellt fest, dass sogar der Text beider Standards aus einer Quelle abgeleitet wird. Die veröffentlichten Arbeitsentwürfe haben daher weitgehend die gleichen Kapitel und sind z. T. wörtlich identisch.

Beide Sprachen wurden nicht mit dem Ziel entwickelt, prädikatenlogische Ausdrücke zu formulieren. Ausdrücke können auch andere als logische Werte liefern. Ziel von XPath ist es, Teile eines XML-Dokumentes adressieren zu können [Berglund 02]. Das schließt das Extrahieren von Werten oder auch von Mengen von Elementen oder Attributen ein. Ein ähnliches Ziel hat auch XQuery. Hier ist die Sicht jedoch auf die Abfrage und Verarbeitung von Informationen, die aus unterschiedlichen Datenquellen stammen [Boag 02].

Trotzdem können mit beiden Sprachen auch prädikatenlogische Ausdrücke formuliert werden. Da beide für die Verarbeitung von XML-Dokumenten entwickelt wurden, haben sie leistungsfähige Operationen, um mit solchen umzugehen. Es können also Werte extrahiert und mit Prädikaten in Beziehung gesetzt werden. Es gibt auch Prädikate, mit denen Aussagen über die Struktur eines XML-Fragmentes formuliert werden können.

Innerhalb der Ausdrücke werden keine Sonderzeichnen verwendet, wie sie in der Mathematik üblich sind. Statt dessen sind diese wie in Kapitel 7.3.6 gefordert durch Zeichenketten ersetzt. Auch die vollständigen Ausdrücke sind bei XPath und XQuery Zeichenketten, die sich gut in XML-Dokumente einbetten lassen. XSLT [Clark 99b] ist ein Beispiel für eine Sprache, in der das geschieht. Dort sind XPath-Ausdrücke in Attributen enthalten, wobei jedoch XPath 1.0 verwendet wird. Neben der Darstellung als Zeichenkette enthält XQuery noch eine weitere, bei der ein XQuery-Ausdruck selbst als XML-Dokument formuliert wird.

Für beide Sprachen kann in Anspruch genommen werden, dass sie weit verbreitet sind, bzw. weit verbreitet sein werden. XPath 1.0 ist bereits jetzt weit verbreitet. Das eben genannte XSLT ist nur ein Beispiel. Zu erwarten ist, dass nach Abschluss der Standardisierung von XPath 2.0, auch diese Version eine ähnliche Verbreitung erlangen wird. Da XQuery noch in ihrer ersten Version standardisiert wird, hat es noch nicht den Bekanntheitsgrad von XPath. Doch auch hier ist zu erwarten, dass nach Abschluss der Standardisierung viele Entwickler im XML-Umfeld diese Sprache kennen werden. Neben der Standardisierung durch das W3C spricht dafür auch die große Ähnlichkeit zu XPath 2.0, die Entwicklern mit Kenntnissen von XPath 2.0 das Erlernen von XQuery stark vereinfacht.

Implementierungen zur Auswertung von Ausdrücken sind für XPath 1.0 in großer Zahl vorhanden. MSXML (siehe [Microsoft 02d], [Kay 01]) sowie die Open Source Produkte Saxon (siehe [Kay 01], [Kay 02]) und Xalan (siehe ([Apache 01], [Kay 01]) sind Beispiele. Nach Abschluss der Standardisierung werden solche auch für XPath 2.0 vorhanden sein. Saxon enthält ab seiner Version 7.0 eine experimentelle Implementierung. Auch für XQuery gibt es bereits jetzt erste Implementierungen. Beispiele sind der IPSI-XQ XQuery Demonstrator [Fraunhofer 02] der Frauenhofer Gesellschaft und Quip [SoftwareAG 02b] von der Software AG.

Wie die genannten Argumente zeigen, ist sowohl XPath als auch XQuery eine zweckmäßige Wahl für eine Sprache, um prädikatenlogische Ausdrücke zu notieren. Es bleibt die Frage, welche der beiden zur Spezifikation von Webservices verwendet werden sollte. Für SXQT wurde in dieser Arbeit XQuery gewählt. XQuery hat gegenüber XPath zusätzliche Features. Insbesondere die Tatsache, dass in XQuery Funktionen definiert werden können, was in XPath nicht möglich ist, führte zur Entscheidung für XQuery. Funktionen sind ein wichtiges Feature, um Spezifikationen lesbar zu erstellen. In Kapitel 7.6 werden sie verwendet, um unter anderem Hoares Traceoperationen zu implementieren. Im folgenden Kapitel 7.4 wird zunächst XQuery vorgestellt, bevor in Kapitel 7.5 diskutiert wird, wie sich mit dieser Sprache logische Ausdrücke von Spezifikation formulieren lassen.

7.4 XQuery

XQuery ist eine funktionale Sprache zur Verarbeitung XML-strukturierter Daten. Die Syntax erlaubt unterschiedliche Typen von Ausdrücken, die rekursiv andere Ausdrücke als Operanden enthalten. XQuery verwendet ein auf XML-Schema basierende Typsystem. Werte können in Variablen zwischengespeichert werden. Außerdem können benutzerdefinierte Funktionen definiert werden.

Der XQuery-Standard [Boag 02] wird um weitere Dokumente ergänzt, die spezielle Aspekte von XQuery festlegen. So wird in [Fernandez 02] das Datenmodell beschrieben, dass hinter der Sprache XQuery aber auch XPath 2.0 steht. Von beiden Sprachen unterstützte Funktionen und die Semantik unterstützter Operationen wird in [Malhotra 02] beschrieben. [Draper 02] legt die formale Semantik der Sprache XQuery fest. Zusätzlich zur nicht XML-basierten Syntax von XQuery wird in [Malhotra 01] die XML-basierte Syntax XQueryX definiert. Alle diese Dokumente sind noch Arbeitsentwürfe und müssen daher als „Work in Progress“ angesehen werden.

7.4.1 Typsystem

Die Sprache XQuery ist typisiert, wobei das Typsystem auf XML-Schema beruht. Typisiert sind Werte, Ausdrücke und Variablen. Gewonnen werden Typinformationen durch die Validation gegen Schemadokumente. Dem Typsystem von XML-Schema werden

Sequenzen zugefügt, also geordnete Sammlungen von Daten. Sie spielen in XQuery ein zentrale Rolle, da letztlich jeder Wert auch eine Sequenz ist. Daher wird auch eine Syntax eingeführt, mit der Typen von Sequenzen beschrieben werden können.

Geprüft werden Typen in der Analysephase (analysis phase) und dynamisch in der Ausführungsphase (execution phase). In der Analysephase wird ein XQuery-Ausdruck unabhängig von den zu verarbeitenden Eingabedaten geprüft. Dabei werden Variablen und Teilausdrücken statisch Typen zugeordnet. Es wird geprüft, ob als Operanden verwendete Teilausdrücke die erwarteten statischen Typen haben. In der Ausführungsphase wird ein XQuery-Ausdruck abhängig von seinen Eingabedaten ausgewertet. Hier haben alle Werte dynamisch Typen. Wieder wird geprüft, ob als Operanden verwendete Werte den erwarteten Typen entsprechen.

In der Ausführungsphase werden Typen durch Validation gegen XML-Schemadokumente gewonnen. Alle Eingabedaten werden validiert. Elemente und Attribute erhalten die in den Schemadokumenten definierten Typen. Sind Typinformationen unbekannt, wird für Elemente der Typ `xsd:anyType` und für Attribute der Typ `xsd:anySimpleType` angenommen.

In XQuery verarbeitete XML-Fragmente müssen nicht unbedingt aus den Eingabedaten stammen. Sie können auch mit Hilfe sogenannter Konstruktoren in XQuery-Ausdrücken erzeugt werden, was hauptsächlich für Elemente und Attribute geschieht. Erzeugten Elementen können Kindelemente und Attribute zugefügt werden. Sie sind dann aber nicht typisiert, statt dessen werden wieder die Typen `xsd:anyType` für Elemente und `xsd:anySimpleType` für Attribute angenommen. Um speziellere Typen zu gewinnen, muss für solche XML-Fragmente die Validation gegen Schemadokumente explizit aufgerufen werden.

Unterschieden werden in XQuery atomare Werte von Knoten. Atomare Werte haben einfache Typen von XML-Schema, die nicht mit `xsd:list` oder `xsd:union` erzeugt wurden. (Siehe Kapitel 4.2.5.) Ihnen gegenüber stehen Knoten, von denen es verschiedene Arten gibt. Elementknoten repräsentieren z. B. XML-Elemente, Attributknoten Attribute. Textknoten repräsentieren Zeichenketten und Dokumentknoten XML-Dokumente. Anders als atomare Typen haben Knoten eine Identität, können also unabhängig von ihrem Inhalt identifiziert werden.

XML-Dokumente werden als Hierarchie von Knoten repräsentiert. Ein Dokumentknoten hat einen Elementknoten als Kindknoten. Elementknoten haben ihrerseits Kindknoten. Ihre Kindelemente werden wieder als Elementknoten repräsentiert, enthaltener Text als Textknoten. Außerdem können Elementknoten ihre Attribute als Attributknoten zugeordnet werden.

Eine wichtige Rolle im Datenmodell von XQuery spielen Sequenzen. Eine Sequenz ist in XQuery eine geordnete Sammlung von Elementen, die atomare Werte oder Knoten sein können. Sequenzen können als Elemente aber keine Sequenzen enthalten. Eine Sequenz kann leer sein (leere Sequenz) oder dasselbe Element mehrfach enthalten. Außerdem ist eine Sequenz die genau ein Element enthält von diesem Element ununterscheidbar. In dem Sinne ist jeder atomare Wert oder Knoten auch eine Sequenz mit einem Element, die ihn enthält.

Auch Knoten werden mit Hilfe von Sequenzen beschrieben. Knoten haben Eigenschaften, die ihn beschreiben. Eigenschaften können auch Sequenzen sein. Jeder Elementknoten hat z. B. eine Sequenz seiner Kindknoten als Eigenschaft, ebenso eine Sequenz der Knoten seiner Attribute. Das ist in Abbildung 74 veranschaulicht. Es werden drei Elementknoten gezeigt, von denen der obere zwei Attribute und zwei Kindelemente hat.

Der Elementknoten unten links hat einen Textknoten als Kindknoten. Sequenzen sind stets durch ein Klammerpaar notiert, wobei enthaltene, als Pfeil notierte Elemente durch Kommas getrennt sind. Knoten haben weitere Eigenschaften, die in Abbildung 74 nicht gezeigt werden. Genauer werden Sequenzen in XQuery noch in Kapitel 7.5.2 vorgestellt, wo sie mit den von Hoare verwendeten Sequenzen verglichen werden.

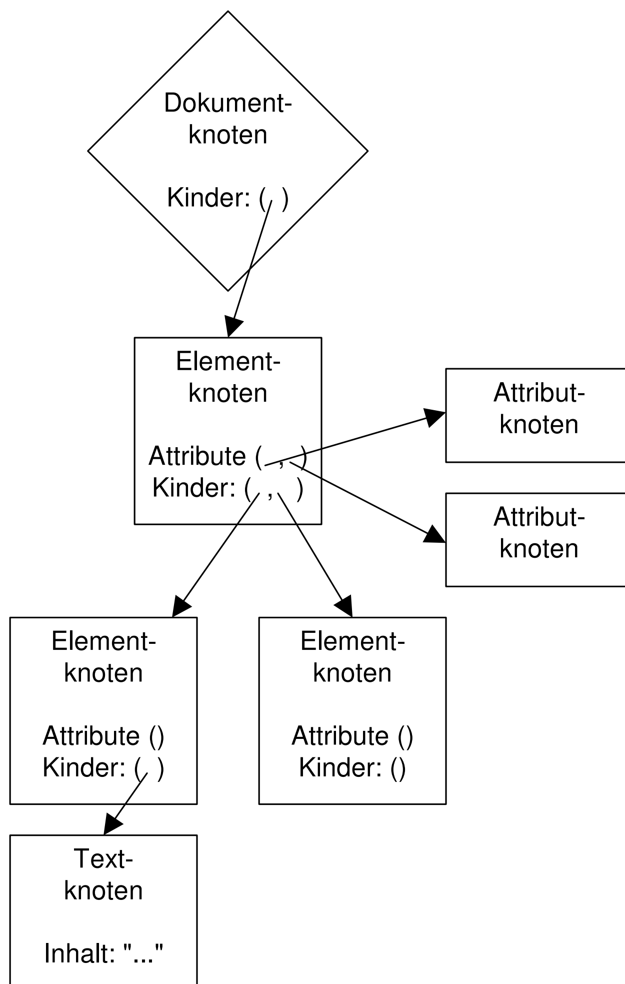

Abbildung 74: XML-Dokument, dargestellt mit Sequenzen

7.4.2 Beschreibung von Typen

XQuery enthält eine Syntax, um Typen zu beschreiben. Das ist zusätzlich zu XML-Schema erforderlich, um die in XQuery eingeführten Sequenzen beschreiben zu können. XML-Schema wird verwendet, um atomare Werte und Knoten zu beschreiben. Mit der Syntax von XQuery wird festgelegt, wie viele atomare Werte oder Knoten welcher Typen in einer Sequenz enthalten sein dürfen. Diese Syntax wird im Folgenden an Beispielen erklärt. Vollständig wird sie in [Boag 02] in Kapitel 2.4.2 beschrieben.

Für in XML-Schema definierte Typen atomarer Werte wird ihr qualifizierter Name angegeben. Z. B. bezeichnet `xsd:string` einen Typ für Zeichenketten und `xsd:boolean` für die Wahrheitswerte `true` und `false`. Gleichzeitig bezeichnen solche Typen auch Sequenzen mit genau einem atomaren Wert als Element, da ein atomarer Wert von einer Sequenz mit diesem Wert als einzigem Element ununterscheidbar ist.

Einem Typ kann ein sogenannter Occurrence-Indicator zugefügt werden um anzuzeigen, dass mit ihm bezeichnete Sequenzen auch leer sein oder mehrere Elemente enthalten dürfen. Der Occurrence-Indicator `?` erlaubt, dass ein Element in der Sequenz enthalten oder sie leer sein darf. Somit ist `xsd:string?` ein Typ für Sequenzen, die

entweder eine Zeichenkette enthalten oder leer sind. Umgekehrt legt der Occurrence-Indicator + fest, dass mindestens ein Element in der Sequenz enthalten sein muss, aber auch mehrere enthalten sein dürfen. Der Occurrence-Indicator * erlaubt schließlich eine beliebige Zahl von Elementen einschließlich der leeren Sequenz. `xsd:string*` ist also ein Typ für Sequenzen, die eine beliebige Zahl von Zeichenketten enthalten dürfen.

Typen für Knoten beginnen mit einem Schlüsselwort, dass die Art des Knotens kennzeichnet. In dieser Arbeit sollen nur die Typen für Elementknoten vorgestellt werden. Sie beginnen mit dem Schlüsselwort `element`. Das Schlüsselwort `element` alleine steht für einen Typ für beliebige Elementknoten. Der Typ kann auf in XML-Schema deklarierte Elemente oder definierte Typen präzisiert werden. Im ersten Fall wird dem Schlüsselwort `element` der deklarierte qualifizierte Name des Elementes zugefügt. Ist in XML-Schema z. B. ein Element `tra:Message` deklariert, muss ein Typ für solche Elemente in XQuery notiert werden als `element tra:Message`. Um Elementknoten auf bestimmte in XML-Schema definierte Typen einzuschränken, müssen dem Schlüsselwort `element` die Schlüsselworte `of type` und danach der qualifizierte Name des Typs zugefügt werden. Z. B. bezeichnet `element of type xsd:QName` einen Typen für Elementknoten von Typ `xsd:QName`, die also als Kindknoten einen Textknoten mit einem qualifizierten Namen haben.

Wie bei atomaren Werten kann auch bei Typen von Knoten ein Occurrence-Indicator zugefügt werden. Z. B. bezeichnet ist `element tra:Message?` einen Typ von Sequenzen, die entweder leer sind oder einen Elementknoten des Elementes `tra:Message` enthalten. Der Typ `element tra:Message*` steht für Sequenzen, die eine beliebige Zahl von Elementknoten des Elementes `tra:Message` enthalten.

In XQuery ist es auch möglich Typen für Sequenzen anzugeben, die beliebige Knoten oder sogar beliebige atomare Werte und Knoten enthalten dürfen. Als Typ für einen beliebigen Knoten wird das Schlüsselwort `node` angegeben. Wieder bezeichnet `node*` eine Sequenz mit einer beliebigen Zahl von Knoten. Wird `item` als Typ angegeben ist ein beliebiger atomarer Werte oder ein Knoten erlaubt. Damit ist `item*` ein Typ für beliebige Sequenzen.

7.4.3 Literale

XQuery erlaubt, Werte atomarer Typen im Ausdruck anzugeben. Zeichenketten und numerische Werte können direkt als Literale notiert werden, ähnlich wie in den meisten Programmiersprachen. Werte anderer atomarer Typen werden mit Konstruktorfunktionen erzeugt. Für qualifizierte Namen existiert hierzu eine spezielle Funktion.

Die für Zeichenketten und numerische Werte verwendete Syntax ist der in vielen Programmiersprachen verwendeten ähnlich. Zeichenketten werden mit Anführungszeichen (`"`) oder Hochkommas (`'`) umschlossen. Soll das zum Umschließen verwendete Zeichen in der Zeichenkette verwendet werden, muss es doppelt angegeben werden. (Z. B: `"Das ""Haus"" ist klein."`) Numerische Werte werden als Ziffern notiert, wobei zusätzlich der Dezimalpunkt (`.`) und nach einem `e` oder `E` ein Exponent erlaubt ist. Z. B. sind 42, 100.1 und 1.2E-20 gültige numerische Werte.

Atomare Werte, die nicht Zeichenketten oder numerische Werte sind, müssen mit Konstruktorfunktionen erzeugt werden. Solche gibt es in XQuery für die einfachen Typen die zu XML-Schema gehören oder mit Hilfe von XML-Schema definiert sind und für deren Definition weder `xsd:list` noch `xsd:union` verwendet wurde. Ausgenommen ist lediglich der Typ `xsd:QName`. Konstruktorfunktionen haben den qualifizierten Namen des Typs dessen Werte sie erzeugen. Als Parameter kann der Wert als Zeichenkette an-

gegeben werden. Z. B. liefert der Ausdruck `xsd:anyURI("http://example.org/")` einen Wert vom Typ `xsd:anyURI`, der die URI `http://example.org/` enthält. Analog liefert der Ausdruck `xsd:boolean("true")` den logischen Wert `true`. Zum Erzeugen der beiden logische Werte stellt XQuery jedoch zusätzlich die Funktionen `fn:true`[43] und `fn:false` zur Verfügung, die die logischen Werte `true` bzw. `false` liefern.

Für den Typ `xsd:QName` gibt es keine Konstruktorfunktion. Qualifizierte Namen werden mit der Funktion `fn:QName-in-context` erzeugt, die zwei Parameter hat. Im ersten wird der qualifiziere Name als Zeichenkette angegeben. Ein im Namen enthaltenes Präfix muss mit den Mechanismen von XQuery an einen Namensraum gebunden sein. Der zweite Parameter von Typ `xsd:boolean` bestimmt was geschieht, wenn kein Präfix angegeben ist. Wird der Wert `false` angegeben, ist der erzeugte qualifizierte Name in keinem Namensraum. Bei `true` befindet sich der qualifizierte Name im Defaultnamensraum, der für Elemente festgelegt ist. Ist das Präfix `tra` z. B. an den Namensraum `http://ti5.tu-harburg.de/venzke/20021015/traces` gebunden, liefert der Ausdruck `fn:QName-in-context("tra:Message", fn:false())` einen qualifizierten Namen in diesem Namensraum mit lokalen Namen `Message`.

7.4.4 Beispiele für Operationen und Funktionen

XQuery stellt eine Vielzahl von Operationen und Funktionen bereit, die in Ausdrücken verwendet werden können. Mit Vergleichsoperationen können z. B. atomare Werte und Knoten verglichen werden. Es gibt die in vielen Programmiersprachen übliche Operationen und Funktionen für logische und numerische Werte sowie Zeichenketten. Außerdem werden Operationen und Funktionen für Sequenzen bereitgestellt. In dieser Arbeit benötigte sollen im Folgenden kurz vorgestellt werden.

Während für Aufrufe von Funktionen stets die gleiche Syntax verwendet wird, ist für jede Operation eine spezielle Syntax definiert. Die Operationen werden in [Boag 02] beschrieben. Von XQuery zur Verfügung gestellte Funktionen beschreibt dagegen [Malhotra 02]. Zusätzliche Funktionen können benutzerdefiniert im Query-Prolog definiert werden, was noch in Kapitel 7.4.10 beschrieben wird. Alle Funktionen haben qualifizierte Namen, die von XQuery zur Verfügung gestellten befinden sich im Namensraum `http://www.w3.org/2002/11/xquery-functions`, für den in dieser Arbeit das Präfix `fn` verwendet wird.

Vergleichsoperationen lassen sich in vier Gruppen einteilen: Wertevergleichsoperationen, universelle Vergleichsoperationen, Knotenvergleichsoperationen und Reihenfolgevergleichsoperationen. Die 6 Wertevergleichsoperationen vergleichen zwei atomare Werte. Sie prüfen, ob ein atomarer Wert gleich (`eq`), ungleich (`ne`), kleiner (`lt`), kleiner oder gleich (`le`), größer (`gt`) oder größer oder gleich (`ge`) einem anderen atomaren Wert ist. Der Ausdruck `(1 eq 2)` liefert z. B. `false`, weil die Zahl 1 nicht gleich der Zahl 2 ist. Entsprechend liefert der Ausdruck `("Hallo" ne "Du")` den Wert `true`, weil sich die beiden verglichenen Zeichenketten unterscheiden.

Wird bei einer Operation für einen Wertevergleich ein Knoten angegeben, wird versucht, den Knoten in einen Wert umzuwandeln. Bei diesem als Atomization bezeichneten Vorgang, wird aus dem Knoten sein typisierter Wert ermittelt. Die Atomization wird in XQuery auch in anderen Fällen verwendet, in denen ein Knoten angegeben und ein atomarer Wert erwartet wird.

---

[43] Das Präfix `fn` stehe für den Namensraum `http://www.w3.org/2002/11/xquery-functions`, in dem sich die zu XQuery definierten Funktionen befinden.

Zu jeder Wertevergleichsoperation gibt es eine universelle Vergleichsoperation. Außer einzelnen Werten dürfen diese auch Sequenzen mit mehreren Werten als Operanden haben. Sie liefern genau dann den logischen Wert `true`, wenn es in beiden Sequenzen jeweils einen Wert gibt, so dass die zugehörige Wertevergleichsoperation für dieses Wertepaar `true` liefern würde. Z. B. ist die Operation `=` die zugehörige universelle Vergleichsoperation zu `eq`, `!=` die zu `ne` und `<=` die zu `le`. Der Ausdruck `(2 = (1,2,3))` vergleicht den atomaren Wert 2 mit der Sequenz `(1,2,3)` auf Gleichheit. Er liefert den Wert `true`, weil in der Sequenz auch der Wert 2 enthalten ist. Allerdings liefert auch der Ausdruck `(2 != (1,2,3))` den Wert `true`, weil sowohl 1 als auch 3 nicht gleich 2 sind.

Knotenvergleichsoperationen haben zwei Knoten als Operanden und prüfen, ob diese die gleiche Identität haben. Die Operation `is` liefert `true`, wenn das der Fall ist, die Operation `isnot`, wenn nicht.

Für einen Knoten kann außerdem mit Reihenfolgevergleichsoperationen geprüft werden, ob er sich in der Dokumentenreihenfolge vor oder nach einem anderen befindet. Die Dokumentenreihenfolge ist eine Ordnungsrelation für Knoten. Innerhalb des gleichen XML-Dokumentes wird sie durch die Reihenfolge bestimmen, in der die Knoten in ihm enthalten sind. Elternknoten befinden sich grundsätzlich vor ihren Nachfahren. Die Reihenfolgevergleichsoperationen `<<` liefert genau dann den logischen Wert `true`, wenn sich der linke Operand in der Dokumentenreihenfolge vor dem rechten befindet. Analog liefert `>>` genau dann `true`, wenn sich der rechte vor dem linken befindet.

Neben Vergleichsoperationen gibt es auch Operationen und Funktionen für logische und arithmetische Werte, Zeichenketten und Sequenzen. Für logische Werte werden die beiden binären Operationen `and` und `or` sowie die Funktion `fn:not` zur Verfügung gestellt. Binäre arithmetische Operationen sind `+`, `-`, `*`, `div`, `idiv` und `mod`. Die Operationen `div` und `idiv` stehen für die Division, wobei `div` einen Wert vom Typ `xsd:double` liefert, während die Operanden und das Ergebnis von `idiv` den Typ `xsd:integer` haben. Die Operation `mod` berechnet den Rest bei der Division. Zusätzlich zu binären arithmetischen Operationen werden auch die beiden unären Operationen `+` und `-` unterstützt.

Für Zeichenketten gibt es eine Reihe von Funktionen. Z. B. können zwei Zeichenketten mit der Funktion `fn:concat` konkateniert werden. Die Funktionen `fn:starts-with` und `fn:contains` prüfen, ob eine Zeichenkette mit einer anderen beginnt bzw. in ihr enthalten ist. Die Länge einer Zeichenkette kann mit der Funktion `fn:string-length` bestimmt werden.

Für Sequenzen gibt es unterschiedliche Funktionen und Operationen. So kann mit der Funktion `fn:count` die Anzahl der Elemente der Sequenz ermittelt werden. Z. B. liefert im Ausdruck `(fn:count( (1,2,3) ))` die Funktion `fn:count` für die Sequenz `(1,2,3)` den Wert 3. Die Funktionen `fn:empty` und `fn:exists` prüfen, ob eine Sequenz leer ist. Dann liefert `fn:empty` den Wert `true`. Die Funktion `fn:exists` liefert dagegen `true`, wenn die Sequenz nicht leer ist. Mit der Funktion `fn:remove` kann ein Element aus einer Sequenz entfernt werden. Als Parameter wird die Sequenz angegeben, sowie die Position des zu entfernenden Elementes. Z. B. entfernt der Ausdruck `(fn:remove( (3,4,5), 2))` aus der Sequenz `(3,4,5)` das zweite Element und liefert daher die Sequenz `(3,5)`. Mit andere Funktionen können Elemente eingefügt oder Teilsequenzen ermittelt werden. Operationen gibt es z. B. für das Konkatenieren von Sequenzen. Werden Mengen als Sequenzen repräsentiert, können mit den Operationen `union`, `intersect` und `except` die Vereinigungs-, Schnitt- bzw. Differenzmenge gebildet werden.

7.4.5 Eingabefunktionen

Drei besondere Funktionen für Sequenzen sind die Eingabefunktionen `fn:input`, `fn:document` und `fn:collection`. Mit ihnen kann ein XQuery-Ausdruck auf die zu verarbeitenden Eingabedaten zugreifen. Sie unterscheiden sich darin, aus welcher Quelle die Eingabedaten stammen.

Die Funktion `fn:input` liefert die sogenannte Input-Sequence. Sie hat keine Parameter. Die Input-Sequence wird dem XQuery-Prozessor, der XQuery-Ausdrücke auswertet, übergeben. Wie das geschieht ist implementierungsabhängig. Z. B. könnte dem Benutzer die Möglichkeit gegeben werden, ein oder mehrere XML-Dokumente anzugeben, die der XQuery-Ausdruck verarbeiten soll. Deren Dokumentknoten wären dann die Elemente der Input-Sequence. In jeden Fall kann die Input-Sequence als die „implizite Eingabe“ angesehen werden, die der XQuery-Ausdruck verarbeiten soll.

Die Funktion `fn:document` liefert eine Sequenz der Dokumentknoten von einem oder mehreren XML-Dokumenten, deren URIs als Parameter angegeben werden. Sie kann außerdem auch XML-Fragmente liefern. Im einfachsten Fall wird ihr nur eine URI als Zeichenkette übergeben, die ein XML-Dokument bezeichnet. Dann liefert die Funktion `fn:document` den Dokumentknoten dieses XML-Dokumentes. Z. B. liefert der Ausdruck `fn:document("http://www.example.org/sample.xml")` den Dokumentknoten des XML-Dokumentes, das unter der URL `http://www.example.org/sample.xml` mit HTTP abgefragt werden kann.

Die dritte Funktion `fn:collection` liefert eine Sammlung von Knoten, die mit einer URI bezeichnet wird. Die URI der Sammlung wird der Funktion als Zeichenkette übergeben. Die Sammlung kann eine Sequenz beliebiger Knoten sein. Die Sequenz wird von der Funktion geliefert.

7.4.6 Pfadausdrücke

Pfadausdrücke ermöglichen, bestimmte Knoten in XML-Fragmenten aufzufinden. Das Ergebnis ist eine Sequenz, die diese Knoten in Dokumentenreihenfolge enthält, jeden Knoten nur einmal. Pfadausdrücke gehören auch mit vergleichbarer Syntax zu XPath 1.0. Hier soll nur anhand einiger Beispiele der für diese Arbeit notwenige Überblick gegeben werden. Vollständig werden Pfadausdrücke in [Boag 02] in Kapitel 3.2 beschrieben. Die Syntax von Pfadausdrücken verwendetet unterschiedliche Schlüsselworte und Symbole. In der nachfolgenden Darstellung sollen sie vereinfachend als drei eigenständige Operationen eingeführt werden.

Die binäre Operation `/` liefert für eine Sequenz von Knoten eine weitere, deren Knoten mit denen der ursprünglichen Sequenz in einer Beziehung stehen. Z. B. können es Kindknoten sein. Der rechte Operand bestimmt in welcher Beziehung sie zueinander stehen und welche Art von Knoten ermittelt werden soll.

Der Ausdruck `fn:input()/tra:Trace` liefert z. B. alle Kindelemente `tra:Trace` der Knoten in der Input-Sequence. Die im vorherigen Kapitel 7.4.5 beschriebene Funktion `fn:input` liefert die Input-Sequence. Sie dient als linker Operand für die Operation `/`. Der rechte Operand ist ein qualifizierter Name. Solche haben für die Operation `/` die Bedeutung, dass zu jedem Knoten der Sequenz Kindelemente ermittelt werden sollen, die den qualifizierten Namen haben. Die Sequenz der Kindelemente wird als Ergebnis geliefert.

Durch mehrfache Anwendung der Operation `/` kann eine Sequenz der Kindelemente der Kindelemente ermittelt werden. Der Ausdruck `fn:input()/tra:Trace/tra:Message` ist ein Beispiel. Er liefert eine Sequenz aller Kindelemente `tra:Message` der Kindelemente `tra:Trace` der Knoten in der Input-Sequence.

Ist der rechte Operand ein qualifizierter Name mit vorangestelltem Zeichen `@`, wird statt der Sequenz der Kindelemente die Sequenz der Attributknoten ermittelt. Enthält z. B. die Variable `$m` eine Sequenz von Elementknoten, dann liefert der Ausdruck `($m/@to)` die Sequenz aller Knoten von ihren Attributen, die den Namen `to` (in keinem Namensraum) haben.

Statt einem qualifizierten Namen kann als rechter Operanden auch das Zeichen `*` angegeben werden, ein sogenannter Wildcard. Dann werden alle Kindelemente bzw. Attribute geliefert, unabhängig von ihrem qualifizierten Namen. Z. B. liefert `(fn:input()/*)` die Sequenz aller Kindelemente der Knoten der Input-Sequence. Analog liefert `(fn:input()/*/tra:Message)` die Sequenz der Kindelemente `tra:Message` von Kindelementen beliebiger Namen der Knoten der Input-Sequence.

Werden im rechten Operanden explizit sogenannte Achsen angegeben, können auch Sequenzen mit Knoten des linken Operanden oder deren Elternknoten geliefert werden. Wieder können Knoten auf solche mit bestimmten Namen beschränkt werden. Z. B. liefert der Ausdruck `($m/self::tra:Message)` alle Elementknoten der Sequenz in der Variablen `$m`, die den Namen `tra:Message` haben. Der Ausdruck `($m/parent::*)` liefert zu den Knoten in der Sequenz in `$m` eine Sequenz ihrer Elternknoten. Die Schlüsselworte `self` und `parent` bezeichnen die Achse, die angibt, in welcher Beziehung Knoten im linken Operanden mit denen in der gelieferten Sequenz stehen. Neben diesen beiden gibt es weitere Achsen.

Die Operation `//` ist ähnlich zu `/`. Statt Kindelementen können mit ihr jedoch alle Elemente bestimmt werden, die Nachfahren sind. Neben Kindelementen werden also auch die Kindelemente der Kindelemente, deren Kindelemente, u.s.w. geliefert. Die Knoten aller dieser Elemente können wieder auf solche mit bestimmten qualifizierten Namen eingeschränkt werden. So liefert z. B. der Ausdruck `(fn:input()//my:a)` eine Sequenz aller Elementknoten, die Nachfahren der Knoten in der Input-Sequence sind und den qualifizierten Namen `my:a` haben.

Mit einer Syntax, die ebenfalls zu den Pfadausdrücken gehört, kann eine Sequenz auf solche Elemente beschränkt werden, für die ein logischer Ausdruck (Prädikat) zutrifft. Hierzu wird das Prädikat hinter der Sequenz in eckigen Klammern angegeben. Enthält z. B. die Variable `$m` eine Sequenz mit Elementknoten, dann liefert der Ausdruck `($m[@to = "Service"])` eine Sequenz, die alle die Elementknoten enthält, auf die das Prädikat `(@to = "Service")` zugrifft.

Solche Prädikate werden relativ zu den Knoten ausgewertet, die sie testen. Mit relativen Pfadausdrücken kann implizit auf den Knoten zugegriffen werden. Der relative Pfadausdruck `@to` im obigen Prädikat ist relativ zum Elementknoten, für den geprüft wird, ob er in der gelieferten Sequenz enthalten sein soll. Er liefert dessen Attributknoten mit Namen `to`. Wäre der Elementknoten an eine Variable `$e` gebunden, wäre der relative Pfadausdruck äquivalent zu `($e/@to)`. Der Ausdruck `($m[@to = "Service"])` liefert also eine Sequenz der Elementknoten in `$m`, die ein Attribut mit Namen `to` und dem Wert `Service` haben.

Mit Prädikaten kann auch geprüft werden, ob ein Element ein bestimmtes Attribut oder einen bestimmten Nachfahren hat. Hierzu wird der relative Pfadausdruck zum Attribut bzw. zum Nachfahren als Prädikat angegeben. Enthalte die Variable `$m` wieder eine Sequenz mit Elementknoten. Dann liefert der Ausdruck `($m[@to])` eine Sequenz mit allen den Elementknoten, die ein Attribut mit Namen `to` haben. Analog liefert dann `($m[env:Envelope/env:Body])` alle die Elementknoten, die (mindestens) ein Kindelement `env:Envelope` haben, das seinerseits (mindestens) ein Kindelement `env:Body` hat.

Prädikate können auch mit den Operationen / und // kombiniert werden. So liefert der Ausdruck ($m/tra:Message[@to]) eine Sequenz mit den Elementknoten tra:Message, die Kindelemente eines Elementknotens der Sequenz in $m sind und die ein Attribut mit Namen to haben. Das Ergebnis eines solchen Ausdrucks kann wieder Operand für die Operation / sein. So liefert der Ausdruck ($m/tra:Message[@to]/env:Envelope) die Kindknoten env:Envelope der Elementknoten tra:Message in der eben beschriebenen Sequenz.

Mit einer Zahl als Prädikat kann angegeben werden, dass nur ein bestimmtes Element einer Sequenz geliefert werden soll. Z. B. liefert der Ausdruck ((10,11,12)[2]) das zweite Element der Sequenz (10,11,12), also 11. Der Pfadausdruck ($m/tra:Message[2]) liefert das zweite Kindelement tra:Message jedes Knotens in der Sequenz in $m. Enthält $m nur einen Knoten, wird dessen zweites Kindelement tra:Message geliefert.

7.4.7 Bedingte Ausdrücke

Das von bedingten Ausdrücken gelieferte Ergebnis wird von einem von zwei Teilausdrücken berechnet. Ein dritter, logischer Teilausdruck bestimmt, welcher von beiden gewählt wird. Abbildung 75 zeigt ein Beispiel für einen bedingten Ausdruck. Hinter dem Schlüsselwort if wird der logische Ausdruck angegeben. Liefert er den Wert true, wird das Ergebnis durch den Ausdruck nach dem Schlüsselwort then bestimmt, ansonsten durch den nach dem Schlüsselwort else. Der Ausdruck in Abbildung 75 liefert also die Zeichenkette „empty“, wenn die Variable $m eine leere Sequenz enthält, ansonsten die Zeichenkette „not empty“.

```
if (fn:empty ($m))
  then "empty"
  else "not empty"
```

Abbildung 75: Beispiel für bedingten Ausdruck in XQuery

7.4.8 FLWOR-Ausdrücke

FLWOR-Ausdrücke erlauben es, berechnete Zwischenergebnisse an Variablen zu binden, über Sequenzen zu iterieren und sie zu sortieren. FLWOR steht für die fünf Schlüsselwörter, die in solchen Ausdrucke verwendet werden: for, let, where, order by und return. Nachfolgend werden nur einfache FLWOR-Ausdrücke vorgestellt, wie sie in dieser Arbeit verwendet werden.

Um ein Zwischenergebnis an eine Variable zu binden, wird das Schlüsselwort let verwendet. Ihm folgt der Name der Variable, wie im Beispiel in Abbildung 76 gezeigt. An sie wird das Zwischenergebnis gebunden, das der Ausdruck nach dem Symbol := berechnet. Die Variable kann dann im Ausdruck nach dem Schlüsselwort return verwendet werden, der das Ergebnis des FLWOR-Ausdrucks berechnet. In Abbildung 76 wird an die Variable $m die Sequenz von Elementknoten tra:Message gebunden, die der Pfadausdruck (fn:input()/tra:Trace/tra:Message) berechnet. Im Ausdruck nach dem Schlüsselwort return wird zweimal über die Variable $m auf die Sequenz verwiesen.

```
let $m := fn:input()/tra:Trace/tra:Message
  return ($m[1]/@to = "Service") and (fn:count ($m) lt 3)
```

Abbildung 76: Beispiel für FLWOR-Ausdruck mit let in XQuery

Für das Iterieren über eine Sequenz wird das Schlüsselwort `for` verwendet. Ihm folgt der Name der Variablen, an die nacheinander alle Elemente der Sequenz gebunden werden. Danach folgt nach dem Schlüsselwort `in` der Ausdruck, der die Sequenz berechnet. In Abbildung 77 ist dort direkt die Sequenz `(1,2,3)` angegeben. Für jedes Element der Sequenz wird der Ausdruck nach dem Schlüsselwort `return` einmal berechnet. Die Sequenz der berechneten Ergebnisse wird vom FLWOR-Ausdruck als Ergebnis geliefert. Der Ausdruck in Abbildung 77 liefert also die Sequenz `(2,4,6)`, weil der Teilausdruck `($m * 2)` jeden Wert der Sequenz `(1,2,3)` verdoppelt.

```
for $m in (1, 2, 3)
  return $m * 2
```

Abbildung 77: Beispiel für FLWOR-Ausdruck mit `for` in XQuery

Sollen nicht alle Elemente einer Sequenz verarbeitet werden, wird zusätzlich zu `for` das Schlüsselwort `where` verwendet. Nur wenn der ihm folgende logische Ausdruck für ein Element zutrifft, wird es verarbeitet. Ansonsten wird das Element für das Ergebnis des FLWOR-Ausdrucks nicht berücksichtigt. Im Beispiel in Abbildung 78 prüft der logische Ausdruck, ob das Element kleiner als 10 ist. Aus der Sequenz `(11,42,3,7,15,6)` trifft das nur auf die Zahlen 3, 7 und 6 zu. Nur diese werden durch den Ausdruck `($m * 2)` verdoppelt. Daher liefert der FLWOR-Ausdruck die Sequenz `(6,14,12)`.

```
for $m in (11, 42, 3, 7, 15, 6)
  where ($m lt 10)
  return $m * 2
```

Abbildung 78: Beispiel für FLWOR-Ausdruck mit `for` und `where` in XQuery

Die Reihenfolge der Elemente der gelieferten Sequenz wird durch den FLWOR-Ausdruck nicht verändert, wenn nicht sortiert werden soll. Hierin unterscheiden sich FLWOR-Ausdrücke von Pfadausdrücken, die stets Sequenzen von Knoten in Dokumentenreihenfolge liefern. Mit FLWOR-Ausdrücken ist es jedoch auch möglich, Sequenzen explizit zu sortieren, wozu das Schlüsselwort `order by` verwendet wird. (Siehe [Boag 02], Kapitel 3.8.3.)

7.4.9 Quantoren

Mit den beiden Quantoren `every` und `some` kann für eine Sequenz geprüft werden, ob für alle ihre Elemente bzw. zumindert für ein Element ein logischer Ausdruck zutrifft. Sie entsprechen den Quantoren „für alle" (∀) und „es gibt" (∃) aus der Prädikatenlogik. Wie bei FLWOR-Ausdrücken wird jedes Element der Sequenz nacheinander an eine Variable gebunden. Diese wird dann im logischen Ausdruck verwendet.

Abbildung 79 zeigt ein Beispiel für einen Ausdruck mit dem Quantor `every`. Nach dem Schlüsselwort `every` folgt der Name der Variablen `$m`, an die nacheinander die Elemente der Sequenz gebunden werden. Die Sequenz wird durch den Ausdruck nach dem Schlüsselwort `in` berechnet. Im Beispiel ist die Sequenz `(2,41,7)` direkt angegeben. Konzeptionell wird für jedes Element der Sequenz der Teilausdruck hinter dem Schlüsselwort `satisfies` berechnet. Wird nur für ein Element der logische Wert `false` geliefert, liefert auch der Quantor den logischen Wert `false`, ansonsten `true`. Ist die Sequenz also leer, wird also der logische Wert `true` geliefert. Der Ausdruck in Abbildung 79 liefert den logischen Wert `false`, weil das zweite Element größer als 10 ist und daher für dieses der Teilausdruck `($m le 10)` den logischen Wert `false` liefert.

```
every $m
  in (2, 41, 7)
  satisfies $m le 10
```

Abbildung 79: Beispiel für Ausdruck mit Quantor `every` in XQuery

Der Quantor `some` unterscheidet sich syntaktisch von `every` nur durch das erste Schlüsselwort. Er wird auch ähnlich ausgewertet. Er liefert genau dann den logischen Wert `true`, wenn der Teilausdruck hinter dem Schlüsselwort `satisfies` zumindest für ein Element der Sequenz hinter dem Schlüsselwort `in` den logischen Wert `true` liefert. Ist die Sequenz leer, ist das Ergebnis daher der logische Wert `false`. Das Beispiel in Abbildung 80 liefert den logischen Wert `true`. Sowohl das erste als auch das dritte Element der Sequenz `(2,41,7)` sind kleiner oder gleich 10, was der Teilausdruck `($m le 10)` prüft.

```
some $m
  in (2, 41, 7)
  satisfies $m le 10
```

Abbildung 80: Beispiel für Ausdruck mit Quantor `some` in XQuery

7.4.10 Query-Prolog

XQuery-Ausdrücke, wie die bisher vorgestellten, können um den sogenannten Query-Prolog ergänzt werden. Er enthält Deklarationen, die für den Ausdruck als ganzes gültig sind. Z. B. werden dort im Ausdruck verwendete Präfixe an die URIs von Namensräumen gebunden. Außerdem können dort benutzerdefinierte Funktionen definiert werden. Syntaktisch wird der Query-Prolog dem Ausdruck vorangestellt.

Ein Präfix wird an die URI eines Namensraumes mit den Schlüsselworten `declare` und `namespace` gebunden. Danach folgt der Name des Präfixes, das Symbol `=` und dann die URI des Namensraumes als Zeichenkette. Im XQuery-Ausdruck in Abbildung 76 werden die beiden Präfixe `tra` und `fn` verwendet. In Abbildung 81 ist ihm ein Query-Prolog mit Deklarationen für diese Präfixe zugefügt worden. Grundsätzlich müssen alle verwendeten Präfixe deklariert werden. Das Präfix `fn` gehört jedoch zu einem von vier in XQuery vordeklarierten Präfixen. Er ist an die in Abbildung 81 gezeigte URI des Namensraumes gebunden, in dem sich die von XQuery zur Verfügung gestellten Funktionen aus [Malhotra 02] befinden.

```
declare namespace tra="http://ti5.tu-harburg.de/venzke/20021015/traces"
declare namespace fn="http://www.w3.org/2002/11/xquery-functions"
let $m := fn:input()/tra:Trace/tra:Message
  return ($m[1]/@to = "Service") and (fn:count ($m) lt 3)
```

Abbildung 81: Beispiel für Ausdruck mit Query-Prolog in XQuery

Neben den von XQuery zur Verfügung gestellten Funktionen können dem Query-Prolog benutzerdefinierte zugefügt werden. Solche kapseln einen Teilausdruck, der an verschiedenen Stellen im XQuery-Ausdruck wiederverwendet werden kann. Das erlaubt, den XQuery-Ausdruck übersichtlicher zu notieren.

Abbildung 82 zeigt einen XQuery-Ausdruck in dem eine Funktion definiert und verwendet wird. Die Deklaration der Funktion beginnt mit den Schlüsselworten `define` und `function`, danach folgt der Name der Funktion. Funktionen haben qualifizierte Namen. Im Beispiel ist es `my:filter`. Der gleiche Name darf nur für eine Funktion verwendet werden, das Überladen von Funktionen ist nicht erlaubt.

```
declare namespace tra="http://ti5.tu-harburg.de/venzke/20021015/traces"
declare namespace my="http://example.org/sample"

define function my:filter ($t as element tra:Trace,
                           $toVal as xsd:string)
  as element tra:Message*
{ $t/tra:Message[@to = $toVal] }

my:filter(fn:input()/tra:Trace[1], "Service")
```

Abbildung 82: Beispiel für Ausdruck mit benutzerdefinierter Funktion in XQuery

Nach dem Namen der Funktion folgen in runden Klammern die formalen Parameter der Funktion, durch Kommas getrennt. Für jeden Parameter wird ein Name angegeben. Danach kann nach dem Schlüsselwort `as` der Typ des Parameters festgelegt werden. Ist kein Typ angegeben, wird `xsd:anyType` angenommen. Im Beispiel hat der erste Parameter `$t` den Typ `element tra:Trace`. Erwartet wird also ein Elementknoten mit qualifiziertem Namen `tra:Trace`. Der zweite Parameter `$toVal` hat den Typ `xsd:string`, so dass für ihn eine Zeichenkette erwartet wird.

Auch für den Rückgabewert der Funktion kann der Typ festgelegt werden. Der Typ wird nach den Parametern und dem Schlüsselwort `as` angegeben. Fehlt diese Deklaration wird wieder der Typ `xsd:anyType` angenommen. Im Beispiel ist der Typ `element tra:Message*`. Die Funktion gibt also eine Sequenz mit beliebig vielen Elementknoten `tra:Message` zurück.

Danach folgt in geschweiften Klammen der Ausdruck, der das Ergebnis der Funktion berechnet. In ihm können die formalen Parameter wie Variablen verwendet werden. Auch Rekursion ist erlaubt. Im Beispiel wird das Ergebnis der Funktion durch einen Pfadausdruck berechnet.

Aufgerufen werden benutzerdefinierte Funktionen ebenso, wie die von XQuery zur Verfügung gestellten. Der XQuery-Ausdruck in Abbildung 82 ruft die benutzerdefinierte Funktion `my:filter` auf, um das Ergebnis des XQuery-Ausdrucks zu berechnen.

Obwohl der Query-Prolog in der Regel für XQuery-Ausdrücke erforderlich ist, wird er in vielen Beispielen dieser Arbeit nicht mit gezeigt. So kann die Darstellung verkürzt werden. In den XQuery-Ausdrücken verwendete Präfixe werden im Text eingeführt und sind dem Leser daher bekannt. Verwendete Funktionen werden erklärt.

7.4.11 Fehlerbehandlung

Bei der Verarbeitung von XQuery-Ausdrücken können Fehler auftreten. Unterschieden wird zwischen statischen und dynamischen Fehlern. Statische Fehler werden in der Analysephase erkannt, in der der XQuery-Ausdruck unabhängig von den zu verarbeitenden Eingabedaten geprüft wird. Zu ihnen gehören statische Typfehler[44], die bereits in Kapitel 7.4.1 erwähnt wurden, aber auch syntaktische Fehler im XQuery-Ausdruck.

Dynamische Fehler treten erst in der Ausführungsphase auf, wenn der XQuery-Ausdruck abhängig von den Eingabedaten ausgewertet wird. Hierzu gehören in der Ausführungsphase erkannte Typfehler (siehe Kapitel 7.4.1). Außerdem gehören dazu numerische Überläufe und das Fehlen von Eingabedaten, die mit den Operationen `fn:document` oder `fn:collection` ermittelt werden. Außerdem stellt XQuery die Funktion `fn:error` zur Verfügung, mit der explizit ein dynamischer Fehler erzeugt werden kann.

---

[44] In [Boag 02] werden Typfehler nicht als statische bzw. dynamische Fehler bezeichnet. Sie bilden eine dritte Kategorie. In dieser Arbeit sollen sie jedoch für eine einheitlichere Darstellung als statische bzw. dynamische Fehler angesehen werden.

7.5 XQuery für logische Aussagen von Spezifikationen

Die Sprache XQuery soll in SXQT, dem Spezifikationsverfahren dieser Arbeit, zum Notieren prädikatenlogischer Ausdrücke verwendet werden, um Anforderungen an Schnittstellen von Webservices zu formulieren. Solche Ausdrücke sollen nachfolgend als SXQT-Ausdrücke bezeichnet werden. In diesem Kapitel wird vorgestellt, was für die Konstruktion von SXQT-Ausdrücken zu berücksichtigen ist. Besonders wird auf die Darstellung von Traces in XQuery eingegangen. Zwei Anforderungen dienen als Beispiele, um zu demonstrieren, wie sie als SXQT-Ausdrücke formuliert werden können.

7.5.1 Grundidee

Anforderungen an Schnittstellen von Webservices, die nicht in WSDL ausgedrückt werden können, werden als SXQT-Ausdrücke formuliert. Sie sind XQuery-Ausdrücke, die einen logischen Wert liefern, also einen Wert vom Typ `xsd:boolean`. Trifft die Anforderung für einen beobachteten Trace zu, muss der SXQT-Ausdruck den logischen Wert `true` liefern. Trifft die Anforderung nicht zu, muss der logische Wert `false` geliefert werden.

Durch Fehler kann die Ausführung eines SXQT-Ausdrucks unmöglich sein. In Kapitel 7.4.11 wurden statische und dynamische Fehler voneinander unterschieden. Da statische Fehler unabhängig von einem beobachteten Trace auftreten, sind sie als Fehler im SXQT-Ausdruck anzusehen.

Anders ist es bei dynamischen Fehlern. Sie können z. B. auftreten, wenn die zu verarbeiteten Daten nicht so aufgebaut sind, wie es der Entwickler des XQuery-Ausdrucks erwartet hat. Im Falle eines SXQT-Ausdrucks könnte z. B. eine im Trace enthaltene SOAP-Nachricht nicht so aufgebaut sein, wie es für den Ausdruck vorausgesetzt wurde. Damit widerspricht der Trace dem, was für die SXQT-Spezifikation erwartet wurde. Deshalb sollen dynamische Fehler als Spezifikationsverstöße behandelt werden. Tritt ein dynamischer Fehler für einen SXQT-Ausdruck auf, widerspricht der beobachtete Trace der SXQT-Spezifikation.

Mit diesen Annahmen ist XQuery eine attraktive Sprache zum Formulieren von Anforderungen. Sie bietet eine Reihe von Operationen, mit denen Eigenschaften von Werten geprüft werden können. Hierzu gehören vor allem die Vergleichsoperationen mit denen Werte auf unterschiedliche Arten miteinander verglichen werden können (siehe Kapitel 7.4.4). Außerdem bietet XQuery die in der Aussagenlogik gebräuchlichen logische Operationen für die Konjunktion (`and`) und die Disjunktion (`or`). Die Negation wird als Funktion fn:`not` zur Verfügung gestellt. Auch die in der Prädikatenlogik üblichen Quantoren, der Allquantor ($\forall$) und der Existenzquantor ($\exists$), sind verfügbar (siehe Kapitel 7.4.9).

Neben diesen Möglichkeiten zum Formulieren logischer Ausdrücke enthält XQuery vor allem durch seine Pfadausdrücken die Möglichkeit, mit XML-Dokumenten und -Fragmenten umzugehen. Aus ihnen können Fragmente oder Werte aus einem XML-Dokument extrahiert werden, so dass diese miteinander in Beziehung gesetzt werden können, wozu die Vergleichsoperationen oder Funktionen verwendet werden können. Z. B. ist es so möglich, die Anzahl von Elementen mit bestimmten Eigenschaften im XML-Dokument zu bestimmen und sie mit dem Wert eines Attributes in Beziehung zu setzen.

7.5.2 Sequenzen in XQuery und bei Hoare

In XQuery gibt es Sequenzen, die als Basis zur Darstellung von Traces benötigt werden. Bei Hoare sind Traces Sequenzen aus Ereignisnamen. Auch für diese Arbeit wurde im

Kapitel 7.2.2 festgelegt, dass ein Trace eine Sequenz ist, bei der jedes Element eine Beobachtung eines Ereignisses beschreibt. Um eine Basis für die Darstellung von Traces in XQuery zu schaffen, sollen nachfolgend die Sequenzen von XQuery mit denen von Hoare verglichen werden.

Sequenzen haben bei Hoare und in XQuery eine große Ähnlichkeit. Für XQuery sind sie in [Fernandez 02] beschrieben und wurden bereits in Kapitel 7.4.1 erwähnt. Eine Sequenz ist eine geordnete Sammlung von 0 oder mehr Elementen. In ihr kann dasselbe Element auch mehrfach vorkommen. Enthält die Sequenz kein Element, wird sie als leere Sequenz bezeichnet.

Bei Hoare und in XQuery werden Sequenzen ähnlich notiert. Hoare trennt die Elemente durch Kommas und umschließt die Sequenz mit spitzen Klammern. `<1,2,3>` ist ein Beispiel für eine Sequenz, die als Elemente die drei Zahlen 1, 2 und 3 in dieser Reihenfolge enthält. In XQuery werden statt der spitzen Klammern runde verwendet werden. Die eben genannte Sequenz würde als `(1,2,3)` notiert. Die leere Sequenz wird bei Hoare als `<>` geschrieben und in XQuery als `()`.

Obwohl in beiden Fällen eine Sequenz offenbar sehr ähnlich notiert werden kann, ist interessant, dass die Notation genaugenommen eine unterschiedliche Bedeutung hat. Bei Hoare handelt es sich um eine Notation, mit der eine Sequenz direkt angegeben werden kann. In XQuery steht ein Komma jedoch für die Operation, die zwei Sequenzen konkateniert, also eine neue Sequenz erzeugt, die zunächst die Elementen des linken Operanden und danach die Elemente des rechten Operanden enthält. Eine solche Operation ist auch bei Hoare definiert, es ist die Operation `Concatenation`. (Siehe Kapitel 7.1.3.)

Die Operation `Concatenation` kann zum Erzeugen neuer Sequenzen verwendet werden, weil in XQuery Sequenzen, die nur ein Element enthalten, von diesem Element ununterscheidbar sind. Somit bezeichnet z. B. die Zahl 1 neben dem atomaren Wert auch eine Sequenz, die diesen Wert enthält. Damit werden durch den Ausdruck `(1,2)` zunächst zwei Sequenzen mit jeweils einem Element (1 bzw. 2) erzeugt und diese danach zu einer Sequenz konkateniert. Analog werden durch den Ausdruck `(1,2,3)` zunächst 3 Sequenzen erzeugt, die dann zu einer konkateniert werden. Das würde bei Hoare durch den Ausdruck `(<1>^<2>^<3>)` notiert.

Ein weitere Unterschied zwischen Sequenzen von Hoare und in XQuery ist, welche Elemente eine Sequenz enthalten kann. Hierzu gibt es bei Hoare keine Einschränkungen. Insbesondere kann eine Sequenz auch Sequenzen als Elemente enthalten. Z. B. bezeichnet `<<1,2>,<3,4>>` eine Sequenz mit zwei Elementen. Beides sind Sequenzen, die selbst zwei Elemente enthalten.

Diese Sequenz ist bei XQuery nicht erlaubt. Sequenzen dürfen als Elemente keine Sequenzen enthalten. Erlaubt sind nur atomare Werte oder Knoten. Atomare Werte sind Werte der einfachen Typen von XML-Schema, also z. B. Zahlen oder Zeichenketten. Knoten sind die Entitäten, die in einem XML-Dokument die Baumstruktur bilden. Es sind die 7 Knotentypen Dokument, Element, Attribut, Text, Namensraum, Processing-Instruction und Kommentar (siehe [Fernandez 02]).

Obwohl keine Sequenzen von Sequenzen erlaubt sind, ermöglichen Sequenzen von Knoten, die baumartige Struktur von XML-Dokumenten abzubilden. Dazu ist zu beachten, dass Knoten selbst Sequenzen mit anderen Knoten enthalten können. Z. B. enthält jeder Elementknoten eine Sequenz, die seine Kindelemente als Knoten enthält. In einer weiteren Sequenz sind die Knoten seiner Attribute enthalten.

```
<?xml version="1.0" encoding="UTF-8"?>
<A>
  <B D="1" />
  <C D="2" />
</A>
```

Abbildung 83: XML-Dokument zur Veranschaulichung von Sequenzen

So würde in XQuery das einfache XML-Dokument in Abbildung 83 als eine Sequenz dargestellt, die nur den Dokumentknoten enthält. Der Dokumentknoten hat eine Sequenz für seine Kindknoten. Sie enthält im Beispiel den Knoten des Elementes A. In diesem Knoten ist die Sequenz für Attribute leer. Die Sequenz für die Kindelemente enthält die beiden Knoten der Elemente B und C. Bei diesen beiden Knoten ist die Sequenz für die Kindelemente leer. Ihre Sequenzen für die Attribute enthalten dagegen jeweils einen Knoten für ein Attribut D.

Während Knoten in XQuery eine eigene Identität besitzen, ist das für Sequenzen nicht der Fall. Die Identität von Knoten ist unabhängig von ihrem Inhalt. Zwei Knoten können auch bei gleichem Inhalt eine unterschiedliche Identität haben. Ist ein Knoten in unterschiedlichen Sequenzen enthalten, kann daher trotzdem festgestellt werden, ob es sich um den gleichen Knoten handelt. Das ist bei Sequenzen anders. Sie haben keine eigene Identität. Ein Vergleich von Sequenzen kann daher nur durch den Vergleich ihrer Inhalte erfolgen. Zwei Sequenzen sind gleich, wenn sie die gleichen Elemente in der gleichen Reihenfolge enthalten.

7.5.3 Traces in SXQT-Ausdrücken

Mit Hilfe der Sequenzen von XQuery können Traces dargestellt werden. Die in Kapitel 7.2.2 auf Seite 125 beschriebenen Traces sind Sequenzen von Knoten von Elementen `tra:Message` (Message-Elemente). Jedes Message-Element enthält die Informationen zur Beobachtung einer SOAP-Nachricht. Die beobachtete SOAP-Nachricht ist selbst Kindelement des Message-Elementes. Mit den Attributen `to` und `operation` wird festgehalten, in welcher Richtung die SOAP-Nachricht ausgetauscht wurde und zu welchem Request-/Response-Paar sie gehört.

Solche Traces werden in logischen Ausdrücken verwendet. Hoare verwendet hierzu in den Ausdrücken eine freie Variable `tr`, die den beobachteten Trace bezeichnet, wie in Kapitel 7.1.4 dargestellt. Soll ein logischer Ausdruck in XQuery formuliert werden, stellt sich das Problem, dass XQuery keine freien Variablen zulässt. Alle Variablen müssen explizit an Werte gebunden werden. Das ist mit FLWOR-Ausdrücken (siehe Kapitel 7.4.8) oder Quantoren (siehe Kapitel 7.4.9) möglich. In beiden Fällen muss ein Ausdruck angegeben werden, der den Wert berechnet. Da sich der Trace jedoch nicht berechnen lässt, muss ein anderer Weg festgelegt werden, aus SXQT-Ausdrücken auf den Trace zu verweisen.

In dieser Arbeit soll der Trace mit Hilfe der Input-Sequence von XQuery ermittelt werden, die nach dem XQuery-Standard [Boag 02] die „implizite Eingabe" eines XQuery-Ausdrucks ist. Auf sie kann aus einem SXQT-Ausdruck mit der Funktion `fn:input` verwiesen werden. Auf diese Art auf den Trace zu verweisen, ist nahe liegend, weil ein SXQT-Ausdruck den Trace auf einen logischen Wert abbildet und damit der Trace als dessen „implizite Eingabe" angesehen werden kann.

Der XQuery-Standard [Boag 02] legt nicht fest, was die Input-Sequence enthält. Das muss implementierungsspezifisch für jeden XQuery-Prozessor, der XQuery-Ausdrücke auswertet, festgelegt werden. Für SXQT-Ausdrücke muss der Trace an die Input-Sequence gebunden werden.

Nach dem XQuery-Standard [Boag 02] darf die Input-Sequence eine Sequenz beliebiger Knoten sein. Es könnte jedoch XQuery-Prozessoren geben, die nur vollständige XML-Dokumente als Input-Sequence zulassen. Um die Verwendung solcher XQuery-Prozessoren zu ermöglichen, soll der Trace als vollständiges XML-Dokument in der Input-Sequence enthalten sein. In Kapitel 7.2.2 wurde vorgestellt, wie sich ein Trace zu einem XML-Dokument vervollständigen lässt. Hierzu wird die Sequenz aus Message-Elementen von einem Element `tra:Trace` „umschlossen“. Das Element `tra:Trace` bekommt die Message-Elemente also als Kindelemente.

Für SXQT sei also festgelegt, dass zur Auswertung von SXQT-Ausdrücken die Input-Sequence an ein XML-Dokument gebunden ist, dass den Trace enthält. Der Dokumentknoten des XML-Dokumentes ist das einzige Element der Input-Sequence. Der Trace selbst, also die Sequenz von Message-Elementen, kann innerhalb eines SXQT-Ausdrucks mit den Pfadausdruck `fn:input()/tra:Trace/tra:Message` ermittelt werden.

7.5.4 Beispiel für Anforderung an einzelne SOAP-Nachrichten

Um zu veranschaulichen, wie mit den gegebenen Annahmen SXQT-Ausdrücke formuliert werden können, sollen im Folgenden zwei Beispiele gegeben werden. In den Kapiteln 6.1 und 6.2 wurde eine Reihe von Anforderungen vorgestellt, die sich nicht in WSDL ausdrücken lassen. Zwei dieser Anforderungen aus dem Kapitel 6.1 sollen nun als Beispiele dienen. In diesem Kapitel 7.5.4 wird eine Anforderung an einzelne SOAP-Nachrichten betrachtet. Im folgenden Kapitel 7.5.5 geht es um eine Anforderung an einzelne Request-/Response-Paare.

Das erste Beispiel bezieht sich auf eine Anforderung aus Kapitel 6.1.1. Diese soll als SXQT-Ausdruck formuliert werden. Zur Anforderung zeigt Abbildung 51 auf Seite 98 einen Response der Operation `Categories` der Webkomponente für eine Internetzeitung. Im Response wird geliefert, welche Rubriken die Internetzeitung hat. Dabei ist der Name jeder Rubrik in einem Element `nwst:Category` enthalten ist. Zusätzlich enthält der Response die Anzahl der Rubriken in einem Element `nwst:NoCategories`.

Im Kapitel 6.1.1 wurde festgestellt, dass sich in WSDL nicht die Anforderung ausdrücken lässt, dass die im Element `nwst:NoCategories` enthaltene Zahl gerade die Anzahl der im Response enthaltenen Elemente `nwst:Category` sein muss. Mit einem SXQT-Ausdruck ist das jedoch möglich. Abbildung 84 zeigt einen SXQT-Ausdruck, der das leistet.

```
declare namespace tra="http://ti5.tu-harburg.de/venzke/20021015/traces"
declare namespace env="http://www.w3.org/2002/06/soap-envelope"
declare namespace nwst="http://example.org/NewsService020917/Types"

every $m
  in fn:input()/tra:Trace/tra:Message
              [env:Envelope/env:Body/nwst:CategoriesResponse]
  satisfies
    $m/env:Envelope/env:Body/nwst:CategoriesResponse
      /nwst:NoCategories
    eq fn:count($m/env:Envelope/env:Body/nwst:CategoriesResponse
                 /nwst:CategoriesSequence/nwst:Category)
```

Abbildung 84: Anforderung an einzelne SOAP-Nachrichten als SXQT-Ausdruck

Der SXQT-Ausdruck in Abbildung 84 beginnt mit einem Query-Prolog, in dem die drei benötigten Namensräume an Präfixe gebunden werden. Erst danach folgt der eigentliche Ausdruck. Er besteht aus einem Allquantor, der für jeden beobachteten Response der Operation `Categories` im Trace prüft, ob die Anforderung für ihn erfüllt ist.

Hierzu wird beim Allquantor (`every`) hinter dem Schlüsselwort `in` eine Sequenz der Message-Elemente berechnet, die einen Response der Operation `Categories` enthalten. Wie in Kapitel 7.5.3 festgestellt, liefert `fn:input()/tra:Trace/tra:Message` die Sequenz aller Message-Elemente im Trace. In Abbildung 84 wird diese mit einem Prädikat (in eckigen Klammern) auf die Message-Elemente mit einem Response der Operation `Categories` beschränkt. Das Prädikat prüft hierzu, ob der Bodyeintrag der SOAP-Nachricht den Namen `nwst:CategoriesResponse` hat.

Alle Message-Elemente, auf die das zutrifft, werden durch den Allquantor jeweils an die Variable `$m` gebunden. Auf die Variable wird im Ausdruck nach dem Schlüsselwort `satisfies` verwiesen, in dem geprüft wird, ob die durch das Message-Element repräsentierte Beobachtung die obige Anforderung erfüllt. Geprüft werden muss also, ob im Bodyeintrag der SOAP-Nachricht die im Element `nwst:NoCategories` enthaltene Zahl gleich der Anzahl der Elemente `nwst:Category` ist.

Dieser Vergleich wird mit der Vergleichsoperation `eq` durchgeführt, die zwei Werte auf Gleichheit prüft. Als ersten Operanden hat diese Operation einen Pfadausdruck, mit der das Element `nwst:NoCategories` aus der SOAP-Nachricht extrahiert wird. Die Semantik der Vergleichsoperation `eq` führt dazu, dass aus diesem Element sein typisierter Wert extrahiert wird. Das ist die Zahl, die im Element enthalten ist, also die in der SOAP-Nachricht angegebene Anzahl der Rubriken.

Mit dem zweiten Operanden des Vergleichsoperation `eq` wird bestimmt, wie viele Rubriken in der SOAP-Nachricht enthalten sind. Hierzu wird ein Pfadausdruck verwendet, um eine Sequenz mit allen Elementen `nwst:Category` aus der SOAP-Nachricht zu ermitteln. Aus dieser Sequenz wird mit der Funktion `fn:count` die Anzahl der in ihr enthaltenen Elemente bestimmt, also die Anzahl der in der SOAP-Nachricht enthaltenen Rubriken.

Sind die angegebene und enthaltene Anzahl von Rubriken für alle Responses der Operation `Categories` gleich, liefert der SXQT-Ausdruck den logischen Wert `true`. Damit ist die Anforderung an den beobachteten Trace erfüllt. Sind die Werte nur für einen Response dieser Operation nicht gleich, ist das nicht der Fall. Die Anforderung ist für den beobachteten Trace nicht erfüllt.

Die vorgestellte Anforderung ließe sich in XQuery auch kürzer formulieren. Das zeigt Abbildung 85. Der Ausdruck aus Abbildung 84 wurde auf zwei Arten verkürzt. Die Variable des Allquantors wurde an ein anderes als das Message-Element gebunden und statt qualifizierten Namen von Elementen wurde der Wildcard `*` verwendet.

```
declare namespace tra="http://ti5.tu-harburg.de/venzke/20021015/traces"
declare namespace env="http://www.w3.org/2002/06/soap-envelope"
declare namespace nwst="http://example.org/NewsService020917/Types"

every $frag
  in fn:input()/*/*/*/env:Body/nwst:CategoriesResponse
  satisfies $frag/nwst:NoCategories
            eq fn:count($frag/nwst:CategoriesSequence/*)
```

Abbildung 85: Anforderung an einzelne SOAP-Nachrichten als SXQT-Ausdruck, verkürzt notiert

Statt an das Message-Element wurde die Variable des Allquantors in Abbildung 85 direkt an den Bodyeintrag der enthaltenen SOAP-Nachricht gebunden. Da die Variable nachfolgend nur verwendet wird, um auf Kindelemente des Bodyeintrages zuzugreifen, verkürzen sich die verwendeten Pfadausdrücke. Würde im SXQT-Ausdruck die Variable auch verwendet, um auf Headereinträge oder die beiden Attribute des Message-Elementes zuzugreifen, würde sich diese Verkürzung nicht ergeben. Die SXQT-

Ausdrücke dieser Arbeit sollen möglichst nach einem einheitlichen Muster konstruiert werden. Daher soll durch den Allquantor die Variable stets an die Message-Elemente gebunden werden, wie in Abbildung 84 gezeigt.

Die qualifizierten Namen der Elemente können in Pfadausdrücken nur dann durch den in Abbildung 85 verwendeten Wildcard `*` ersetzt werden, wenn alle durch ihn bezeichneten Elemente ohnehin den gleichen qualifizierten Namen haben. Nur dann ändert sich durch seine Verwendung der Pfadausdruck nicht. Das ist z. B. bei den Kindelementen von `tra:Trace` der Fall, weil `tra:Trace` nur Message-Elemente als Kindelemente hat, die stets den Namen `tra:Message` haben. Pfadausdrücke auf diese Art zu verkürzen, ist häufig möglich. Es erschwert aber menschlichen Lesern, die Ausdrücke zu verstehen. Für sie sind die qualifizierten Namen eine Möglichkeit den Überblick zu wahren, welche Elemente erwartet werden. Diese Möglichkeit entfällt durch die Verwendung des Wildcards `*`.

In Kapitel 7.3.4 wurde festgestellt, dass für logische Ausdrücke von Spezifikationen stets bekannt sein muss, in welchen Sichten sie verwendet werden können. Das muss auch für den eben gegebenen SXQT-Ausdruck gelten. Er ist in jeder Sicht gültig. Der Grund ist, dass er sich nur auf einzelne SOAP-Nachrichten bezieht. Solche Ausdrücke sind stets von der Sicht unabhängig, weil es für die Beurteilung keine Rolle spielt, welche anderen SOAP-Nachrichten beobachtet wurden.

7.5.5 Beispiel für Anforderung an einzelne Request-/Response-Paare

Auch für die Anforderung im nun folgenden, zweiten Beispiel spielt die Sicht keine Rolle. Sie bezieht sich auf einzelne Request-/Response-Paare. Solche Anforderungen sind von der Sicht unabhängig, weil in jeder Sicht sowohl Request als auch Response beobachtet werden oder beide nicht.

Für den SXQT-Ausdruck werde zunächst ignoriert, dass Responses beobachtet werden können, für die der Request vor Beginn der Beobachtung ausgetauscht wurde und daher im Trace fehlt. Wie im Kapitel 7.3.5 diskutiert, müsste das eigentlich berücksichtigt werden, was jedoch zur Vereinfachung zunächst nicht geschehen soll. Später in dieser Arbeit wird gezeigt, wie mit dem Problem umgegangen wird.

Das Beispiel bezieht sich auf eine Anforderung aus Kapitel 6.1.2. Sie beschreibt eine Beziehung zwischen einem Request und dem zugehörigen Response eines Aufrufs der Operation `Get` der Webkomponenten für eine Internetzeitung. Abbildung 53 auf Seite 100 und Abbildung 54 auf Seite 100 zeigen je ein Beispiel für einen solchen Request bzw. Response. Die Anforderung legt fest, dass es im Request ebenso viele Elemente `nwst:MessageID` geben muss, wie im Response Elemente `nwst:Message`.

```
declare namespace tra="http://ti5.tu-harburg.de/venzke/20021015/traces"
declare namespace env="http://www.w3.org/2002/06/soap-envelope"
declare namespace nwst="http://example.org/NewsService020917/Types"

every $m
  in fn:input()/tra:Trace/tra:Message
         [env:Envelope/env:Body/nwst:GetResponse]
  satisfies
    fn:count($m/env:Envelope/env:Body
               /nwst:GetResponse/nwst:Message)
    eq fn:count(fn:input()/tra:Trace/tra:Message
                  [@to="Service" and @operation=$m/@operation]
                  /env:Envelope/env:Body/nwst:Get
                  /nwst:MessageIDsSequence/nwst:MessageID)
```

Abbildung 86: Anforderung an einzelne Request-/Response-Paare als SXQT-Ausdruck

Das wird durch den SXQT-Ausdruck in Abbildung 86 ausgedrückt. Auffallend ist, dass seine Grundstruktur dem SXQT-Ausdruck aus Abbildung 84 gleicht. Auch hier werden zunächst benötigte Namensräume an Präfixe gebunden. Der nachfolgende, eigentliche Ausdruck besteht aus einem Allquantor, der für eine Sequenz von Message-Elementen die Anforderung prüft, wozu jedes Message-Element an die Variable `$m` gebunden wird. Hier sind es jedoch Message-Elemente, die Beobachtungen von Responses der Operation `Get` repräsentieren.

Nach dem Schlüsselwort `satisfies` werden wieder zwei Werte mit der Vergleichsoperation `eq` verglichen. Der erste Operand ermittelt die Anzahl der Elemente `nwst:Message` im Response. Verwendet wird wie in Abbildung 84 ein Pfadausdruck und die Funktion `fn:count`. Im Pfadausdruck wird direkt mit der Variable `$m` auf das Message-Element des Responses verwiesen.

Die Anzahl der Elemente `nwst:MessageID` im Request zu ermitteln, ist aufwendiger, weil sich der Request im Trace in einem anderen Message-Element befindet. Auf dieses verweist keine Variable. Daher wird ein Pfadausdruck benötigt, der erst das Message-Element sucht, bevor die Sequenz der Elemente `nwst:MessageID` bestimmt werden kann.

Dazu verweist der zweite Operand der Vergleichsoperation `eq` erneut mit dem Ausdruck `fn:input()/tra:Trace/tra:Message` auf den Trace. Er beschränkt die Message-Elemente im Trace auf dasjenige, das den Request zu dem Response enthält, dessen Message-Element an die Variable `$m` gebunden ist. Hierzu wird das Prädikat in eckigen Klammern verwendet. Das Message-Element des Requests muss im Attribut `to` den Wert `Service` haben, weil ein Request stets vom Webclient zum Webservice gesendet wird. Außerdem muss es im Attribut `operation` den gleichen Wert haben, wie das Message-Element des Responses, weil beide SOAP-Nachrichten zum gleichen Request-/Response-Paar gehören.

Auf diese Art wird eine Sequenz erzeugt, die als einziges Element das Message-Element mit dem Request enthält. Durch Fortsetzung des Pfadausdruckes wird daraus eine Sequenz mit allen Elementen `nwst:MessageID` im Request erzeugt. Wieder wird die Funktion `fn:count` verwendet, um die Menge der Elemente in der Sequenz zu ermitteln, also die Anzahl der Elemente `nwst:MessageID` im Request.

Im SXQT-Ausdruck in Abbildung 86 wurde von den Responses im Trace ausgegangen und für jeden der zugehörige Response gesucht. Dann wurden Request und Response miteinander in Beziehung gesetzt. Der SXQT-Ausdruck könnte auch so formuliert werden, dass von den Requests ausgegangen wird und zu diesen jeweils der zugehörige Response im Trace gesucht wird. Der SXQT-Ausdruck in Abbildung 87 ist auf diese Art konstruiert.

Dazu wird nach dem Schlüsselwort `in` des Allquantors die Sequenz der Message-Elemente auf solche mit Bodyeinträgen `nwst:Get` eingeschränkt, also auf die Message-Elemente mit Requests der Operation `Get`. Im zweiten Operanden der Vergleichsoperation `eq` kann so direkt die Sequenz der Elemente `nwst:MessageID` eines Requests gebildet werden, weil die Variable `$m` auf das Message-Element des Requests verweist. Im ersten Operanden muss dagegen zunächst der zugehörige Response gesucht werden, bevor die Sequenz von dessen Elementen `nwst:Message` gebildet werden kann.

```
declare namespace tra="http://ti5.tu-harburg.de/venzke/20021015/traces"
declare namespace env="http://www.w3.org/2002/06/soap-envelope"
declare namespace nwst="http://example.org/NewsService020917/Types"

every $m
  in fn:input()/tra:Trace/tra:Message
        [env:Envelope/env:Body/nwst:Get]
  satisfies
    fn:count(fn:input()/tra:Trace/tra:Message
             [@to="Client" and operation=$m/@operation]
             /env:Envelope/env:Body
             /nwst:GetResponse/nwst:Message)
    eq fn:count($m/env:Envelope/env:Body/nwst:Get
                  /nwst:MessageIDsSequence/nwst:MessageID)
```

Abbildung 87: Anforderung an einzelne Request-/Response-Paare als SXQT-Ausdruck, anders formuliert

Auf diese Art scheint die Anforderung aus Kapitel 6.1.2 durch den SXQT-Ausdruck in Abbildung 87 korrekt formuliert worden zu sein. Es wurde vereinfachend jedoch nicht berücksichtigt, dass der SXQT-Ausdruck auch für solche Traces `true` liefert, bei denen bisher nur der Request beobachtet wurde, aber noch nicht der Response. Im SXQT-Ausdruck in Abbildung 87 würde dann der Response nicht gefunden. Er würde den logischen Wert `false` liefern.

Zwar könnte der SXQT-Ausdruck das Fehlen des Responses im Trace explizit berücksichtigen, so dass er auch den Wert `true` liefert, wenn der Response fehlt. Einfacher ist es jedoch, so wie im SXQT-Ausdruck in Abbildung 86 vom Response auszugehen. Ist dieser beobachtet, muss auch vorher der Request beobachtet worden sein[45].

Werden mehr als zwei Beobachtungen von SOAP-Nachrichten miteinander in Beziehung gesetzt, lässt sich diese Regel verallgemeinern. Dann könnte prinzipiell von jeder dieser SOAP-Nachrichten ausgegangen werden. Um zu vermeiden, dass dann die Beobachtung einer der in Beziehung gesetzten SOAP-Nachrichten im Trace noch fehlt, sollte von der letzten beobachteten SOAP-Nachricht ausgegangen werden, wenn eines keine guten Gründe gibt, die dagegen sprechen. Zu dieser SOAP-Nachricht sollten dann die zuvor beobachteten im Trace ermittelt und miteinander in Beziehung gesetzt werden.

Durch diese Regel wird erleichtert, dass SXQT-Ausdrücke korrekt formuliert werden. Außerdem führt sie zu einheitlicheren SXQT-Ausdrücken, weil sie eine Anleitung gibt, wie diese konstruiert werden sollen. Einheitliche SXQT-Ausdrücke erleichtern es menschlichen Lesern auch, sie zu verstehen. Bei der maschinellen Auswertung können solche Regeln außerdem für Optimierungen genutzt werden.

7.6 Operationen für SXQT-Ausdrücke

Die bisherige Darstellung von SXQT reicht prinzipiell, um SXQT-Ausdrücke zu formulieren. In diesem Kapitel 7.6 sollen Operationen vorgestellt werden, die das Formulieren von SXQT-Ausdrücken vereinfachen, sie prägnanter und ihre Konstruktion weniger Fehleranfällig machen.

---

[45] Wie bereits am Anfang dieses Kapitels 7.5.5 erwähnt, wird vorausgesetzt, dass nicht vor Beginn der Beobachtung ein Request ausgetauscht wurde, für den später der Response beobachtet wird. Ansonsten stellt sich prinzipiell das gleiche Problem wie beim noch fehlenden Response. Für das Problem vor Beginn der Beobachtung ausgetauschter Requests wird aber in Kapitel 7.6.5 eine generelle Lösung gegeben.

7.6.1 Notwendigkeit von Operationen

Dass Operationen notwendig sind, kann schon aus den beiden in den Kapiteln 7.5.4 und 7.5.5 vorgestellten Beispielen geschlossen werden. Solche SXQT-Ausdrücke zu erstellen oder zu verstehen, erfordert Kenntnisse über den genauen Aufbau des Traces, also der Elemente `tra:Trace` und `tra:Message` sowie der SOAP-Nachrichten. Außerdem gibt es Teilausdrücke, die sich häufig wiederholen. Beides erschwert das Erstellen oder Verstehen von SXQT-Ausdrücken.

Beides kann außerdem zu Fehlern in den SXQT-Ausdrücken führen, die z. T. schwierig zu finden sind. So kann z. B. ein Irrtum über den Aufbau oder ein „kleiner Tippfehler" dazu führen, dass SXQT-Ausdrücke unabhängig von beobachteten Trace stets den Wert `true` oder `false` liefern. Würde z. B. in einem der SXQT-Ausdrücke in Abbildung 84 oder Abbildung 86 der Trace statt mit `fn:input()/tra:Trace/tra:Message` mit `fn:input()/tra:Trace/tra:Mesage` bezeichnet, also im Wort `Message` ein Buchstabe vergessen, würde dieser Fehler nicht vom XQuery-Prozessor erkannt. Der Fehler führt jedoch dazu, dass der Ausdruck stets den Wert `true` liefert. Es würde nämlich nie ein Element tra:Mesage gefunden, die Sequenz von Elementen, die der Allquantor testet, wäre also immer leer. Der Allquantor in XQuery ist nach [Boag 02] so definiert, dass er dann `true` liefert.

Solche Probleme könnten mit Operationen gelöst werden, mit denen SXQT-Ausdrücke prägnanter formuliert werden können. Für häufig auftretende Teilausdrücke müssten Operationen definiert werden, die ihnen in gewisser Weise einen Namen geben. Ein Tippfehler in einem Operationsnamen würde dann zu einer unbekannten Operation führen, also zu einem statischen Fehler, der vom XQuery-Prozessor erkannt wird. Gleichzeitig würde der Name der Operation auch für einen menschlichen Leser des Ausdrucks verständlicher sein als der Teilausdruck selbst.

Drei Beispiele für häufig auftretende Teilausdrücke können bereits aus den Beispielen der Kapitel 7.5.4 und 7.5.5 abgeleitet werden. So wird in den SXQT-Ausdrücken in Abbildung 84, Abbildung 86 und Abbildung 87 jeweils mit dem Teilausdruck `fn:input()/tra:Trace/tra:Message` auf den Trace zu verwiesen. Ebenso wird in jedem der Ausdrücke der Trace auf eine Sequenz abgebildet, die nur bestimmte Message-Elemente enthält. Schließlich wurde es in den SXQT-Ausdrücken in Abbildung 86 und Abbildung 87 notwendig, zu einem Response den zugehörigen Request zu ermitteln (bzw. umgekehrt). Für solche Teilausdrücke sollten Operationen definiert werden.

Ähnlich geht auch Hoare in [Hoare 85] vor. Er definiert Operationen für Traces. Bei den meisten handelt es sich um Operationen für Sequenzen, die auch unabhängig von Traces verwendbar sind. Er verwendet sie in Spezifikationen, um diese prägnanter formulieren zu können.

Auch XQuery definiert eine Reihe von Operationen. Sie erlauben bereits den Umgang mit XML-Dokumenten und -Fragmenten und damit auch mit Traces. Das wurde in den SXQT-Ausdrücken in den Kapiteln 7.5.4 und 7.5.5 demonstriert. Realisiert werden Operationen in XQuery auf zwei Arten. Für einige gibt es eine spezielle Syntax. Hierzu gehören z. B. Pfadausdrücke. Andere sind als Funktionen realisiert, z. B. die in den Kapiteln 7.5.4 und 7.5.5 verwendete Funktion `fn:count`.

Neben Standardfunktionen erlaubt XQuery auch die Definition von benutzerdefinierten Funktionen im Query-Prolog oder in Bibliotheken. Das ermöglicht Entwicklern von Spezifikationen, häufig auftretende Teilausdrücke einmalig als Funktionen zu definie-

ren. Diese können in mehreren SXQT-Ausdrücken einer Spezifikation oder auch in unterschiedlichen Spezifikationen wiederverwendet werden.

Nachfolgend soll untersucht werden, welche Operationen generell für die Spezifikation von Webservices hilfreich sind. Dazu werden vor allem die Operationen für Traces von Hoare aus [Hoare 85] betrachtet. Für sie wird untersucht, wie sie am zweckmäßigsten auf XQuery abgebildet werden können. In vielen Fällen erfolgt das durch Angabe einer benutzerdefinierten Funktion. Ihre Namen befinden sich im Namensraum mit der URI `http://ti5.tu-harburg.de/venzke/20021015/operations`, für den das Präfix `opr` verwendet wird. Auch die von XQuery als Standardfunktionen oder in Form einer speziellen Syntax zur Verfügung gestellten Operationen werden betrachtet und weitere für die Spezifikation zweckmäßige zugefügt.

Die Untersuchung der Operationen soll in den nachfolgenden Kapiteln 7.6.2 bis 7.6.5 danach sortiert erfolgen, auf welche Teile des Traces sie sich beziehen und aus welcher Quelle sie stammen. Kapitel 7.6.2 behandelt Operationen, die sich unabhängig vom Trace auf Message-Elemente beziehen. Die Kapitel 7.6.3 bis 7.6.5 beziehen sich dagegen auf Operationen für Traces. Die von Hoare in [Hoare 85] definierten werden im Kapitel 7.6.3 vorgestellt. Viele seiner Operationen sind universell für Sequenzen einsetzbar. Solche Operationen enthält auch XQuery, von denen die für Traces besonders sinnvollen in Kapitel 7.6.4 behandelt werden. Weitere Operationen für Traces werden in Kapitel 7.6.5 zugefügt.

7.6.2 Operationen für Message-Elemente

Operationen auf Message-Elementen dienen dazu, Details des Aufbaues von Message-Elementen und seiner Kindelemente zu verbergen. Mit ihnen können SXQT-Ausdrücke formuliert oder verstanden werden, ohne dass Detailwissen über die Elemente `tra:Message`, `env:Envelope`, `env:Header` oder `env:Body` erforderlich ist. Es lassen sich zwei Arten von Operationen unterscheiden. Fünf Operationen dienen dazu, Teile aus einem Message-Element zu extrahieren. Eine weitere erlaubt es zu testen, ob Message-Elemente zur gleichen Ereignisklasse gehören.

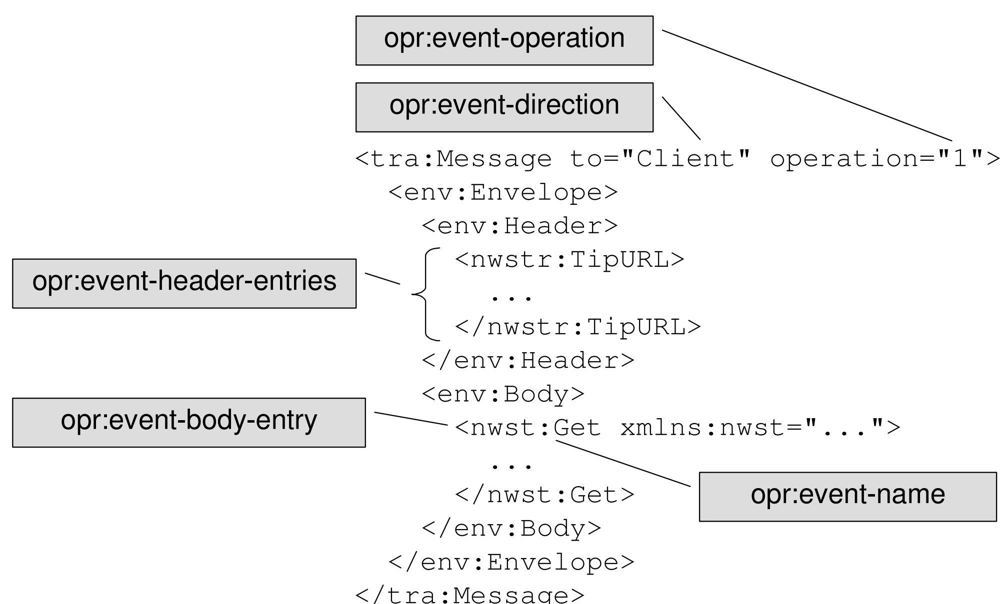

Abbildung 88: Operationen zum Extrahieren von Teilen aus Message-Element

Abbildung 88 veranschaulicht die fünf Operationen, die Teile aus einem Message-Element extrahieren. Hierzu ist ein Beispiel für eine SOAP-Nachricht gezeigt, der auch einen Headereintrag enthält[46]. Für jede Operation wird in Abbildung 88 gezeigt, welchen Teil der SOAP-Nachricht sie liefert. Dazu hat jede Operation als einzigen Operanden das Message-Element, das die SOAP-Nachricht enthält.

Die Operationen `opr:event-direction`[47] und `opr:event-operation` liefern die Werte der Attribute `to` bzw. `operation` des Message-Elementes. Ohne Wissen über den Aufbau von Message-Elementen lässt sich mit ihnen für ein Message-Element ermitteln, in welche Richtung die beobachtete SOAP-Nachricht ausgetauscht wurde und zu welchem Request-/Response-Paar sie gehört (vergleiche Kapitel 7.2.2).

Welchen Inhalt die SOAP-Nachricht enthält, kann mit den Operationen `opr:event-body-entry` und `opr:event-header-entries` ermittelt werden. Die Operation `opr:event-body-entry` liefert den Bodyeintrag, also das einzige Kindelement des Elementes `env:Body`. Dagegen liefert `opr:event-header-entries` eine Sequenz mit allen Headereinträgen der SOAP-Nachricht. Diese Operationen können verwendet werden, ohne zu wissen, dass das Element `env:Envelope` das Kindelement des Message-Elementes ist und ohne Kenntnis über die genaue Struktur einer SOAP-Nachricht.

Mit der Operation `opr:event-name` kann der Ereignisname des Ereignisses ermittelt werden, dass das Message-Element repräsentiert. In Kapitel 7.2.1 wurde festgelegt, dass der qualifizierte Name des Bodyeintages der Ereignisname ist. Ihn ermittelt die Operation `opr:event-name`.

Realisiert werden können die fünf Operationen als benutzerdefinierte Funktionen. Diese enthalten Pfadausdrücke sowie Standardfunktionen von XQuery. Abbildung 89 zeigt beispielhaft die Realisierung der beiden Funktionen `opr:event-body-entry` und `opr:event-name`. Das Message-Element wird jeweils als Parameter übergeben, daher hat der Parameter `$m` den Typ `element tra:Message`. Aus ihm wird der Bodyeintrag bzw. der Ereignisname extrahiert.

```
define function opr:event-body-entry($m as element tra:Message)
  as element
{
   $m/env:Envelope/env:Body/*[1]
}

define function opr:event-name($m as element tra:Message)
  as xsd:QName
{
  opr:node-name($m/env:Envelope/env:Body/*[1])
}
```

Abbildung 89: Realisierungen der Funktionen `opr:event-body-entry` und `opr:event-name`

Für die Funktion `opr:event-body-entry` ist nicht bekannt, welchen Typ der Bodyeintrag hat. Daher kann als Typ für den Rückgabewert hinter dem Schlüsselwort `as` nur der Typ `element` angegeben werden, der Elemente beliebiger Typen erlaubt. Der Rückgabewert wird durch den Pfadausdruck `$m/env:Envelope/env:Body/*[1]` bestimmt. Darin bezeichnet `$m` den Parameter der Funktion, also das Message-Element.

---

[46] Um die Darstellung zu verkürzen, wurden ausgelassene Teile durch drei Punkte („...“) ersetzt.

[47] Das Präfix `opr` stehe für den Namensraum `http://ti5.tu-harburg.de/venzke/20021015/operations`.

Der von der Funktion `opr:event-name` gelieferte Ereignisname ist ein qualifizierter Name, der Typ des Rückgabewertes daher `xsd:QName`. Um den Ereignisnamen zu ermitteln, wird zunächst der Pfadausdruck aus der Funktion `opr:event-body-entry` verwendet, der den Bodyeintrag liefert. Auf diesen wird die Standardfunktion `fn:node-name` angewendet, die zu einem Knoten seinen qualifizierten Namen liefert (siehe [Malhotra 02]).

Neben der Möglichkeit Teile aus Message-Elementen zu extrahiert, ist auch wichtig prüfen zu können, ob zwei Message-Elemente zur gleichen Ereignisklasse gehören. Wie in Hoares Modell können hierzu die zum Ereignis gehörenden Ereignisnamen verglichen werden, die sich mit der Funktion `opr:event-name` ermitteln lassen. Den so realisierten Vergleich leistet die Funktion `opr:same-event-class`.

```
define function opr:same-event-class($m1 as element tra:Message,
                                     $m2 as element tra:Message)
  as xsd:boolean
{
  opr:event-name($m1) eq opr:event-name($m2)
}
```

Abbildung 90: Realisierung der Funktion `opr:same-event-class`

Die Realisierung der Funktion `opr:same-event-class` wird in Abbildung 90 gezeigt. Da es sich bei ihr um ein Prädikat handelt, liefert sie einen Wert vom Typ `xsd:boolean`. Sie hat zwei Parameter für die zu vergleichenden Message-Elemente. Die für sie ermittelten Ereignisnamen werden mit der Vergleichsoperation `eq` verglichen.

Die Vergleichsoperation `eq` vergleicht Ereignisnamen korrekt, obwohl es qualifizierte Namen sind. Qualifizierte Namen werden zwar mit Hilfe von Präfixen notiert. Für ihre Gleichheit ist das Präfix jedoch nicht relevant. Zwei qualifizierte Namen sind gleich, wenn ihre Präfixe auf den gleichen Namensraum verweisen und sie den gleichen lokalen Namen haben. Daher kann die Vergleichsoperation `eq` zum Vergleich von qualifizierten Namen verwendet werden.

7.6.3 Operationen für Traces von Hoare

Die in Kapitel 7.6.2 vorgestellten Operationen beziehen sich auf einzelne Message-Elemente sowie deren Kindelemente. Vom Trace als ganzes sind sie unabhängig. Die nachfolgend vorgestellten Operationen beziehen sich dagegen gerade auf Traces als ganzes. Zunächst sollen solche untersucht werden, die von Hoare in [Hoare 85] für Traces oder allgemein für Sequenzen definiert wurden.

7.6.3.1 In XQuery vorhanden

Drei der Operationen von Hoare lassen sich direkt auf Operationen für Sequenzen abbilden, die XQuery zur Verfügung stellt. Es sind Operationen `Subscription`, `Concatenation` und `Length`. Bei der Operation `Subscription` wird sogar die gleiche Syntax verwendet.

Die Operation `Subscription` verwendet Hoare, um ein Element einer Sequenz zu bezeichnen. Er notiert diese Operation mit eckigen Klammern hinter der Sequenz, die die Position des bezeichnenden Elementes umschließt. So bezeichnet z. B. der Ausdruck `<12,18,11,32>[2]` das dritte Element, also die Zahl 11. Dabei ist zu beachten, dass

Hoare für das erste Element den Index 0 verwendet. XQuery verwendet für diese Operation die gleiche Syntax, aber für das erste Element den Index 1. So bezeichnet der XQuery-Ausdruck (12,18,11,32)[3] ebenfalls das Element 11.

Eine ähnliche Syntax wird auch für die Operation Concatenation verwendet. Sie hat als Operanden zwei Sequenzen und liefert eine Sequenz, die zunächst die Elemente des ersten Operanden und danach die des zweiten enthält. Hoare notiert die Operation Concatenation mit dem Symbol ^. So liefert der Ausdruck (<12,18> ^ <11,32>) die Sequenz <12,18,11,32>. Wie bereits in Kapitel 7.5.2 beschrieben, verwendet XQuery für diese Operation als Symbol das Komma. Daher würde der XQuery-Ausdruck ((12,18), (11,32)) die Sequenz (12,18,11,32) liefern.

Direkt abbilden lässt sich auch Hoares Operation Length, mit der die Anzahl der Elemente einer Sequenz bestimmt werden kann. Hoare verwendet für sie das Symbol #. So liefert z. B. #<12,18,11,32> den Wert 4, weil die Sequenz 4 Elemente hat. Die gleiche Operation gibt es in XQuery als Standardfunktion fn:count. Sie wurde bereits in den SXQT-Ausdrücken in den Kapiteln 7.5.4 und 7.5.5 verwendet. So liefert der Ausdruck fn:count((12,18,11,32)) ebenfalls den Wert 4.

7.6.3.2 Realisierbar als benutzerdefinierte Funktion universell für Sequenzen

Nicht alle Operationen von Hoare lassen sich direkt auf eine spezielle Syntax oder eine Standardfunktionen von XQuery abbilden. Eine Reihe von ihnen lässt sich jedoch als benutzerdefinierte Funktion realisieren. Für beliebige Sequenzen kann das für die Operationen Head, Tail und Reverse erfolgen.

Die Operation Head dient dazu, das erste Element einer Sequenz zu ermitteln. Hoare verwendet für sie als Symbol eine tiefgestellte 0. So liefert z. B. <12,18,11,32>$_0$ den Wert 12. Um die Operation Head in XQuery zu realisieren, kann ausgenutzt werden, dass auch mit der Operation Subscription das erste Element einer Sequenz ermittelt werden kann, da für jede Sequenz s gilt: s_0 = s[0]. Die Realisierung dieser Operation als Funktion ist in Abbildung 91 gezeigt. Als Parameter wird eine Sequenz einer beliebigen Zahl beliebiger atomarer Werte oder Knoten übergeben. Angegeben wird daher der Typ item*. Zurückgegeben wird ein atomarer Wert oder Knoten. Der Typ ist also item.

Die Operation Tail liefert die gleiche Sequenz, die ihr im einzigen Operanden übergeben wird, jedoch ohne das erste Element. Hoare notiert sie mit einem Hochkomma hinter dem Operanden. So liefert z. B. <12,18,11,32>' die Sequenz <18,11,32>. Auch diese Operation gibt es in XQuery nicht. Abbildung 91 zeigt, wie sie als benutzerdefinierte Funktion realisiert werden kann. Dazu wird die Standardfunktion fn:remove verwendet, die in [Malhotra 02] beschrieben wird. Sie hat zwei Parameter, eine Sequenz und eine Zahl. Sie liefert die Sequenz, die als erster Parameter übergeben wird, jedoch ohne das Element, an der Position, die als zweiter Parameter angegeben wird. Daher liefert der in Abbildung 91 verwendete Ausdruck fn:remove($seq, 1) die Sequenz im Parameter $seq, jedoch ohne das erste Element.

Die dritte in Abbildung 91 gezeigte Funktion realisiert Hoares Operation Reverse. Sie liefert eine Sequenz mit den Elementen in ihrer umgekehrten Reihenfolge. Hoare notiert sie mit einer waagerechten Linie über dem Operanden. Z. B. liefert $\overline{<12, 18, 11, 32>}$ die Sequenz <32, 11, 18, 12>. In XQuery kann die Operation als Funktion auf unterschiedliche Arten realisiert werden, z. B. auch rekursiv. In Abbildung 91 wurde sie mit Hilfe eines FLWOR-Ausdrucks realisiert. (Siehe Kapitel 7.4.8.)

```
define function opr:head($seq as item*)
  as item
{
  $seq[1]
}

define function opr:tail($seq as item*)
  as item*
{
  fn:remove($seq, 1)
}

define function opr:reverse($seq as item*)
  as item*
{
  for $i in (fn:count($seq) to 1)
  returns $seq[$i]
}
```

Abbildung 91: Realisierungen der Funktionen `opr:head`, `opr:tail` und `opr:reverse`

7.6.3.3 Realisierbar als benutzerdefinierte Funktion für Sequenzen von Knoten

Einige der Operationen von Hoare können zwar in XQuery als benutzerdefinierte Funktionen realisiert werden. Mit Sicht auf diese Arbeit ist es für sie jedoch zweckmäßig, als Operanden nur Sequenzen von Knoten zuzulassen, obwohl Hoare sie für beliebige Sequenzen definiert. Erst so bekommen sie für die Verwendung mit den Traces dieser Arbeit eine sinnvolle Bedeutung.

Es handelt sich um die Prädikate `Präfix`, `Subsequence` und `Interleaves`. Sie haben zwei bzw. drei Sequenzen als Operanden. Als Prädikate liefern stets einen der logischen Werte `true` oder `false`.

Die Operation `Präfix` prüft, ob der erste Operand `s` der Anfang des zweiten Operanden `t` ist, so dass es also eine Sequenz `u` gibt, für die gilt `s^u = t`. Hoare notiert diese Operation mit dem Symbol ≤, z. B. liefert der Ausdruck `<12,18> ≤ <12,18,11,32>` den logischen Wert `true`.

Dem gegenüber prüft die Operation `Subsequence`, ob der erste Operand `s` irgendwo im zweiten Operanden `t` enthalten ist, so dass es also zwei Sequenzen `u`, `v` gibt, für die gilt `u^s^v = t`. Diese Operation notiert Hoare mit dem Schlüsselwort `in`. Z. B. liefert `<18,11> in <12,18,11,32>` den logischen Wert `true`.

Die Operation `Interleaves` notiert Hoare mit dem Schlüsselwort `interleaves` in der Form `(s interleaves(t, u))`, wobei die drei Operanden `s`, `t` und `u` drei Sequenzen sind. Die Operation testet, ob `s` gerade die Elemente sowohl aus `t` als auch `u` enthält, so dass die Reihenfolge der Elemente aus `t` erhalten bleibt, ebenso wie die Reihenfolge aus `u`. Wie die Elemente von `t` und `u` „gemischt" worden sind, spielt jedoch keine Rolle. Z. B. liefert der folgende Ausdruck den logischen Wert `true`:
`<4,7,12,18,9,11,32> interleaves(<12,18,11,32>, <4,7,9>)`.

Die Realisierungen der drei Prädikate in XQuery sollen vor allem für Traces eine sinnvolle Semantik haben. Traces sind in dieser Arbeit Sequenzen von Message-Elementen, wobei jedes Message-Element ein beobachtetes Ereignis repräsentiert. Die drei Prädikate müssen die Elemente der Sequenzen, also die Message-Elemente, auf Gleichheit prüfen können. Das wirft die Frage auf, wann zwei Message-Elemente als gleich angesehen werden sollen.

Für diese Arbeit sei festgelegt, dass zwei Message-Elemente als gleich angesehen werden, wenn sie dasselbe Ereignis repräsentieren. Da ein Ereignis nie im Trace durch mehrere Message-Elemente repräsentiert wird, werden Message-Elemente nur dann als gleich angesehen, wenn ihre Elementknoten dieselbe Identität haben. Damit kann ihre Gleichheit in XQuery mit der Knotenvergleichsoperation `is` geprüft werden.

Diese Vorstellung von Gleichheit weicht von der von Hoare ab. Er notiert in Traces nur Ereignisnamen, die die Ereignisklasse bezeichnen. Aus den Ereignisnamen kann jedoch nicht mehr erschlossen werden, um welches der ununterscheidbaren Ereignisse der Ereignisklasse es sich gehandelt hat. Daher definiert er die Gleichheit der Elemente im Trace nur über die Gleichheit der Ereignisklassen. Diese Art der Gleichheit wird auch für die Prädikate Präfix, Subsequence und Interleaves verwendet.

Die Knotenvergleichsoperation `is` von XQuery kann nur für den Vergleich von Knoten verwendet werden, da nur diese eine Identität haben. Wird sie für die drei Prädikate `Präfix`, `Subsequence` und `Interleaves` verwendet, können diese nur für auf Sequenzen von Knoten definiert werden, und nicht mehr für Sequenzen von atomaren Werten. Abbildung 92 zeigt, wie die Prädikate in XQuery als benutzerdefinierte Funktionen definiert werden. Für die Parameter wird jeweils der Typ `node*` verwendet, der Sequenzen mit beliebigen Knoten erlaubt. Da es sich um Prädikate handelt, ist der Typ des Rückgabewertes `xsd:boolean`.

```
define function opr:prefix($s as node*,$t as node*)
  as xsd:boolean
{
  if fn:empty($s)
    then fn:true()
    else every $i
           in (1 to fn:count($s))
           satisfies $s[$i] is $t[$i]
}

define function opr:subsequence($s as node*, $t as node*)
  as xsd:boolean
{
  if fn:empty($t)
    then fn:empty($s)
    else opr:prefix($s, $t) or opr:subsequence($s, opr:tail($t))
}

define function opr:interleaves($s as node*,
                                $t as node*,
                                $u as node*)
  as xsd:boolean
{
  if fn:empty($s)
    then (fn:empty($t) and fn:empty($u))
    else (fn:exists($t)
          and (opr:head($s) is opr:head($t))
          and opr:interleaves(opr:tail($s), opr:tail($t), $u)
         )
         or (fn:exists($u)
             and (opr:head($s) is opr:head($u))
             and opr:interleaves(opr:tail($s), $t, opr:tail($u))
             )
}
```

Abbildung 92: Realisierungen der Funktionen `opr:prefix`, `opr:subsequence` und `opr:interleaves`

Die Realisierung der Operation `Präfix` verwendet einen bedingten Ausdruck sowie den Allquantor. Der bedingte Ausdruck stellt sicher, dass das Prädikat zutrifft, wenn der erste Parameter `$s` die leere Sequenz ist. Das ist notwendig, weil die leere Sequenz Präfix jeder anderen Sequenz ist. Der Test, ob `$s` die leere Sequenz ist, erfolgt mit der Standardfunktion `fn:empty`, die für leere Sequenzen den logischen Wert `true` liefert und für nicht leere den logischen Wert `false`.

Mit den Operationen `Präfix` sowie der Operation `Tail` aus Kapitel 7.6.3.2 lässt sich die Operation `Subsequence` leicht rekursiv realisieren. Die Operation testet, ob der erste Parameter `$s` im zweiten `$t` enthalten ist. Ist das der Fall, dann muss `$s` entweder ein Präfix von `$t` sein oder `$s` muss in `opr:tail($t)` enthalten sein, was rekursiv geprüft werden kann. Damit die Rekursion abbricht, muss getestet werden, ob die Sequenz in `$t` leer ist.

Auch die Operation `Interleaves` kann rekursiv als benutzerdefinierte Funktion realisiert werden. Abbildung 92 zeigt eine mögliche Realisierung. In ihr wird die Standardfunktion `fn:exists` verwendet, die für eine Sequenz prüft, ob diese mindestens ein Element enthält.

7.6.3.4 Realisierbar als benutzerdefinierte Funktion für Traces

Zwei Operationen von Hoare lassen sich in XQuery auch für Sequenzen beliebiger Knoten nicht sinnvoll definieren, sondern nur für Traces. Es sind die Operationen `Restriction` und `Count-Restricted`. In ihnen muss für die Message-Elemente im Trace der Ereignisname ermittelt werden. Das ist nur mit „Wissen" über den Aufbau von Message-Elementen möglich, aber nicht für beliebige Sequenzen, so dass die Operationen nur für Traces definiert werden können. Da in Hoares Modell Traces direkt Sequenzen von Ereignisnamen sind, mussten die Ereignisnamen bei ihm nicht erst ermittelt werden. Daher könnte er die Operationen für beliebige Sequenzen so definieren, dass sie trotzdem für Traces sinnvoll anwendbar sind.

Die Operation `Restriction` hat bei Hoare zwei Operanden, eine Sequenz `s` und eine Menge m. Die Operation liefert die Sequenz `s`, aus der aber alle Elemente entfernt wurden, die nicht in der Menge m enthalten sind. Die gelieferte Sequenz enthält also nur Elemente aus der Menge m. Hoare notiert die Operation `Restriction` mit dem Symbol ↑. Z. B. liefert `(<4,7,12,18,9,11,32>` ↑ `{11,12,13,18})` die Sequenz `<12,18,11>`.

Diese Operation kann im Modell von Hoare auf Traces angewendet werden, um sie auf Beobachtungen nur bestimmter Ereignisklassen zu beschränken. Hierzu wird als Menge m die Menge der Ereignisnamen angegeben, deren Ereignisklassen im gelieferten Trace noch enthalten sein sollen. So kann z. B. der Trace `<coin,choc,coin,choc,coin>` aus Kapitel 7.1.3 nur auf die Ereignisse der Ereignisklasse `coin` beschränkt werden. Hierzu wird der Ausdruck `(<coin,choc,coin,choc,coin>` ↑ `{coin})` verwendet, der die Sequenz `<coin,coin,coin>` liefert.

In diesem Sinne soll die Operation `Restriction` auch für Traces dieser Arbeit als Funktion `opr:restrict` definiert werden. Parameter sind ein Trace und eine Menge von Ereignisnamen, wie in Abbildung 93 gezeigt. Der Typ für einen Trace ist in XQuery `element tra:Message*`, eine Sequenz einer beliebigen Anzahl von Elementen `tra:Message`. Eine Sequenz gleichen Typs wird von der Operation auch geliefert.

Für die Menge von Ereignisnamen stellt sich das Problem, dass es in XQuery keine Typen für Mengen gibt. Daher werden für sie Sequenzen verwendet, obwohl diese geordnet sind und erlauben, dass ein Element mehrfach enthalten ist. XQuery stellt Funk-

tionen und Operationen zur Verfügung, um Sequenzen als Mengen verwenden zu können. So können mit den Funktionen `fn:distinct-nodes` bzw. `fn:distinct-values` aus einer Sequenz mehrfach enthaltene Elemente entfernt werden. Außerdem sind die Operationen `union`, `intersect` und `except` mit der Semantik der Mengenoperationen für die Vereinigungsmenge, Schnittmenge und Differenzmenge definiert. Zusammen erlaubt das, Sequenzen als Mengen zu verwenden. Als zweiter Parameter wird der Funktion `opr:restrict` daher die Menge der Ereignisnamen als eine Sequenz von qualifizierten Namen übergeben. Der Parameter hat daher den Typ `xsd:QName*`.

Zur Realisierung der Operation `Restriction` wird in Abbildung 93 ein FLWOR-Ausdruck und ein Existenzquantor verwendet. Der FLWOR-Ausdruck betrachtet jedes Message-Element im Trace. Er fasst nur die Message-Elemente zu einer Sequenz zusammen, die einen der gewünschten Ereignisnamen haben. Welche das sind, wird hinter dem Schlüsselwort `where` mit einem Existenzquantor geprüft. Er ermittelt zum Message-Element mit der in Kapitel 7.6.2 vorgestellten Funktion `opr:event-name` den Ereignisnamen. Diesen vergleicht er mit allen Ereignisnamen der als Parameter übergebenen Menge.

```
define function opr:restrict($trace as element tra:Message*,
                             $eventnames as xsd:QName*)
  as element tra:Message*
{
  for $m in $trace
    where some $en in $eventnames
          satisfies $en eq opr:event-name($m)
    return $m
}

define function opr:count-restricted
                       ($trace as element tra:Message*,
                        $eventnames as xsd:QName*)
  as xsd:nonNegativeInteger
{
  fn:count(opr:restrict($trace, $eventnames))
}
```

Abbildung 93: Realisierungen der Funktionen `opr:restrict` und `opr:count-restricted`

Neben der Restriction definiert Hoare die Operation `Count-Restricted`[48], mit der die Anzahl der Ereignisse in einem Trace bestimmt wird, die einen bestimmten Ereignisnamen haben. Die Operation hat den Trace sowie den gewünschten Ereignisnamen als Operanden. Hoare notiert sie mit dem Symbol ↓ und definiert sie mit Hilfe der Operation `Restriction` wie folgt: `s↓x = #(s↑{x})`, wobei `s` ein Trace und `x` ein Ereignisname ist.

Hier wird die Operation `Count-Restricted` ebenso wie die Restriction nur für Traces definiert. Anders als bei Hoare wird statt einem Ereignisnamen eine Menge von Ereignisnamen als Parameter angegeben. Die Operation liefert die Anzahl der Ereignisse im Trace, die einen Ereignisnamen der Menge haben. Da in XQuery ein einzelner Ereignisname von einer Sequenz mit diesem Ereignisnamen ununterscheidbar ist, kann statt der Menge trotzdem weiter ein einzelner Ereignisname als Parameter angegeben werden.

---

[48] Der Name `Count-Restricted` wurde vom Autor dieser Arbeit gewählt. Hoare gibt dieser Operation keinen Namen.

Wie die Operation `Count-Restricted` in XQuery als benutzerdefinierte Funktion realisiert werden kann, ist in Abbildung 93 gezeigt. Die Parameter der Funktion sind die gleichen, wie bei der Operation `Restriction`. Geliefert wird ein Wert vom Typ `xsd:nonNegativeInteger`, also eine ganze Zahl größer oder gleich null.

7.6.3.5 Spezifikationsspezifisch realisierbar

Operation `Selection` ließe sich zwar in XQuery realisieren, zweckmäßig ist das jedoch nur spezifikationsspezifisch. Hoare definiert sie nur für Sequenzen, deren Elemente Paare sind. Das erste Element des Paares ist ein Symbol, das als Index für das zweite Element dient. Die Operation `Selection` erhält die Sequenz `s` von Paaren und ein Symbol `x` als Operanden. Sie sucht in der Sequenz `s` alle die Paare bei denen das Symbol `x` das erste Element ist. Von diesen Paaren liefert sie die zweiten Elemente als Sequenz.

Paare notiert Hoare, indem er ihre Elemente durch einen Punkt trennt. So ist `<a.12, b.18, a.11, c.32>` ein Beispiel für eine Sequenz von vier Paaren. Für die Operation `Selection` verwendet er ebenso wie für die Operation `Count-Restricted` (siehe Kapitel 7.6.3.4) das Symbol ↓. Um die Operation `Selection` handelt es sich, wenn der erste Operand eine Sequenz von Paaren ist. Ein Beispiel für einen Ausdruck mit dieser Operation ist `(<a.12, b.18, a.11, c.32> ↓ a)`. Er liefert die Sequenz `<12,11>`.

Da in XQuery Sequenzen von Paaren nicht möglich sind, kann die Operation nicht direkt auf XQuery übertragen werden. Möglich sind aber XML-Elemente, die z. B. mit zwei Kindelementen oder Attributen Paare realisieren. Von diesen sind Sequenzen erlaubt, für die die Operation `Selection` in XQuery als benutzerdefinierte Funktion definiert werden kann. Dazu muss jedoch bekannt sein, mit welchen XML-Elementen die Paare wie realisiert werden. Davon abhängig müssen unterschiedliche Funktionen, spezifikationsspezifisch definiert werden. Daher soll hier keine Funktion angegeben werden.

Statt einer solchen Funktion können in XQuery auch direkt FLWOR- oder Pfadausdrücke verwendet werden, um ähnliches zu erreichen. In Sequenzen von XML-Elementen können einige XML-Elemente über in Attributen oder Kindelementen enthaltenen Werten ausgewählt werden. Andere in den ausgewählten XML-Elementen enthaltenen Werte können als Sequenz geliefert werden.

Um das zu veranschaulichen, wird die SOAP-Nachricht in Abbildung 54 auf Seite 100 als Beispiel verwendet. Ihr Element `/env:Envelope/env:Body/nwst:GetResponse` enthält eine Sequenz von Kindelementen `nwst:Message`. Jedes dieser Kindelemente hat seinerseits die Kindelemente `nwst:Category` und `nwst:Title`. Analog zur Operation `Selection` könnte nun gefordert sein, die Sequenz der Elemente `nwst:Title` zu ermitteln, für die das Element `nwst:Category` einen bestimmten Wert hat. Sei dieser Wert an die Variable `$category` gebunden und außerdem das Element `nwst:GetResponse` an die Variable `$getresponse`. Dann lässt sich die gesuchte Sequenz mit dem Pfadausdruck `$getresponse/nwst:Message[nwst:Category eq $category]/nwst:Title` ermitteln.

7.6.3.6 Nicht realisierbar

Während es spezifikationsspezifisch noch möglich ist, die Operation `Selection` als benutzerdefinierte Funktion zu realisieren, ist das für Hoares `Sternoperation für`

Funktionen[49] nicht der Fall. Die Sternoperation hat als Operanden eine Funktion, die eine Menge von Symbolen auf eine andere abbildet. Geliefert wird auch eine Funktion, die die gleiche Abbildung für alle Elemente einer Sequenz leistet. Hoare verwendet für die Sternoperation das Symbol *. Sei z. B. double eine Funktion, die eine Zahl verdoppelt. Dann ist double* eine Funktion, die eine Sequenz von Zahlen auf eine Sequenz von verdoppelten Zahlen abbildet. Z. B. liefert double*(<12,18,11,32>) die Sequenz <24,36,22,64>.

Die Sternoperation für Funktionen kann in XQuery nicht als benutzerdefinierte Funktion realisiert werden. Weder lässt es XQuery zu, Funktionen als Parameter von Funktionen zu übergeben, noch können Funktionen andere Funktionen als Ergebnis liefern. XQuery stellt auch keine spezielle Syntax zur Verfügung, auf die Sternoperation für Funktionen abgebildet werden könnte.

Statt die Sternoperation auf eine Funktion f anzuwenden, ist es daher Aufgabe von Entwicklern, eine benötigte Funktion f* zu realisieren. Solche können in XQuery nach einem festen Muster konstruiert werden, wobei stets ein FLWOR-Ausdruck verwendet wird. Das soll kurz vorgestellt werden. Es stehe eine Funktion f zur Verfügung, die einen Datentyp B in einen Datentyp A abbildet, wobei weder A noch B ein Datentyp für eine Sequenz sein darf. Dann lässt sich stets die Funktion f* (f-star) nach dem Muster in Abbildung 94 konstruieren. Darin sind f, f-star, A und B durch die wirklichen Namen zu ersetzen.

```
define function f-star($sx as B*)
  as A*
{
  for $x in $sx
    return f($x)
}
```

Abbildung 94: Muster zur Konstruktion von Funktionen f*

7.6.3.7 Im Kontext dieser Arbeit nicht zweckmäßig

Die anderen drei Operationen, die Hoare für Traces definiert, erscheinen im Kontext von SXQT nicht sinnvoll. Es sind die Sternoperation für Alphabete sowie die Operationen Catenation und Composition.

Die Sternoperation für Alphabete[50] bildet ein Alphabet auf die Menge aller endlichen Traces mit Elementen aus dem Alphabet ab. Diese Menge hat eine unendlich Zahl von Elementen. Wird die Operation in XQuery als benutzerdefinierte Funktion realisiert, kann schon das zu einem Problem führen. Einige XQuery-Prozessoren könnten versuchen dieser Menge tatsächlich im Hauptspeicher zu repräsentieren, was aufgrund der unendlichen Zahl von Elementen selbstverständlich nicht möglich ist.

Darüber hinaus muss beachtet werden, dass in SXQT Alphabete nicht wie bei Hoare Mengen von Ereignisnamen sind. In Kapitel 7.2.1 wurde festgelegt, dass ein Alphabet die in WSDL definierte Menge von SOAP-Nachrichten ist, die Webkomponenten austauschen können, die durch das WSDL-Dokument beschrieben werden. So wie XQuery in dieser Arbeit zum Formulieren von SXQT-Ausdrücken verwendet wird, enthält es

[49] Der Name Sternoperation für Funktionen wurde vom Autor dieser Arbeit gewählt. Hoare gibt dieser Operation keinen Namen an.

[50] Der Name Sternoperation für Alphabete wurde vom Autor dieser Arbeit gewählt. Hoare gibt dieser Operation keinen Namen an.

jedoch kein „Wissen" über WSDL-Dokumente. Damit können auch keine Aussagen über das Alphabet gemacht werden. Aus beiden genannten Gründen ist die `Sternoperation` für diese Arbeit nicht zweckmäßig.

Für diese Arbeit nicht sinnvoll ist auch Operation `Catenation`. Sie hat eine Sequenz als Operanden, die selbst Sequenzen als Elemente enthält. Die Konkatenation dieser Sequenzen liefert sie als Ergebnis. Hoare verwendet für sie das Symbol `^/` vor dem Operanden. Z. B. liefert der Ausdruck `(^/<<12,18>, <11,32>, <42,1>>)` die Sequenz `<12,18,11,32,42,1>`. Da XQuery keine Sequenzen von Sequenzen erlaubt, ist diese Operation für diese Arbeit nicht sinnvoll.

Die Operation `Composition`, verwendet Hoare, um Traces zu komponieren, die bei terminierenden Entitäten beobachtet werden. Hierzu definiert er das Symbol √, das für die Beobachtung der Terminierung einer Entität steht. Da in dieser Arbeit nur der Austausch von SOAP-Nachrichten beobachtet wird, ist es nicht möglich, die Terminierung einer Entität, also eines Webservices oder Webclients, zu beobachten. Damit entfällt für diese Arbeit auch der Bedarf für die Operation `Composition`. Sie wird ebenso wie die Operation `Catenation` und der `Sternoperation für Alphabete` in dieser Arbeit nicht weiter betrachtet.

Tabelle 4 fasst noch einmal die Operationen für Traces von Hoare zusammen.

Operation von Hoare	Kapitel	Realisierung
`Subscription`	7.6.3.1	Abgebildet auf Operation `[]` von XQuery.
`Concatenation`	7.6.3.1	Abgebildet auf Operation `,` von XQuery.
`Length`	7.6.3.1	Abgebildet auf Funktion `fn:count` von XQuery.
`Head`	7.6.3.2	Funktion `opr:head` angegeben.
`Tail`	7.6.3.2	Funktion `opr:tail` angegeben.
`Reverse`	7.6.3.2	Funktion `opr:reverse` angegeben.
`Prefix`	7.6.3.3	Funktion `opr:prefix` angegeben.
`Subsequence`	7.6.3.3	Funktion `opr:subsequence` angegeben.
`Interleaves`	7.6.3.3	Funktion `opr:interleaves` angegeben.
`Restriction`	7.6.3.4	Funktion `opr:restrict` angegeben.
`Count-Restricted`	7.6.3.4	Funktion `opr:count-restricted` angegeben.
`Selection`	7.6.3.5	Muss spezifikationsspezifisch realisiert werden.
`Sternoperation für Funktionen`	7.6.3.6	Nicht realisierbar.
`Sternoperation für Alphabete`	7.6.3.7	Für SXQT nicht zweckmäßig.
`Catenation`	7.6.3.7	Für SXQT nicht zweckmäßig.
`Composition`	7.6.3.7	Für SXQT nicht zweckmäßig.

Tabelle 4: Operationen für Traces von Hoare

7.6.4 Operationen für Traces von XQuery

Neben den Operationen für Traces, die Hoare definiert, gibt es Operationen für Sequenzen von XQuery, die für Traces geeignet sind. XQuery stellt für sie eine spezielle Syntax bereit oder realisiert sie als Standardfunktionen. Hier sollen nur solche Operationen von XQuery vorgestellt werden, die für die Verwendung mit Traces für SXQT besonders zweckmäßig erscheinen.

Das gilt zumindest für Pfadausdrücke, Quantoren und FLWOR-Ausdrücke, die mit einer speziellen Syntax realisiert sind. Die in Kapitel 7.4.6 vorgestellten Pfadausdrücke wurden zusammen mit Quantoren bereits in den Kapiteln 7.5.4 und 7.5.5 verwendet, um

erste Anforderungen als SXQT-Ausdrücke zu formulieren. Dabei wurde gezeigt, wie universell Pfadausdrücke sind. In Kapitel 7.6.1 wurde jedoch auch festgestellt, dass ihre Verwendung zu weniger übersichtlichen SXQT-Ausdrücken führt als die Verwendung von spezielleren Operationen. Sie zu verwenden, kann auch leicht zu Fehlern führen, die schwer zu finden sind. Zusätzlich zu spezielleren Operationen sind Pfadausdrücke jedoch wegen ihrer Universalität durchaus zweckmäßig.

Bei Pfadausdrücken ist zu beachten, dass sie stets Sequenzen liefern, deren Elemente in Dokumentenreihenfolge sind, also in der Reihenfolge, in der sie sich im XML-Dokument befinden. Für andere Reihenfolgen müssen die in Kapitel 7.4.8 vorgestellten FLWOR-Ausdrücke verwendet werde werden, wie bereits in Kapitel 7.6.3 mehrfach geschehen. Mit ihnen können auch Sequenzen sortiert werden.

Allerdings ist die Dokumentenreihenfolge für Message-Elemente in sofern interessant, weil es die Reihenfolge ist, in der die Ereignisse beobachtet wurden. Wird ein Ereignis beobachtet, wird es als Message-Element in Dokumentenreihenfolge am Ende des Traces zugefügt. Daher kann aus der Reihenfolge der Message-Elemente auf die Reihenfolge der Ereignisse geschlossen werden. Ob sich ein Message-Element in Dokumentenreihenfolge vor oder nach einem anderen im Trace befindet, kann mit einer der beiden in Kapitel 7.4.4 vorgestellten Reihenfolgevergleichsoperationen `<<` bzw. `>>` von XQuery ermittelt werden.

Auch die in Kaptitel 7.6.3.4 angesprochenen Mengenoperationen `union` (Vereinigungsmenge), `intersect` (Schnittmenge) und `except` (Differenzmenge) können für Traces interessant sein. Zwar sind Traces keine Mengen, vor allem weil die Reihenfolge ihrer Message-Elemente relevant ist. Die Mengenoperationen von XQuery sind aber so definiert, dass sie Sequenzen stets in Dokumentenreihenfolge liefern, also Ereignisse in Traces in der Reihenfolgen ihrer Beobachtung. Ebenso ist es für Traces eine zweckmäßig Semantik, dass Mengenoperationen Sequenzen liefern, die jedes Element nur einmal enthalten. Jedes Ereignis ist ohnehin nur einmal im Trace enthalten.

In XQuery gibt es eine Reihe von Standardfunktionen für den Umgang mit Sequenzen, die ebenfalls für Traces zweckmäßig sind. So erlaubt die Funktion `fn:insert`, eine Sequenz in eine andere an einer bestimmten Position einzufügen. Umgekehrt kann mit der Funktion `fn:remove` ein Element an einer bestimmten Position aus einer Sequenz entfernt werden. Sie wurde bereits in Kapitel 7.6.3.2 für die Realisierung der Funktion `opr:tail` verwendet. Mit der Funktion `fn:subsequence` kann ein Teil aus einer Sequenz extrahiert werden. Zu beachten ist, dass sie eine andere Semantik hat als das in Kapitel 7.6.3.3 vorgestellte Prädikat `opr:subsequence`.

Als Standardfunktionen stellt XQuery auch vier für Traces interessante Prädikate zur Verfügung. Es sind zunächst die Prädikate `fn:empty` und `fn:exists`, die prüfen, ob eine Sequenz leer bzw. nicht leer ist. Beide wurden bereits in Kapitel 7.6.3.3 für die Realisierung von Operationen von Hoare verwendet. Außerdem gibt es die Prädikate `fn:sequence-node-equal` und `fn:sequence-deep-equal`, die zwei Sequenzen auf Gleichheit prüfen. Beide Prädikate betrachten zwei Sequenzen als gleich, wenn sie die gleichen Elemente in der gleichen Reihenfolge enthalten. Sie unterschieden sich jedoch darin, wann sie zwei Elemente als gleich ansehen.

Für das Prädikat `fn:sequence-node-equal` sind zwei Elemente gleich, wenn sie die gleiche Identität haben. Da nur Knoten Identitäten haben, kann dieses Prädikat nur für Sequenzen von Knoten verwendet werden. Für Traces ist das Prädikat also anwendbar. Für sie wird geprüft, ob beide Trace die Beobachtungen derselben Ereignisse in der gleichen Reihenfolge enthalten.

Das Prädikat `fn:sequence-deep-equal` kann auch für Sequenzen verwendet werden, die nicht Knoten enthalten. Dazu werden zwei Elemente als gleich angesehen, wenn sie die gleichen Werte haben. Zwei Knoten haben den gleichen Wert, wenn sie vom gleichen Typ sind (Element, Attribut, ...), den gleichen qualifizierten Namen haben, sich ihre Kindelemente in der gleichen Reihenfolge befinden und diese ebenso wie ihre Attribute die gleichen Werte haben[51]. Traces können mit diesem Prädikat verglichen werden, wenn ihre Message-Elemente als gleich betrachtet werden sollen, wenn sie in Struktur und Werten übereinstimmen. Traces sind damit gleich, wenn sie die Beobachtungen identischer Ereignisse in der gleichen Reihenfolge repräsentieren[52].

7.6.5 Operationen für Traces dieser Arbeit

Nach den Operationen für Traces von Hoare und XQuery, sollen nun sechs weitere Operationen vorgestellt werden, die sich auf Traces beziehen und im Rahmen dieser Arbeit entwickelt wurden. Für sie werden Funktionen angegeben. Ihnen gemeinsam ist, dass sie „Wissen“ über die Struktur des Traces verwenden und diese in SXQT-Ausdrücken verbergen, so dass SXQT-Ausdrücke besser verständlich werden.

In SXQT-Ausdrücken in den Kapiteln 7.5.4 und 7.5.5 wurde stets mit dem Teilausdruck `fn:input()/tra:Trace/tra:Message` auf den Trace verwiesen. Um die Ausdrücke zu verkürzen, soll für diesen Teilausdruck die benutzerdefinierte Funktion `opr:tr` definiert werden, deren Realisierung in Abbildung 95 gezeigt wird. Sie benötigt keine Parameter und liefert den Trace, also die Sequenz von Message-Elementen.

```
define function opr:tr()
  as element tra:Message*
{
  fn:input()/tra:Trace/tra:Message
}
```

Abbildung 95: Realisierung der Funktion `opr:tr`

Die Funktion `opr:tr` kann in SXQT-Ausdrücken zum Ermitteln des Traces verwendet werden, ohne dass bekannt sein muss, was dazu im Detail zu geschehen hat. Sie verbirgt die Tatsache, dass die Input-Sequence nicht die Sequenz der Message-Elemente ist, sondern ein vervollständigtes XML-Dokument, das diese Sequenz enthält. (Siehe Kapitel 7.5.3.) Verborgen wird sogar die Tatsache, dass der Trace überhaupt mit Hilfe der Input-Sequence ermittelt wird.

In Kapitel 7.6.1 wurde auch festgestellt, dass für Teilausdrücke Operationen definiert werden sollten, mit denen zu einem Response der Request ermittelt wird, bzw. umgekehrt zu einem Request der Response. Hierzu werden die benutzerdefinierte Funktionen `opr:associated-request` und `opr:associated-response` definiert. Die Funktion `opr:associated-request` liefert für das Message-Element des Responses das des zugehörigen Requests. Die Funktion `opr:associated-response` leistet die entgegengesetzte Abbildung.

Realisierungen für beide Funktionen werden in Abbildung 96 gezeigt. Beide Funktionen haben einen Parameter mit dem Typ `element tra:Message` für ein Message-Element.

---

[51] Wann zwei Knoten im Sinne der Funktion `fn:sequence-deep-equal` den gleichen Wert haben, ist vereinfacht dargestellt. Für eine vollständige Darstellung siehe [Malhotra 02].

[52] Tatsächlich gilt diese Aussage nur, wenn für in beiden Traces vom Beobachter für die beobachteten Request-/Response-Paare im Attribut `tra:Message/@operation` stets die gleichen Werte gewählt wurden. Ansonsten unterscheiden sich auch bei gleicher Beobachtung die Message-Elemente in diesen Werten.

Im Normalfall liefern auch beide Funktionen ein Message-Element. Fehlt jedoch der gesuchte Request bzw. Response im Trace, kann auch die leere Sequenz geliefert werden. Daher ist der Typ des Rückgabeparameters `element tra:Message?`, der auch die leere Sequenz erlaubt. Der gesuchte Response kann zu einem Request fehlen, weil er noch nicht beobachtet wurde. Der Request kann zu einem Response fehlen, weil er vor Beginn der Beobachtung ausgetauscht wurde, wie in Kapitel 7.3.5 diskutiert.

```
define function opr:associated-request($m as element tra:Message)
  as element tra:Message?
{
   opr:tr()[@to="Service" and @operation = $m/@operation]
}

define function opr:associated-response($m as element tra:Message)
  as element tra:Message?
{
   opr:tr()[@to="Client" and @operation = $m/@operation]
}
```

Abbildung 96: Realisierungen von `opr:associated-request` und `opr:associated-response`

Obwohl beiden Funktionen ein Message-Element als Parameter übergeben wird und sie höchstens eines liefern, beziehen sie sich doch auf implizit auf den Trace. Anders als bei den Operationen für Traces von Hoare und allen bisher gezeigten, wird hier der Trace jedoch nicht explizit als Parameter übergeben. Statt dessen verwenden beide Funktionen die eben eingeführte Funktion `opr:tr`, um den Trace zu ermitteln. Der Trace wird danach auf das gesuchte Message-Element eingeschränkt. Hierzu werden die gleichen Prädikate verwendet, wie im Kapitel 7.5.5 in Abbildung 86 bzw. Abbildung 87.

Zwei weitere Operationen sollen vorgeschlagen werden, mit denen die Requests bzw. die Responses im Trace ermittelt werden können. Das hat Ähnlichkeit mit der in Kapitel 7.6.3.4 vorgestellten Operation `Restriction` von Hoare haben. Auch mit ihr werden Traces auf bestimmte Ereignisklassen eingeschränkt.

Beide Operationen sollen nicht nur für den beobachteten Trace verwendbar sein, der von der Funktion `opr:tr` geliefert wird. Sie sollen auch auf andere Sequenzen von Message-Elementen angewendet werden können, die mit Hilfe anderer Operationen aus dem Trace berechnet worden sein können. Z. B. könnte der Trace zunächst mit einem FLWOR-Ausdruck sortiert worden sein. Auch die sortierte Sequenz soll danach auf alle Requests bzw. Responses beschränkt werden können. Daher haben die beiden in Abbildung 97 gezeigten Funktionen, die die Operationen realisieren, jeweils einen Parameter in dem eine Sequenz auf Message-Elementen übergeben wird.

```
define function opr:requests($t as element tra:Message*)
  as element tra:Message*
{
  for $m in $t
    where opr:event-direction($m) eq "Service"
    return $m
}

define function opr:responses($t as element tra:Message*)
  as element tra:Message*
{
  for $m in $t
    where opr:event-direction($m) eq "Client"
    return $m
}
```

Abbildung 97: Realisierungen der Funktionen `opr:requests` und `opr:responses`

Die Tatsache, dass Funktionen auch bereits vorsortierte Sequenzen von Message-Elementen verarbeiten können sollen, verbietet, dass für ihre Realisierung Pfadausdrücke verwendet werden. Pfadausdrücke liefern Sequenzen stets in Dokumentenreihenfolge. Eine vorherige Sortierung würde verloren gehen. Statt dessen muss ein FLWOR-Ausdruck verwendet werden, wie in Abbildung 97 geschehen. Bei diesen bleibt eine Sortierung erhalten.

In der Funktion `opr:requests` prüft der FLWOR-Ausdruck für jedes Message-Element der übergebenen Sequenz `$t`, ob es ein Request ist. Nur solche Message-Elemente werden in einer Sequenz in ungeänderter Reihenfolge zurückgeliefert. Analog liefert die Funktion `opr:responses` eine Sequenz mit den Message-Elementen die Responses enthalten. In beiden Fällen verwendet der FLWOR-Ausdruck die in Kapitel 7.6.2 vorgestellte Funktion `opr:event-direction`, um aus dem Message-Element die Richtung zu ermitteln, in der die beobachtete SOAP-Nachricht ausgetauscht wurde. Diese liefert bei einem Request den Wert `Service` und bei einem Response den Wert `Client`.

Die sechste in dieser Arbeit entwickelte Operation wird als Funktion `opr:tr-safe` realisiert. Sie liefert wie die Funktion `opr:tr` den Trace, wobei aber die Message-Elemente aller Responses entfernt wurden, für die der Request im Trace fehlt. In Kapitel 7.3.5 wurde festgestellt, dass für das Formulieren von Anforderungen an Request-/Response-Paare das Problem existiert, dass Responses beobachtet werden können, für die der Request vor Beginn der Beobachtung ausgetauscht wurde und daher im Trace fehlt. Es wurde empfohlen solche Responses im Trace zu ignorieren. Greift ein SXQT-Ausdruck statt mit `opr:tr` mit der Funktion `opr:tr-safe` auf den Trace zu, filtert diese solche Responses aus, so dass es kein Aufwand verursacht sie zu ignorieren.

Die Realisierung der Funktion `opr:tr-safe` ist in Abbildung 98 gezeigt. Verwendet wird ein FLWOR-Ausdruck, der für jedes Message-Element im Trace prüft, ob es entweder ein Request ist oder ob im Falle eines Responses der zugehörige Request existiert. Alle Message-Elemente, auf die das zutrifft, werden in einer Sequenz geliefert.

```
define function opr:tr-safe()
  as element tra:Message*
{
  for $m in opr:tr()
    where (opr:event-direction($m) eq "Client")
          or fn:exists(opr:associated-request($m))
    return $m
}
```

Abbildung 98: Realisierung der Funktion `opr:tr-save`

7.6.6 Spezifikationsspezifische Filteroperationen für die Initialisierung

In Kapitel 7.3.5 wurde festgestellt, dass alle SXQT-Ausdrücke vom Startzustand unabhängig gemacht werden müssen. In einfachen Fällen sind sie es ohnehin, ohne zusätzlichen Aufwand. In Fällen, bei denen das Problem nur in nicht beobachteten Requests zu beobachten Responses besteht, kann die gerade im Kapitel 7.6.5 vorgestellte Funktion `opr:tr-safe` statt der Funktion `opr:tr` verwendet werden, um es zu lösen. Reicht auch das nicht, müssen Filteroperationen spezifikationsspezifisch realisiert werden, die Message-Elemente ausfiltern, die vor einem definierten Startzustand beobachtete SOAP-Nachrichten repräsentieren. SXQT-Ausdrücke sehen dann nur noch nachfolgende SOAP-Nachrichten und können daher den Startzustand voraussetzen.

Realisiert werden spezifikationsspezifische Filteroperationen ähnlich wie `opr:tr-safe` als Funktionen. Sie liefern den gefilterten Trace. In SXQT-Ausdrücken wird mit ihnen statt mit der Funktion `opr:tr` auf den gefilterten Trace zugegriffen. Sie können dann vom definierten Startzustand ausgehen, der durch die Filteroperation „simuliert“ wird.

Als Beispiel soll eine Filteroperation für SXQT-Ausdrücke zur Spezifikation des Sperrmechanismusses der Webkomponente für eine Internetzeitung entwickelt werden. In Kapitel 6.1.3 wurden Anforderungen an den Sperrmechanismus beschrieben. Für die SXQT-Ausdrücke soll ein Startzustand vorausgesetzt werden können, in dem keine Sperren gesetzt sind.

Wie schon in Kapitel 7.3.5 festgestellt, muss die Schnittstelle der Webkomponente hierzu um eine Operation erweitert werden, die alle Sperren freigibt. Es sei die Operation `UnlockAll`. Nach Beobachtung eines Responses dieser Operation kann angenommen werden, dass alle Sperren freigegeben sind. Abbildung 99 zeigt einen solchen Response.

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope"
              xmlns:nwst="http://example.org/NewsService020917/Types">
  <env:Body>
    <nwst:UnlockAllResponse />
  </env:Body>
</env:Envelope>
```

Abbildung 99: Beispiel für Response der Operation `UnlockAll`

Die Filteroperation muss eine Sequenz mit allen Message-Elementen nach dem ersten Message-Element mit einem Response der Operation `UnlockAll` liefern. Wird diese Sequenz in SXQT-Ausdrücken als Trace verwendet, kann in ihnen als Startzustand vorausgesetzt werden, dass keine Sperren gesetzt sind. Abbildung 100 zeigt eine mögliche Realisierung der Filteroperation als Funktion.

```
define function my:tr-filtered()
  as element tra:Message*
{
  let $u
    := opr:restrict(opr:tr(),
                    fn:QName-in-context("nwst:UnlockAllResponse",
                                        fn:false()))
       [1]
    return
      if (fn:empty($u)) then
        ()
       else
         for $m in opr:tr()
           where $u << $m
           return $m
}
```

Abbildung 100: Beispiel für spezifikationsspezifische Filteroperationen

Zunächst bestimmt die Filteroperation das erste Message-Element eines Responses der Operation `UnlockAll`. Dazu ermittelt sie den Trace mit der Funktion `opr:tr` und beschränkt ihn mit der Restriction auf alle Message-Elemente, die Responses der Operation `UnlockAll` enthalten. Das Prädikat `[1]` entnimmt aus der so berechneten Sequenz das erste Message-Element. Mit einem FLWOR-Ausdruck (`let $u` ...) wird es an die Variablen `$u` gebunden. Gab es kein solches Message-Element im Trace wird die leere Sequenz an `$u` gebunden. Mit einem bedingten Ausdruck (`if (fn:empty($u))` ...) wird geprüft, ob das der Fall war. Wenn ja, liefert der Filterausdruck die leere Sequenz. Das entspricht einem Trace, wenn noch kein Ereignis beobachtet wurde.

Gab es dagegen bereits einen Response der Operation `UnlockAll`, ist dessen Message-Element an `$u` gebunden. Dann wird mit einem zweiten FLWOR-Ausdruck (`for $m` ...) die Sequenz der Message-Elemente erstellt, die nach dem Response beobachtete Ereignisse repräsentieren.

7.6.7 Anwendung von Operationen in Spezifikationen

Mit den vorgestellten Operationen können SXQT-Ausdrücke einfacher, prägnanter und weniger fehleranfällig konstruiert werden. Dass soll nun an zwei Beispielen aus den Kapiteln 7.5.4 und 7.5.5 demonstriert werden. Dort wurden SXQT-Ausdrücke nur mit grundlegenden Operationen von XQuery formuliert.

Zunächst soll der SXQT-Ausdruck Abbildung 84 auf Seite 153 neu formuliert werden, was in Abbildung 101 geschehen ist. Dabei wurde die Funktion `opr:tr` verwendet, um den Trace zu ermitteln, auf den sich die Anforderung bezieht. Ihr Aufruf ersetzt den Ausdruck `fn:input()/tra:Trace/tra:Message` aus Abbildung 84, was viel prägnanter ausgedrückt, dass der Trace gemeint ist.

```
declare namespace opr="http://ti5.tu-harburg.de/venzke/20021015/operations"
declare namespace nwst="http://example.org/NewsService020917/Types"

every $m
  in opr:restrict(opr:tr(),
                  fn:QName-in-context("nwst:CategoriesResponse",
                                      fn:false()))
  satisfies
    opr:event-body-entry($m)/nwst:NoCategories
    eq fn:count(opr:event-body-entry($m)
                /nwst:CategoriesSequence/nwst:Category)
```

Abbildung 101: Anforderung an einzelne SOAP-Nachrichten, notiert mit Operationen

Die Prägnanz wird auch durch die Verwendung der Funktion `opr:restrict` erhöht. Ihr wird der mit der Funktion `opr:tr` ermittelte Trace übergeben. Sie liefert nur die Message-Elemente, deren Ereignisse den Ereignisnamen `nwst:CategoriesResponse` haben. In Abbildung 84 wird gleiches durch ein Prädikat in einem Pfadausdruck erreicht, was weniger prägnant ist, als die Verwendung der spezielleren Funktion `opr:restrict`.

Der Funktion `opr:restrict` wird der qualifizierter Name `nwst:CategoriesResponse` als Ereignisname übergeben. Leider erlaubt es XQuery nicht, qualifizierte Namen direkt als Literal anzugeben. Statt dessen müssen sie, wie in Kapitel 7.4.3 festgestellt, mit der Funktion `fn:QName-in-context` erzeugt werden. Das ist in Abbildung 101 geschehen. Im ersten Parameter wird ihr der qualifizierte Name als Zeichenkette angegeben.

In Abbildung 84 war es an zwei Stellen notwendig, aus einem Message-Element den Bodyeintrag zu ermitteln. Verwendet wurde hierzu in beiden Fällen der Pfadausdruck `$m/env:Envelope/env:Body/nwst:CategoriesResponse`, der in Abbildung 101 prägnanter als `opr:event-body-entry($m)` formuliert werden konnte.

Die Verwendung der Funktionen `opr:tr`, `opr:restrict` und `opr:event-body-entry` erlaubt, den SXQT-Ausdruck ohne Kenntnisse über den genauen Aufbau des Traces sowie der SOAP-Nachrichten zu notieren. Die zugehörigen Elemente `tra:Trace`, `tra:Message`, `env:Envelope` und `env:Body` wurden durch die Funktionen verborgen. Daher müssen im Query-Prolog nicht mehr die beiden Präfixe `tra` und `env` definiert werden. Bekannt sein muss jedoch der genaue Aufbau des anwendungsspezifischen Bodyeintrages.

Auch der in Abbildung 86 auf Seite 155 gezeigte SXQT-Ausdruck, mit dem eine Anforderung an einzelne Request-/Response-Paare formuliert wurde, lässt sich mit Hilfe der Operationen prägnanter notieren, wie in Abbildung 102 gezeigt. Wieder wird der

ermittelte Trace mit der Funktion `opr:restrict` auf bestimmte Message-Elemente beschränkt, nämlich auf solche der Ereignisklassen `nwst:GetResponse`. Auch die Funktion `opr:event-body-entry` wird wieder eingesetzt, um aus Message-Elementen die Bodyeinträge zu ermitteln.

```
declare namespace opr="http://ti5.tu-harburg.de/venzke/20021015/operations"
declare namespace nwst="http://example.org/NewsService020917/Types"

every $m
  in opr:restrict(opr:tr-safe(),
                  fn:QName-in-context ("nwst:GetResponse",
                                       fn:false()))
  satisfies
    fn:count(opr:event-body-entry($m)/nwst:Message)
    eq fn:count(opr:event-body-entry(opr:associated-request($m))
                /nwst:MessageIDsSequence/nwst:MessageID)
```

Abbildung 102: Anforderung an einzelne Request-/Response-Paare, notiert mit Operationen

Neu ist in Abbildung 102, dass zum Message-Element des Responses mit der Funktion `opr:associated-request` das Message-Element des zugehörigen Requests ermittelt wird. Hierzu war es in Abbildung 86 erforderlich, mit einem Pfadausdruck zunächst den Trace zu bestimmen und diesen mit einem Prädikat auf das Message-Element mit dem Request einzuschränken. Wieder führt die Verwendung der spezielleren Operation `opr:associated-request` zu einem prägnanteren SXQT-Ausdruck.

Der SXQT-Ausdruck liefert nur korrekte Ergebnisse, wenn zu jedem betrachteten Response der zugehörige Request ermittelt werden kann. Diese Tatsache wurde im SXQT-Ausdruck in Abbildung 86 ignoriert. In Abbildung 102 wurde sie dadurch berücksichtigt, dass der Trace statt mit der Funktion `opr:tr` mit `opr:tr-safe` ermittelt wird. Die Funktion `opr:tr-safe` liefert den Trace ohne Responses für die der zugehörige Request fehlt. Im SXQT-Ausdruck müssen fehlende Requests daher nicht mehr berücksichtigt werden.

Die erhöhte Prägnanz der SXQT-Ausdrücke ist also ein stets erreichter Vorteil von Operationen. Der Name einer Operation sagt aus, was mit einem Trace bzw. Message-Element geschehen soll. Das ist besser verständlich als der Teilausdruck, den die Operation ersetzt. Gleichzeitig können so die Elemente des Traces und der SOAP-Nachricht verborgen werden, so das detaillierte Kenntnisse über sie für das Verständnis der SXQT-Ausdrücke nicht erforderlich sind. Lediglich die im WSDL-Dokument beschriebenen Bodyeinträge müssen bekannt sein, ebenso wie Header- und Detaileinträge sowie ihre Nachfahren.

Die erhöhte Prägnanz und das damit verbundene bessere Verständnis der SXQT-Ausdrücke kann z. T. vermeiden, dass Entwickler sie fehlerhaft formulieren. Außerdem führt die Verwendung von Operationen dazu, dass Fehler in SXQT-Ausdrücken leichter zu erkennen sind. Im Kapitel 7.6.1 wurde ausgeführt, wie „kleine Tippfehler" in Pfadausdrücken dazu führen können, dass ein SXQT-Ausdruck einen falschen Wert, z. B. immer `true` oder `false` liefert. Wird der Name einer Funktion falsch geschrieben, führt das bereits vor Auswertung des SXQT-Ausdrucks zu einem statischen Fehler. Dieser kann eindeutig erkannt werden. Die erhöhte Prägnanz und die Vermeidung von Fehlern sind also zwei wichtige Gründe, warum Operationen für das Formulieren von SXQT-Ausdrücken äußerst hilfreich sind.

7.7 Beispiele und Klassifikation von SXQT-Ausdrücken

Um die Mächtigkeit des entwickelten Spezifikationsverfahren für Webservices weiter zu demonstrieren, soll in diesem Kapitel für weitere Anforderungen gezeigt werden, wie sie sich als SXQT-Ausdrücke formulieren lassen. Ebenso wie die in den Kapiteln 7.5.4 und 7.5.5 bzw. 7.6.7, sollen auch hier Anforderungen aus den Kapiteln 6.1 und 6.2 als Beispiele dienen.

Dabei sollen die Anforderungen zunächst in sieben Klassen[53] geordnet werden. Die verwendete Klassifikation wird nachfolgend in Kapitel 7.7.1 vorgestellt. Neben der Möglichkeit mit ihr Anforderungen zu ordnen, erlaubt es die Klassifikation auch Eigenschaften abzuleiten, die für alle Anforderungen einer Klasse gelten. Insbesondere gilt das für die Relevanz von Sichten und Startzuständen, was in Kapitel 7.7.2 untersucht wird. Danach wird in Kapitel 7.7.3 und den nachfolgenden Kapiteln 7.7.4, 7.7.5 und 7.7.6 geordnet nach Klassen für eine Reihe von Anforderungen gezeigt werden, wie sie als SXQT-Ausdruck formuliert werden können.

7.7.1 Eine Klassifikation für Anforderungen

Die Klassifikation ordnet die Anforderungen an Schnittstellen von Webkomponenten nach zwei Kriterien. Sie untersucht welche SOAP-Nachrichten miteinander in Beziehung gesetzt werden und welche Entität die Anforderung erfüllen muss, der Webclient oder der Webservice. Da sich beide Kriterien auf Anforderungen beziehen und nicht auf SXQT-Ausdrücke, kann die Klassifikation auch unabhängig von SXQT verwendet werden.

Für das erste Kriterium, welche SOAP-Nachrichten miteinander in Beziehung gesetzt werden, sollen vier Fälle unterschieden werden. Drei wurden bereits in Kapitel 6.1 unterschieden. Anforderungen konnten sich auf einzelne SOAP-Nachrichten (A), einzelne Request-/Response-Paare (B) oder auf mehrere Operationsaufrufe (C) beziehen. Die drei Fällen sollen mit den Buchstaben A, B bzw. C gekennzeichnet werden, wie in Klammern angegeben.

Darüber hinaus wurde in Kapitel 6.2.3 eine Anforderung betrachtet, nach der in einer SOAP-Nachricht enthaltene Werte aus einer Menge stammen müssen, die von einer Operation des Webservices geliefert wird. Die Operation muss jedoch nie aufgerufen worden sein. Alleine die Tatsache, dass die Operation einen Wert liefern würde, wenn sie aufgerufen werden würde, macht den Wert in der SOAP-Nachricht gültig. Daher werden durch solche Anforderungen nicht mehrere Operationsaufrufe in Beziehung gesetzt. Die Anforderungen lassen sich in die bisherigen drei Falle nicht einordnen und werden daher einem vierten Fall zugeordnet, der mit dem Buchstaben D gekennzeichnet wird.

Das zweite Kriterium ordnet die Anforderungen danach, welche Entität sie erfüllen muss. Unterschieden wird, ob es ein Webclient (C) oder ein Webservice (S) ist. Die Klammern enthalten wieder die Buchstaben, mit denen die Möglichkeit gekennzeichnet ist.

Die Buchstaben der beiden Kriterien ergeben kombiniert ein zweibuchstabiges Kürzel, mit denen die 8 Klassen der Klassifikation bezeichnet werden. Dazu wird der Buchstabe des zweiten Kriterium (C oder S) zuerst notiert, danach der des ersten (A, B, C oder D). Tabelle 5 zeigt die möglichen Kürzel.

[53] In Kapitel 7.7.1 werden zunächst acht Klassen vorgestellt. Von diesen ist jedoch eine leer.

Anforderung bezogen auf: \ Anforderung muss erfüllen:	Webclient (C)	Webservice (S)
Einzelne SOAP-Nachrichten (A)	CA	SA
Einzelne Request-/Response-Paare (B)	CB	SB
Mehrere Operationensaufrufe (C)	CC	SC
Dynamischen Aufzählungstypen (D)	CD	SD

Tabelle 5: Klassen von Anforderungen an Schnittstellen von Webkomponenten

Leider ist die Frage, welche Entität eine Anforderung erfüllen muss, nicht trivial zu beantworten. Eine Anforderung kann mehrere SOAP-Nachrichten miteinander in Beziehung setzen, die von unterschiedlichen Entitäten gesendet bzw. empfangen werden. Welche dieser Entitäten ist dann dafür verantwortlich, die Anforderung zu erfüllen?

Eine ähnliche Frage wurde bereits in Kapitel 7.3.3 behandelt. Hier wurde gefragt, welche Entität gegen eine Spezifikation verstoßen hat, wenn ein Trace beobachtet wurde, der die Spezifikation nicht erfüllt. Im Unterschied zur Frage in diesem Kapitel ging es dort also um eine konkrete Beobachtung und die gesamte Spezifikation und nicht um eine Anforderung unabhängig von einer Beobachtung.

In Kapitel 7.3.3 wurde zunächst festgelegt, dass ein SOAP-Sender für die von ihm gesendeten SOAP-Nachrichten verantwortlich ist. Der Trace enthält viele Beobachtungen von SOAP-Nachrichten, auf die sich die Spezifikation bezieht. Es wurde festgelegt, dass das erste Ereignis im Trace für das die Spezifikation nicht mehr zutrifft, dasjenige ist, das gegen die Spezifikation verstößt. Sein SOAP-Sender ist für den Verstoß verantwortlich.

Leider lässt sich die Zuordnung der Verantwortung aus Kapitel 7.3.3 nicht direkt auf dieses Kapitel übertragen, weil es hier nicht um einen beobachteten Trace geht. Unabhängig von konkreten Traces muss die Verantwortung, dass eine Anforderung eingehalten wird, einer Entität zugeordnet werden.

Interessant ist, dass diese Zuordnung intuitiv in der Praxis oft kein Problem darstellt. So wurde z. B. in Kapitel 6.1.3 für die Webkomponente für eine Internetzeitung die Anforderung aufgestellt, dass der Webservice eine Fehlernachricht zurücksenden muss, wenn mit einem Operationsaufruf auf einen Artikel zugegriffen werden soll, für den aus einer anderen Session eine Sperre gesetzt ist. Ob eine Sperre gesetzt ist, kann mit Hilfe vorheriger Operationsaufrufe ermittelt werden, die Sperren setzten oder freigeben.

Hier sind offenbar eine Vielzahl von SOAP-Nachrichten zu betrachten, um zu entscheiden, ob die Anforderung nicht erfüllt wird, wenn keine Fehlernachricht gesendet wird. Neben dem Response, der eine Fehlernachricht sein müsste, sind das die unterschiedlichen SOAP-Nachrichten, mit denen Sperren gesetzt oder freigegeben werden. Trotzdem wird schon aus der Formulierung der Anforderung klar, dass es in der Verantwortung des Webservices liegt, sie zu erfüllen. Denn wird keine Fehlernachricht geschickt, obwohl der Artikel gesperrt ist, würde man nicht das Setzen der Sperre als Fehler betrachten. Als Fehler würde angesehen werden, dass die Fehlernachricht nicht geliefert wurde, mit der angezeigt wird, dass der Artikel gesperrt ist. Von allen SOAP-Nachrichten, die durch die Anforderung in Beziehung gesetzt werden, ist es im eben beschriebenen Beispiel also die letzte, der ein Verstoß gegen die Anforderung zugeordnet wird.

In dieser Arbeit soll diese für das Beispiel intuitive Regel verallgemeinert werden. Von allen SOAP-Nachrichten, die durch eine Anforderung miteinander in Beziehung gesetzt werden, ist es stets die letzte beobachtete, die einen Verstoß gegen eine Anforderung verursacht. Ihr SOAP-Sender sei für sie verantwortlich. Damit ist es die Aufgabe dieser Entität, die Anforderung zu erfüllen.

Diese Festlegung steht im Einklang mit Kapitel 7.3.3. Dort wurde festgelegt, welche Entität verantwortlich ist, wenn ein beobachteter Trace gegen die Spezifikation verstößt. Dort war es die Entität, die die erste SOAP-Nachricht im Trace gesendet hat, die gegen die Spezifikation verstößt.

7.7.2 Relevanz von Sichten und Startzustand

Die Klassifikation erlaubt, Eigenschaften von Anforderungen abzuleiten, die für alle Anforderungen einer Klasse gelten. Solche Eigenschaften sollen hier untersucht werden. Es geht darum, in wieweit Sichten (siehe Kapitel 7.3.4) und der Startzustand (siehe Kapitel 7.3.5) für SXQT-Ausdrücke eine Rolle spielen, mit die Anforderungen formuliert werden.

In Kapitel 7.3.4 wurde festgestellt, dass für einen SXQT-Ausdruck bekannt sein muss, ob er in der Webclientsicht (WSC), Webservicesicht (WSS) oder in beiden gültig ist. Durch ihn in Beziehung gesetzte SOAP-Nachrichten, müssen im Trace vorhanden sein. Dazu muss der Beobachter eine geeignete Sicht einnehmen, in der er alle benötigten SOAP-Nachrichten beobachten kann.

Es gibt jedoch auch SXQT-Ausdrücke, die von der Sicht unabhängig sind. Befinden sich Anforderungen in der Klasse CA oder SA, beziehen sie sich nur auf einzelne SOAP-Nachrichten. Werden sie als SXQT-Ausdrücke formuliert, setzten diese keine SOAP-Nachrichten miteinander in Beziehung. Daher spielt die Sicht für sie keine Rolle. Ähnlich verhält es sich auch bei Anforderungen der Klassen CB und SB. Unabhängig von der Sicht werden stets beide SOAP-Nachrichten eines Request-/Response-Paares beobachtet. Damit können sie auch in SXQT-Ausdrücken miteinander in Beziehung gesetzt werden.

Auch für Anforderungen der Klassen CD und SD spielt die Sicht keine Rolle. Sie beziehen sich auf eine SOAP-Nachricht und setzen sie mit einem Operationsaufruf in Beziehung, der jedoch nie stattgefunden haben muss. Wichtig ist lediglich, was der Operationsaufruf geliefert hätte, wenn er aufgerufen worden wäre. Da der Operationsaufruf ohnehin nicht stattgefunden haben muss, spielt es auch keine Rolle, ob der Beobachter ihn beobachtet hat. Damit kann die Sicht nur für Anforderungen der Klassen CC und SC eine Rolle spielen.

Ebenso kann für Klassen untersucht werden, ob für ihre Anforderungen der Startzustand eine Rolle spielen kann. Anforderungen der Klassen CB, SB, CC und SC setzen mehrere SOAP-Nachrichten miteinander in Beziehung. Es kann geschehen, dass einige dieser SOAP-Nachrichten schon vor Beginn der Beobachtung ausgetauscht wurden. Dann hat sie der Beobachter nicht beobachtet und in den Trace aufgezeichnet. Damit können sie nicht durch SXQT-Ausdrücke mit beobachteten SOAP-Nachrichten in Beziehung gesetzt werden.

SXQT-Ausdrücke müssen so formuliert werden, dass sie trotzdem korrekte Ergebnisse liefern, also vom Startzustand unabhängig sind. Da bei Anforderungen der Klasse CB und SB nur Request und Response eines Request-/Response-Paares miteinander in Beziehung gesetzt werden, muss für ihre SXQT-Ausdrücke nur das Fehlen nicht beobachteter Requests berücksichtigt werden. Das kann erreicht werden, indem der Trace mit der in Kapitel 7.6.5 vorgestellten Operation `opr:tr-safe` ermittelt wird. Damit gibt es für alle Anforderungen der Klassen CB und SB einen grundsätzlichen Weg, wie ihre SXQT-Ausdrücke vom Startzustand unabhängig konstruiert werden können. Das ist bei Anforderungen der Klassen CC und SC nicht der Fall.

Für Anforderungen der Klassen CA, SA, CD und SD spielt der Startzustand ohnehin keine Rolle. Im Falle von CA und SA werden nicht mehrere SOAP-Nachrichten miteinander in Beziehung gesetzt. Bei CD und SD muss eine SOAP-Nachricht mit einem Operationsaufruf in Beziehung gesetzt werden, der ohnehin nicht stattgefunden haben muss. Dann spielt es auch keine Rolle, ob er vor dem Anfang der Beobachtung stattgefunden hat.

Die Sicht und der Startzustand spielen also für unterschiedliche Klassen eine unterschiedliche Rolle. Für Anforderungen der Klassen CA, SA, CD und SD spielen Sicht und Startzustand überhaupt keine Rolle. Für Anforderungen der Klassen CB und SB ist die Sicht uninteressant. Der Startzustand kann jedoch wichtig sein. Es gibt aber eine generelle Lösung ihre SXQT-Ausdrücke vom Startzustand unabhängig zu machen. Für Anforderungen der Klassen CC und SC müssen schließlich Sicht und Startzustand beachtet werden. Für welche Sicht ihre SXQT-Ausdrücke gültig sind, muss angegeben werden. Spezifikationsspezifisch muss ein Weg gefunden werden, sie vom Startzustand unabhängig zu machen. Hinweise hierzu gab Kapitel 7.6.6.

7.7.3 Klassen CA und SA

Anforderungen der Klassen CA und SA beziehen sich nur auf einzelne SOAP-Nachrichten. Sie drücken in der SOAP-Nachricht Beziehungen zwischen Werten in Attributen und Blattelementen aus, die sich in WSDL nicht ausdrücken lassen. Das können Werte sein, die sich in einem XML-Fragment befinden, das im WSDL-Dokument vollständig in XML-Schema beschrieben wird, also innerhalb eines Body-, Header- oder Detaileintrages. Oder es werden Werte miteinander in Beziehung gesetzt, die sich in der SOAP-Nachricht in unabhängig voneinander beschriebenen XML-Fragmenten befinden.

Die beiden Klassen CA und SA unterscheiden sich darin, ob ein Webclient oder Webservice die Anforderung erfüllen muss. Diese Verantwortlichkeit lässt hier leicht angeben, weil nur einzelne SOAP-Nachrichten betrachtet werden. Ihr SOAP-Sender muss sicherstellen, dass die SOAP-Nachricht der Anforderung entspricht, die SOAP-Nachricht also im Sinne der Anforderung konsistent ist.

Für eine Anforderung der Klasse SA wurde bereits in den Kapiteln 7.5.4 und 7.6.7 untersucht, wie sie als SXQT-Ausdruck formuliert werden kann. Die Anforderung bezog sich auf den Bodyeintrag eines Responses einer Operation `Categories` und wurde in Kapitel 6.1.1 vorgestellt. In Kapitel 7.5.4 wurde zunächst gezeigt, wie die Anforderung mit grundlegenden Operationen von XQuery dargestellt werden kann. Der SXQT-Ausdruck in Abbildung 101 in Kapitel 7.6.7 wurde dagegen mit Hilfe der spezielleren Operationen aus Kapitel 7.6 prägnanter formuliert. Das soll auch für die nachfolgend gezeigten SXQT-Ausdrücke geschehen.

7.7.3.1 Optionale Parameter

Ein Beispiel für eine Anforderung der Klasse CA wurde in Kapitel 6.2.1 als Grenzfall vorgestellt. Es wurde gezeigt, dass es in XML-Schema für den Request einer Operation `Login` möglich ist festzulegen, dass die Angabe des Passwortes zusätzlich zum Benutzernamen optional ist. Es war jedoch nicht möglich anzugeben, dass das Passwort genau dann fehlen muss, wenn als Benutzername „Gast“ angegeben wird. Diese Anforderung lässt sich als SXQT-Ausdruck formulieren, wie in Abbildung 103 gezeigt. Dabei wurde wie auch in den nachfolgenden Abbildungen auf die Angabe des Query-Prologs verzichtet, um die Abbildung zu verkürzen.

```
every $m
  in opr:restrict(opr:tr(),
                  fn:QName-in-context("nwst:Login", fn:false()))
  satisfies (opr:event-body-entry($m)/nwst:Name eq "Gast")
            eq fn:empty(opr:event-body-entry($m)/nwst:Password)
```

Abbildung 103: Anforderung an optionalen Parameter als SXQT-Ausdruck

Zum Verständnis des SXQT-Ausdrucks in Abbildung 103 sei auf das Beispiel für einen Request der Operation `Login` in Abbildung 57 auf Seite 104 verwiesen. Der SXQT-Ausdruck beschränkt den Trace zunächst mit der Funktion `opr:restrict` auf Message-Elemente, die Requests der Operation `nwst:Login` enthalten. Mit einem Allquantor wird für alle diese Message-Elemente geprüft, ob entweder der Benutzername „Gast“ angegeben wurde oder das Element `nwst:Password`, aber nicht beides.

Das ließe sich gut, mit der logischen Operation `xor` ausdrücken. Da diese in XQuery jedoch nicht definiert ist, wird die Operation `eq` verwendet, für die für logische Werte `A`, `B` gilt: `A xor B = A eq fn:not(B)`. Die Operanden der Operation `eq` sind zwei logische Ausdrücke. Der erste ist `(opr:event-body-entry($m)/nwst:Name eq "Gast")` und bestimmt, ob der angegebene Benutzername „Gast“ ist. Der zweite Ausdruck ist `fn:empty(opr:event-body-entry($m)/nwst:Password)` und bestimmt, ob in der SOAP-Nachricht das Element `nwst:Password` fehlt.

7.7.3.2 Alternative Strukturen

Nach dem Grenzfall des optionalen Parameters, wurde in Kapitel 6.2.2 auf Seite 105 der Grenzfall der alternativen Strukturen vorgestellt. Er behandelt die Tatsache, dass es sich in XML-Schema nicht ausdrücken lässt, dass ein Element entweder Kindelemente oder ein Attribut hat. In XML-Schema lässt sich lediglich ausdrücken, dass sowohl das Attribut als auch die Kindelemente optional sind. Dann ist neben der gewollten Alternative auch erlaubt, dass sowohl das Attribut als auch die Kindelemente fehlen oder dass beides vorhanden ist.

In Kapitel 6.2.2 wurde das am Beispiel eines Elementes `my:a` vorgestellt, das entweder dass Attribut `ref` oder ein Kindelement `my:str` haben soll, aber nicht beides. Abbildung 59 auf Seite 105 zeigt zwei korrekte Elemente `my:a`. Ein Ausschnitt eines Schemadokumentes, das das Element deklariert, wird in Abbildung 60 auf Seite 106 gezeigt. Es erlaubt jedoch auch Elemente `my:a`, die weder das Attribut `ref`, noch das Kindelement `my:str` haben oder beides gleichzeitig, wie es in Abbildung 61 auf Seite 106 gezeigt wird.

Mit Hilfe eines SXQT-Ausdrucks zusätzlich zur Definition in XML-Schema kann festgelegt werden, dass Elemente `my:a` entweder das Attribut `ref` oder das Kindelement `my:str` haben müssen, aber nicht beides haben dürfen. Eine solche Anforderung bezieht sich nur auf eine einzelne SOAP-Nachrichten, die Request oder Response sein kann. Entsprechend ist sie in der Klasse CA bzw. SA. Sie lässt sich für alle Elemente `my:a` in einem Trace mit dem SXQT-Ausdruck in Abbildung 104 ausdrücken.

```
every $a in opr:tr()//my:a
  satisfies fn:not(fn:exists($a/@ref) eq fn:exists($a/my:str))
```

Abbildung 104: Anforderung an alternative Struktur als SXQT-Ausdruck

Anders als die meisten anderen SXQT-Ausdrücke dieser Arbeit, schränkt dieser den Trace nicht auf bestimmte Message-Elemente ein. Statt dessen ermittelt er direkt mit

dem Ausdruck `opr:tr()//my:a` alle Elemente `my:a`, die im Trace vorhanden sind. Es können direkte oder indirekte Nachfahren der Message-Elemente sein. Für jedes von ihnen wird mit dem Allquantor geprüft, ob die obige Anforderung gilt.

Statt der zweckmäßigen, aber nicht vorhandenen Operation `xor`, wird dazu wie im letzten Beispiel die Operation `eq` verwendet. Der Ausdruck `fn:exists($a/@ref)` prüft, ob das Element `my:a` das Attribut `ref` hat. Der Ausdruck `fn:exists($a/my:str)` prüft die Existenz von Kindelementen `my:str`. Beides wird mit der Operation `eq` verknüpft und vom Ergebnis die Negation gebildet, was einer Verknüpfung mit der Operation `xor` gleich kommt, da für logische Werte `A`, `B` gilt: `A xor B = fn:not(A eq B)`.

7.7.3.3 Fehlernachrichten

Als weiterer Grenzfall wurden in Kapitel 6.2.4 Fehlernachrichten betrachtet. In WSDL konnte festgelegt werden, welche Fehlernachrichten eine Operation liefern kann und welche Detailelemente die Fehlernachricht hat. Nicht definiert werden konnte jedoch, welcher Fehlercode, Subfehlercode und welche für Menschen lesbare Beschreibung in einer bestimmten Fehlernachricht geliefert wurden. Ebenso konnte nicht festgelegt werden, zu welchem Fehlercode ein Subfehlercode gehört, oder zu welcher für Menschen lesbaren Beschreibung. Auch Beziehungen von diesen zu Werten in den Detaileinträgen sind nicht möglich.

Anforderungen an solche Beziehungen lassen sich als SXQT-Ausdrücke darstellen. Handelt es sich nur um Beziehungen innerhalb der Fehlernachricht gehören die Anforderungen in die Klasse SA, weil nur Responses Fehlernachrichten sein können. Beispielhaft soll in Abbildung 105 ein SXQT-Ausdruck betrachtet werden, der für einen Subfehlercode angibt, zu welchem Fehlercode er gehört. Für alle Fehlernachrichten im Trace wird festgelegt, dass der Subfehlercode `nwse:InvalidMessageID` stets zum Fehlercode `env:Sender` gehört.

```
every $m
  in opr:restrict(opr:tr(),
                  fn:QName-in-context("env:Fault", fn:false()))
  satisfies
   (opr:event-body-entry($m)/env:Code/env:Subcode/env:Value
     ne fn:QName-in-context("nwse:InvalidMessageID", fn:false()))
   or (opr:event-body-entry($m)/env:Code/env:Value
       eq fn:QName-in-context("env:Sender", fn:false()))
```

Abbildung 105: Anforderung an Zuordnung von Subfehlercode zu Fehlercode als SXQT-Ausdruck

Ein Beispiel für eine Fehlernachricht mit Subfehlercode `nwse:InvalidMessageID` ist in Abbildung 65 auf Seite 109 gegeben. Der SXQT-Ausdruck in Abbildung 105 beschränkt zunächst den Trace mit der Funktion `opr:restrict` auf die Message-Elemente die Fehlernachrichten enthalten. In Kapitel 7.2.1 wurde festgestellt, dass alle Fehlernachrichten den Ereignisnamen `env:Fault` haben. Daher wird dieser bei der Funktion `opr:restrict` angegeben.

Für jedes Message-Element, das eine Fehlernachricht enthält, wird dann hinter dem Schlüsselwort `satisfies` geprüft, ob es entweder <u>nicht</u> den Subfehlercode `nwse:InvalidMessageID` hat oder aber den Fehlercode `env:Sender`. Das sind die beiden Fälle, in denen die obige Anforderung erfüllt ist. Auf diese Art wird die geforderte Beziehung formuliert. Ähnlich lassen sich auch andere Beziehungen innerhalb von Fehlernachrichten formulieren.

7.7.3.4 Existenz von Headereinträgen

Ebenfalls als Grenzfall wurden in Kapitel 6.2.5 Headereinträge untersucht. Solche können in WSDL definiert werden. Der WSDL-Standard erlaubt aber ausdrücklich, dass ein SOAP-Sender nicht definierte Headereinträge einer SOAP-Nachricht zufügt, die in der Regel der SOAP-Empfänger wiederum ignorieren darf. Darf der SOAP-Empfänger den Headereintrag nicht ignorieren, muss ein zwingender Headereintrag zugefügt werden, der also das Attribut `env:mustUnderstand` mit dem Wert `true` hat, wie in Kapitel 3.3 ausgeführt.

In Kapitel 6.2.5 wurde gezeigt, wie in WSDL für einen bestimmten Headereintrag definiert werden kann, ob er zwingend ist oder nicht. Es wurde jedoch auch festgestellt, dass das nicht allgemein für alle Headereinträge möglich ist.

Es lässt sich in WSDL z. B. nicht ausdrücken, dass ein Webservice nur bestimmte Headereinträge kennt und daher nur diese zwingend sein dürfen. Solche Anforderungen können aber als SXQT-Ausdrücke formuliert werden. Der SXQT-Ausdruck in Abbildung 106 ist hierfür ein Beispiel. Er legt fest, dass in Requests nur zwingende Headereinträge mit den qualifizierten Namen `nwst:TipURL` oder `nwfrw:AuthCookie` erlaubt sind.

```
every $m in opr:requests(opr:tr())
  satisfies
    every $h in opr:event-header-entries($m)
    satisfies
      not($h/@env:mustUnderstand)
      or (fn:node-name($h)
          eq fn:QName-in-context ("nwst:TipURL", fn:false()))
      or (fn:node-name($h)
          eq fn:QName-in-context ("nwfrw:AuthCookie", fn:false()))
```

Abbildung 106: Anforderung an Bekanntheit von Headereinträgen als SXQT-Ausdruck

Im Ausdruck werden zwei Allquantoren verwendet. Der erste prüft für alle Requests im Trace, ob der zweite zutrifft. Der zweite prüft ob alle Headereinträge des Requests bekannt sind oder ignoriert werden dürfen. Um eine Sequenz zu ermitteln, die nur die Message-Elemente der Requests enthält, wird die in Kapitel 7.6.5 eingeführte Funktion `opr:requests` verwendet. Aus jedem Message-Element wird im zweiten Allquantor eine Sequenz seiner Headereinträge mit der Funktion `opr:event-header-entries` ermittelt, die in Kapitel 7.6.2 eingeführt wurde.

Für jeden dieser Headereinträge wird geprüft, ob er bekannt ist oder ihm das Attribut `env:mustUnderstand` mit dem Wert `true` nicht zugefügt wurde, so dass er ignoriert werden darf. Letzteres wird im SXQT-Ausdruck zuerst geprüft. Ob das Attribut `env:mustUnderstand` nicht den Wert `true` hat oder fehlt, wird zusammen wird mit dem Ausdruck `fn:not($h/@env:mustUnderstand)` geprüft. Darin bestimmt `$h/@env:mustUnderstand` eine Sequenz, die entweder das Attribut enthält oder leer ist, wenn das Attribut nicht exsistiert. Da die Sequenz der Funktion `fn:not` übergeben wird, wird sie in einem logischen Wert umgewandelt. Ist die Sequenz leer, wird sie auf den logischen Wert `false` abgebildet. Enthält sie das Attribut, wird dessen logischer Wert verwendet. Damit trifft der Ausdruck `fn:not($h/@env:mustUnderstand)` zu, wenn das Attribut nicht vorhanden ist oder den Wert `false` enthält.

Dieser Ausdruck wird mit der Operation `or` mit zwei anderen logischen Ausdrücken verknüpft, die prüfen, ob der Headereintrag bekannt ist. Hierzu vergleichen sie den qualifizierten Namen des Headereintrages mit `nwst:TipURL` bzw. `nwfrw:AuthCookie`,

also den qualifizierten Namen der bekannten Headereinträge. Dabei wird die Standardfunktion `fn:node-name` verwendet, um aus einem Headereintrag seinen qualifizierten Namen zu ermitteln.

7.7.3.5 Beziehungen zu Headereinträgen

Die Anforderung, welche Headereinträge ein Webservice versteht, legt nur fest, welche Headereinträge zwingend sein dürfen. Damit bezieht sich die Anforderung nicht auf die gesamte SOAP-Nachricht, sondern nur auf einzelne Headereinträge. Es gibt jedoch auch Anforderungen, die Headereinträge mit anderen Header- oder Bodyeinträgen in Beziehung setzen. In Fehlernachrichten kann eine Beziehung auch zu Detaileinträgen, Fehlercodes und ähnlichem gefordert werden. Solche Anforderungen wurden bereits in den Kapiteln 6.1.1, 6.2.4 sowie 6.2.5 vorgestellt. Auch sie lassen sich mit SXQT-Ausdrücken formulieren.

Als Beispiel soll hier eine Anforderung aus Kapitel 6.1.1 dienen. Es ist eine Anforderung an eine Beziehung zwischen einem Header- und dem Bodyeintrag. Sie bezieht sich auf einen Response der Operation `List` der Webkomponente für eine Internetzeitung, der eine Sequenz von Elementen `nwst:MessageID` enthält. Dem Response kann ein Headereintrag `nwsext:AvailableMessages` zugefügt werden, für den in Kapitel 6.1.1 gefordert wurde, dass er stets die Anzahl der gelieferten Elemente `nwst:MessageID` enthält. Abbildung 52 auf Seite 99 zeigt ein Beispiel für eine solche SOAP-Nachricht.

```
every $m
  in opr:restrict(opr:tr(),
                  fn:QName-in-context("nwst:ListResponse",
                                      fn:false()))
  satisfies
    let $am := opr:event-header-entries($m)
               /self::nwsext:AvailableMessages
    return
      fn:empty($am)
      or ((fn:count($am) <= 1)
          and ($am
               eq fn:count(opr:event-body-entry($m)
                           /nwst:MessageID)))
```

Abbildung 107: Anforderung an Beziehung zwischen Header- und Bodyeintrag als SXQT-Ausdruck

Die Anforderung wird durch den in Abbildung 107 gezeigten SXQT-Ausdruck ausgedrückt. Der Allquantor betrachtet im Trace alle Message-Elemente mit Ereignisname `nwst:ListResponse`. Für jedes Message-Element wird zunächst die Sequenz seiner Headereinträge `nwsext:AvailableMessages` ermittelt und an die Variable `$am` gebunden. Dazu wird mit der Funktion `opr:event-header-entries` die Sequenz aller Headereinträge bestimmt und diese mit Hilfe eines Pfadausdruckes auf die Headereinträge `nwsext:AvailableMessages` eingeschränkt. Im Pfadausdruck ist die Achse `self` erforderlich, weil die Sequenz bereits die Headereinträge enthält.

Für die ermittelte Sequenz wird geprüft, ob für einen enthaltenen Headereintrag die Anforderung zutrifft. Ist die Sequenz leer, gibt es keinen solchen Headereintrag. Damit ist die Anforderung für das Message-Element erfüllt, was in Abbildung 107 durch den Teilausdruck `fn:empty($am)` ausgedrückt wird. Es sei ferner angenommen, dass nur höchstens ein Headereintrag `nwsext:AvailableMessages` im Response erlaubt ist, was als Teilausdruck `(fn:count($am) <= 1)` enthalten ist. Schließlich wird geprüft, ob der Wert im Headereintrag gerade die Anzahl der Elemente `nwst:MessageID` im Bodyeintrag ist.

7.7.4 Klassen CB und SB

Neben Anforderungen, die sich nur auf eine SOAP-Nachricht beziehen, gibt es auch solche, die mehrere SOAP-Nachrichten miteinander in Beziehung setzen. Setzt eine Anforderung gerade den Request und Response eines Request-/Response-Paares miteinander in Beziehung, gehört sie in die Klasse CB oder SB. Die Anforderung beschreibt dann einen einzelnen Operationsaufruf, der durch das Request-/Response-Paar realisiert wird.

Tatsächlich kann es keine Anforderungen geben, die zur Klasse CB gehören. Das folgt aus der folgenden Überlegung: Die Klassen CB und SB unterscheiden sich darin, welche Entität die Anforderung erfüllen muss. In Kapitel 7.7.1 wurde festgestellt, dass es der SOAP-Sender der letzten SOAP-Nachricht ist, die durch die Anforderung mit anderen SOAP-Nachrichten in Beziehung gesetzt wird. Eine Anforderung der Klassen CB und SB setzt die beiden SOAP-Nachrichten des Request-/Response-Paares miteinander in Beziehung. Von diesen ist die letzte stets der Response. Sein SOAP-Sender ist der Webservice, er ist für die Anforderung verantwortlich. Daher befindet sich eine solche Anforderung stets in der Klasse SB. Die Klasse CB ist leer.

Eine Anforderung der Klasse SB wurde bereits im Kapitel 6.1.2 vorgestellt und für sie in den Kapiteln 7.5.5 und 7.6.7 gezeigt, wie sie als SXQT-Ausdruck formuliert werden kann. Es handelte sich um eine Anforderung, die den Request der Operation `Get` der Webkomponente für eine Internetzeitung mit ihrem Response in Beziehung setzt. Im Bodyeintrag des Requests ist eine Sequenz von Elementen `nwst:MessageID` enthalten. Jede verweist auf einen Artikel der Internetzeitung. Alle diese Artikel sind im Bodyeintrag des Responses in einer Sequenz enthalten, jeweils als Element `nwst:Message` mit seinen Nachfahren.

Die Anforderung sagt aus, dass die Sequenzen im Request und Response die gleiche Anzahl von Elementen haben müssen. Für sie wurde zunächst in Kapitel 7.5.5 gezeigt, wie sie sich mit grundlegenden Operationen von XQuery darstellen lässt. In Abbildung 102 auf Seite 176 wurde der SXQT-Ausdruck dann prägnanter mit spezielleren Operationen aus Kapitel 7.6 formuliert.

Die betrachtete Anforderung setzt vom Request und Response jeweils die Bodyeinträge miteinander in Beziehung. Selbstverständlich können auch beliebig Teile der beiden SOAP-Nachrichten miteinander in Beziehung gesetzt werden, also auch Headereinträge und in Fehlernachrichten Detaileinträge, Fehlercodes, und ähnliches.

Nachfolgend soll das an einem Beispiel gezeigt werden, in dem der Bodyeintrag im Request mit dem Subfehlercode einer Fehlernachricht in Beziehung gesetzt wird, die als Response übertragen wird. Mit Anforderungen dieser Art kann festgelegt werden, wie sich der Webservice verhält, wenn er einen ungültigen Request empfängt.

Das Beispiel soll an die in Kapitel 6.2.1 vorgestellte Anforderung der Klasse CA angelehnt werden, für die in Kapitel 7.7.3.1 ein SXQT-Ausdruck angegeben wurde. Die Anforderung legt für Requests der Operation `Login` der Webkomponente für eine Internetzeitung fest, dass entweder beim Benutzernamen „Gast“ das Element `nwst:Password` fehlen oder es bei einem anderen Benutzernamen angegeben sein muss. Ein Beispiel für einen solchen Request wurde in Abbildung 57 auf Seite104 gezeigt.

Die nun betrachtete Anforderung soll festlegen, wie sich der Webservice verhält, wenn zwar ein anderer Benutzername als „Gast“ angegeben wird, aber das Passwort trotzdem fehlt. In Kapitel 6.2.4 wurde festgelegt, dass er dann eine Fehlernachricht senden muss, wie sie in Abbildung 108 gezeigt wird. Genauer soll die Anforderung festlegen, dass der Subfehlercode der Fehlernachricht den Wert `nwse:AccessDenied` enthalten muss. Mit

einer anderen Anforderung könnte diesem Subfehlercode fest ein bestimmter Fehlercode zugeordnet werden, wie in Kapitel 7.7.3.3 gezeigt und analog auch eine für Menschen lesbare Beschreibung. Dann führt die Beschreibung des Subfehlercode zu einer vollständigen Beschreibung der Fehlernachricht.

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope"
              xmlns:nwse="http://example.org/NewsService020917/Errors">
  <env:Body>
    <env:Fault>
      <env:Code>
        <env:Value>env:Sender</env:Value>
        <env:Subcode>
          <env:Value>nwse:AccessDenied</env:Value>
        </env:Subcode>
      </env:Code>
      <env:Reason>Access can not be granted.</env:Reason>
      <env:Detail />
    </env:Fault>
  </env:Body>
</env:Envelope>
```

Abbildung 108: Beispiel für SOAP-Fehlernachricht mit Subfehlercode `nwse:AccessDenied`

Ein SXQT-Ausdruck, der diese Anforderung ausdrückt, muss den Subfehlercode im Response mit dem Benutzernamen und dem Element `nwst:Password` im Request in Beziehung setzen. Das leistet der SXQT-Ausdruck in Abbildung 109. Er beschränkt den Trace zunächst auf solche Message-Elemente, die Requests der Operation `Login` enthalten. Im FLWOR-Ausdruck (`let $resp := ...`) wird dann für diese Message-Elemente versucht, den Response zu ermitteln und an die Variable `$resp` zu binden. Hierzu wird die in Kapitel 7.6.5 eingeführte Funktion `opr:associated-response` verwendet. Schließlich wird mit einem logischen Ausdruck ermittelt, ob die Anforderung für das Request-/Response-Paar erfüllt ist.

```
every $m
  in opr:restrict(opr:tr(),
                  fn:QName-in-context("nwst:Login", fn:false()))
  satisfies
    let $resp := opr:associated-response($m)
    return
      fn:not(opr:event-body-entry($m)/nwst:Name ne "Gast"
             and fn:empty(opr:event-body-entry($m)/nwst:Password))
      or fn:empty($resp)
      or opr:event-body-entry($resp)
              /env:Code/env:Subcode/env:Value
         eq fn:QName-in-context("nwse:AccessDenied", fn:false())
```

Abbildung 109: Anforderung an Subfehlercode als SXQT-Ausdruck

Der logische Ausdruck, der das leistet besteht aus drei Teilausdrücken, die mit der Operation `or` verknüpft sind. Jeder Teilausdruck beschreibt eine Möglichkeit, warum ein Request-/Response-Paar die Anforderung erfüllt. Der erste Teilausdruck untersucht im Request, ob ein anderer Benutzername als „Gast“ und gleichzeitig kein Element `nwst:Password` angegeben wurde. Ist das nicht der Fall, erfüllt das Request-/Response-Paar die Anforderung, weshalb die Funktion `fn:not` verwendet wird.

Die Anforderung wird auch erfüllt, wenn zu einem Request noch kein Response beobachtet wurde. Da dann nicht beurteilbar ist, ob der Response die geforderte Fehlernachricht enthält, muss die Anforderung zunächst als erfüllt angesehen werden. Statt dem Response hat dann die Funktion `opr:associated-response` eine leere Sequenz geliefert. Diese wurde an die Variable `$resp` gebunden. Daher prüft der zweite Teilausdruck `fn:empty($resp)`, ob die Variable eine leere Sequenz enthält.

Liefern die ersten beiden Teilausdrücke den logischen Wert `false`, wurde im Request ein anderer Benutzername als „Gast" und kein Element `nwst:Password` angegeben und es wurde außerdem bereits der zugehörige Response beobachtet. Der Response muss dann die geforderte Fehlernachricht mit dem Subfehlercode `nwse:AccessDenied` enthalten. Das testet der dritte Teilausdruck.

Interessant ist, dass der SXQT-Ausdruck in Abbildung 109 bereits ohne die Funktion `opr:tr-safe` vom Startzustand unabhängig ist. Für ihn spielt es keine Rolle, ob vor dem Beginn der Beobachtung ein Request übertragen wurde, für den später der Response beobachtet wird. Der Grund ist, dass von den Requests im Trace ausgegangen wird und diesem der jeweils zugehörige Response zugeordnet wird. Ein Response für den der Request nicht beobachtet wurde, wird daher durch die Anforderung überhaupt nicht betrachtet. Diese Vorgehensweise führt umgekehrt jedoch dazu, dass Requests betrachtet werden, für die der Response noch nicht beobachtet wurde. Daher war der eben beschriebene zweite Teilausdruck `fn:empty($resp)` erforderlich, um zu prüfen, ob sich der Response überhaupt im Trace befindet.

7.7.5 Klassen CC und SC

Setzt eine Anforderung mehrere SOAP-Nachrichten miteinander in Beziehung, die nicht nur zu einem Request-/Response-Paar gehören, handelt es sich um eine Anforderung der Klasse CC oder SC. Damit handelt es sich um Beziehungen zwischen mehreren Operationsaufrufen. Mit ihnen kann z. B. festgelegt werden, dass Operationen nur in einer bestimmten Reihenfolge aufgerufen werden dürfen. Häufiger handelt es sich aber um Beziehungen zwischen Parametern von Operationen. So könnte ein Parameter nur dann gültig sein, wenn er von einer anderen Operation vorher geliefert wurde oder gerade dann ungültig.

In Beziehung gesetzt werden in der Regel zumindest zwei SOAP-Nachrichten. Es können auch wesentlich mehr. Die Anzahl muss für eine Anforderung nicht festliegen. Zum Beispiel könnten SOAP-Nachricht miteinander in Beziehung gesetzt werden, für die eine bestimmte Eigenschaft gilt. Sollte es dann nur eine oder gar keine solche SOAP-Nachricht geben, kann es sogar geschehen, dass sich die Anforderung nur auf eine oder keine SOAP-Nachricht bezieht.

Für eine Anforderung liegt in der Regel fest, ob die letzte in Beziehung gesetzte SOAP-Nachricht ein Request oder Response ist. Ist es ein Request, ist der SOAP-Sender ein Webclient und die Anforderung gehört in die Klasse CC. Handelt es sich um einen Response, gehört sie in die Klasse SC, weil der SOAP-Sender der Webservice ist.

7.7.5.1 Stets unterschiedlicher Wert geliefert

Ein einfaches Beispiel für eine Anforderung der Klasse SC wurde bereits in Kapitel 6.1.3 vorgestellt. Sie setzt nur Responses einer einzigen Ereignisklasse miteinander in Beziehung. Zusammen mit anderen Anforderungen beschreibt sie den Mechanismus für Sessions, der von der Webkomponente für eine Internetzeitung verwendet wird. Der Mechanismus basiert auf der Idee, dass die Operation `Login` eine Session startet. Sie

liefert eine Session-ID, die bei nachfolgenden Operationen der gleichen Session als Parameter angegeben werden muss.

Damit Sessions voneinander unterschieden werden können, muss für jede Session eine andere Session-ID vergeben werden. Die betrachtete Anforderung legt daher fest, dass jeder Aufruf der Operation `Login` eine Session-ID liefern muss, die von allen vorher gelieferten unterschiedlich ist. Die Anforderung setzt also alle Responses der Operation `Login` im Trace miteinander in Beziehung und fordert, dass sie eine unterschiedliche Session-ID enthalten. Abbildung 55 auf Seite 102 zeigte ein Beispiel für einen solchen Response. Die Session-ID ist im Body-Eintrag im Element `nwst:SessionID` enthalten.

```
every $m
  in opr:restrict(opr:tr(),
                  fn:QName-in-context("nwst:LoginResponse",
                                      fn:false()))
  satisfies
    every $b
      in opr:restrict(opr:tr(),
                      fn:QName-in-context("nwst:LoginResponse",
                                          fn:false()))
      satisfies
        ($b is $m)
        or (opr:event-body-entry($m)/nwst:SessionID
            ne opr:event-body-entry($b)/nwst:SessionID)
```

Abbildung 110: Anforderung an gelieferte SessionID als SXQT-Ausdruck

Abbildung 110 zeigt einen SXQT-Ausdruck, der die Anforderung realisiert. Er verwendet zwei Allquantoren, die zusammen erlauben, dass jeder Response der Operation `Login` mit jedem anderen (und ihm selbst) verglichen wird. Dabei werden die Message-Elemente der Responses an die Variablen `$m` bzw. `$b` gebunden. Der Vergleich erfolgt im Ausdruck hinter dem zweiten Schlüsselwort `satisfies`. Alle diese Ausdrücke müssen den logischen Wert `true` liefern, damit der gesamte Ausdruck `true` wird. Sie liefern den Wert `true`, wenn es sich um dasselbe Message-Element handelt oder wenn sie eine unterschiedliche Session-ID enthalten. Die beiden Möglichkeiten werden durch zwei Teilausdrücke realisiert, die durch die Operation `or` miteinander verknüpft sind.

In Kapitel 7.7.2 wurde für Anforderungen der Klasse CC und SC festgestellt, dass für sie die Sicht des Beobachters und der Startzustand relevant sein kann. Interessanterweise hat der SXQT-Ausdruck in Abbildung 110 trotzdem in jeder Sicht und unabhängig vom Startzustand seine Gültigkeit. Es wurde keine Filterfunktion benötigt, um ihm vom Startzustand unabhängig zu machen. Der Grund ist, dass die Anforderung, nach der die Operation `Login` unterschiedliche Session-IDs liefern muss, natürlich davon unabhängig ist, welche Operationsaufrufe beobachtet wurden. Für die beobachteten muss sie gelten, unabhängig davon welche es sind. Das prüft der SXQT-Ausdruck. Damit ist er unabhängig von der Sicht und dem Startzustand gültig.

7.7.5.2 Festgelegte Verarbeitungsstrategie

Neben anwendungsspezifischen Anforderungen, wie den bisher diskutierten, kann es auch solche geben, die den Webservice oder Webclient unabhängig von der Anwendung beschreiben. In Kapitel 6.1.3 wurde ein Beispiel für eine solche Anforderung genannt: Es kann gefordert werden, dass der Webservice Requests nach der Strategie First-In-First-Out (FIFO) bearbeitet, also in der Reihenfolge in der er sie empfängt. In dieser Reihenfolge werden dann auch die Responses gesendet. Somit müssen die Requests und zugehörigen Responses im Trace die gleiche Reihenfolge haben.

```
every $m
  in opr:responses(opr:tr-safe())
  satisfies
    every $b
      in opr:responses(opr:tr-safe())
      satisfies
        not($b << $m)
        or (opr:associated-request($b)
            << opr:associated-request($m))
```

Abbildung 111: Anforderung an Verarbeitungsstrategie als SXQT-Ausdruck

Diese Anforderung ist in Abbildung 111 als SXQT-Ausdruck formuliert. Wie im vorherigen Beispiel werden zwei Allquantoren verwendet. Sie setzen alle Responses im Trace miteinander in Beziehung. Hierzu binden sie die Message-Elemente jedes Paares von Responses an die Variablen `$m` und `$b`. Für jedes Paar wird wieder nach den zweiten Schlüsselwort `satisfies` ein Ausdruck geprüft. Dieser prüft mit dem Teilausdruck `($b << $m)`, ob das Message-Element in `$b` vor dem in `$m` im Trace ist. Ist das der nicht Fall, trifft der Ausdruck zu. Ist es der Fall, müssen auch die Message-Elemente der zugehörigen Requests in der gleichen Reihenfolge sein. Das testet der Teilausdruck `(opr:associated-request($b) << opr:associated-request($m))`.

Wie beim vorherigen Beispiel in Kapitel 7.7.5.1 ist auch der SXQT-Ausdruck in Abbildung 111 von der Sicht unabhängig. Zwar befinden sich von ihr abhängig unterschiedliche Request-/Response-Paare im Trace. Für diese muss aber die Verarbeitungsstrategie eingehalten werden, was der SXQT-Ausdruck prüft.

Für den SXQT-Ausdruck ist jedoch wichtig, dass er nur Responses betrachtet, zu denen sich auch der Request im Trace befindet. Das wird erreicht, indem über die in Kapitel 7.6.5 vorgestellte Funktion `opr:tr-safe` auf den Trace zugegriffen wird, die nur Responses liefert, für die sich die Requests im Trace befinden.

7.7.5.3 Sperre muss gesetzt sein

Alle bisher gezeigten Beispiele für Anforderungen ließen sich mit wenigen Zeilen als SXQT-Ausdruck formulieren. Es gibt jedoch auch Anforderungen, über sehr viel komplexere Beziehungen, die auch zu komplexeren SXQT-Ausdrücken führen. Um sie übersichtlich notieren zu können, sollten dann Teilausdrücke als benutzerdefinierte Hilfsfunktionen formuliert werden.

Als Beispiel wird nachfolgend eine Anforderung untersucht, die einen Aspekt des Sperrmechanismus der Webkomponente für eine Internetzeitung beschreibt. Dort werden Sperren für Artikel der Internetzeitung im Kontext einer Session mit der Operation `Get` gesetzt, mit der auch der Inhalt von Artikeln abgefragt wird. Der gesperrte Artikel darf danach nur noch in dieser Session verändert werden, bis die Sperre wieder mit der Operation `Unlock` freigegeben wird. Automatisch wird die Sperre außerdem freigegeben, wenn die Session mit der Operation `Logout` beendet wird. Außerdem können mit der in Kapitel 7.6.6 zugefügten Operation `UnlockAll` alle Sperren freigegeben werden.

Der vollständige Sperrmechanismus kann durch eine Reihe von Anforderungen beschrieben werden, wie bereits im Kapitel 6.1.3 dargestellt. Hier soll nur eine Anforderung der Klasse CC weiter betrachtet werden. Sie besagt, dass ein Webclient die Operation `Unlock` für einen Artikel nur dann aufrufen darf, wenn vorher in der gleichen Session für den Artikel erfolgreich eine Sperre mit der Operation `Get` gesetzt wurde. Dabei bedeutet erfolgreich, dass die Operation `Get` nicht fehlgeschlagen sein darf, also nicht als Response eine Fehlernachricht geliefert hat.

Obwohl die Anforderung in natürlicher Sprache nicht wesentlich komplizierter erscheint als die vorheriger Beispiele, stellt sich bei genauerer Betrachtung heraus, dass bei ihr viele Punkte zu beachten sind. So kann innerhalb einer Session für einen Artikel mehrfach eine Sperre gesetzt und danach wieder freigegeben werden. Um festzustellen ob die Operation `Unlock` für einen Artikel erlaubt ist, reicht es daher nicht festzustellen, ob eine Sperre für den Artikel gesetzt wurde. Es muss auch sichergestellt werden, dass es seit dem Setzen der Sperre keine andere Operation `Unlock` gegeben hat, mit der die Sperre wieder freigegeben wurde.

Dabei muss stets beachtet werden, dass all das in der gleichen Session geschehen sein muss. Die Session kann sowohl bei der Operation `Get` als auch bei der Operation `Unlock` am Request erkannt werden. In beiden Fällen erhält dessen Bodyeintrag das Kindelement `nwst:SessionID`, das die Session-ID enthält, die die Session eindeutig identifiziert. Das ist in den Beispielen für Requests der Operationen `Get` und `Unlock` zu erkennen, die sich in Abbildung 56 auf Seite 103 bzw. in der nachfolgenden Abbildung 112 befinden.

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope"
              xmlns:nwst="http://example.org/NewsService020917/Types">
 <env:Body>
   <nwst:Unlock>
     <nwst:SessionID>8CCF4AB6-FF9E-451d-8675-1B4F56B05296</nwst:SessionID>
     <nwst:MessageIDsSequence>
       <nwst:MessageID>80AF91EA-32EB-4691-9ED5-D0BF047B9424</nwst:MessageID>
       <nwst:MessageID>AD6F371A-ABE9-41f5-8362-AAE7640F1DDD</nwst:MessageID>
     </nwst:MessageIDsSequence>
   </nwst:Unlock>
 </env:Body>
</env:Envelope>
```

Abbildung 112: Beispiel für Request der Operation `Unlock`

Die Komplexität des Problems erhöht sich dadurch, dass durch einen Operationsaufruf Sperren für viele Artikel gleichzeitig gesetzt bzw. freigegeben werden können. Requests der Operation `Get` und `Unlock` enthalten jeweils eine Sequenz von Elementen `nwst:MessageID`, die eine Menge von Artikeln bezeichnet. Der Operationsaufruf bezieht sich auf alle diese Artikel. Dabei muss die Menge, die mit einem Operationsaufruf von Unlock angegeben wurde, nicht mit der beim Operationsaufruf von `Get` identisch gewesen sein.

Erschwerend kommt außerdem hinzu, dass die Beobachtung eines Requests einer dieser Operationen auch nicht bedeutet, dass die Sperren gesetzt bzw. freigegeben wurden. Zunächst werden Sperren bei der Operation `Get` nur dann gesetzt, wenn der Bodyeintrag im Request ein Kindelement `nwst:Lock` hat, das den Wert `true` enthält. Außerdem können beide Operationen fehlschlagen. Dann haben sie keine Sperren verändert und liefern als Response eine Fehlernachricht.

Nebenläufige Operationsaufrufe führen weiterhin zur Frage, zu welchem Zeitpunkt eine Sperre genau gesetzt bzw. freigegeben wird. Sicherlich ist das nach Beobachtung des Requests und vor Beobachtung des Responses der Fall. Genauer lässt es sich aber nicht festlegen. Damit gibt es Fälle, in denen sich nicht entscheiden lässt, ob bei der Ausführung einer Operation eine Sperre gesetzt ist oder nicht. Für solche Fälle soll festgelegt werden, dass ein SXQT-Ausdruck auch dann den logischen Wert `true` liefert, wenn es

nur möglich ist, dass die Anforderung erfüllt ist. Ein Trace widerspricht nur der Spezifikation, wenn „bewiesen" werden kann, dass einer seiner Anforderungen widersprochen wurde. Dieses Prinzip wird auch dann angewendet, wenn zu einem Request noch nicht der Response beobachtet wurde und daher nicht entschieden werden kann, ob der Response eine Fehlernachricht ist.

Das alles muss berücksichtigt werden, wenn die Anforderung als SXQT-Ausdruck formuliert wird. Abbildung 113 zeigt einen SXQT-Ausdruck, bei dem das der Fall ist. Er lässt sich nur deshalb so kurz notieren, weil er zwei benutzerdefinierte Hilfsfunktionen verwendet, von denen die Funktion `my:lock-possibly-set` einen Grossteil der Komplexität verbirgt.

```
every $m
  in opr:restrict(my:tr-filtered(),
                  fn:QName-in-context("nwst:Unlock", fn:false()))
  satisfies
    every $messageID
      in opr:event-body-entry($m)
           /nwst:MessageIDsSequence/nwst:MessageID
    satisfies
      my:lock-possibly-set($messageID, sessionID($m), $m)
```

Abbildung 113: Anforderung an notwendige Sperre als SXQT-Ausdruck

Der SXQT-Ausdruck in Abbildung 113 prüft mit einem Allquantor, ob die Anforderung für alle Requests der Operation `Unlock` erfüllt ist. Dazu wird ein weiterer Allquantor verwendet, um jede im Request enthaltene Message-ID daraufhin zu untersuchen, ob für den Artikel, den sie bezeichnet, eine Sperre gesetzt ist. Sie muss zum Zeitpunkt direkt vor Beobachtung des Requests der Operation `Unlock` gesetzt sein, wobei das Sperren in der gleichen Session geschehen sein muss. Es reicht auch, wenn für einen Artikel nicht feststellbar ist, ob er gesperrt ist.

Dieser Test findet für einen Artikel in der Hilfsfunktion `my:lock-possibly-set` statt und kann daher in Abbildung 113 durch ihren Aufruf notiert werden. Sie benötigt hierzu drei Parameter, nämlich die Message-ID des Artikels, die Session-ID und das Message-Element im Trace, vor dessen Beobachtung die Sperre gesetzt sein muss. Die Message-ID und das Message-Element sind bereits durch die beiden Allquantoren an Variablen gebunden und können daher direkt an die Funktion übergeben werden. Die Session-ID muss dagegen zunächst aus dem Message-Element extrahiert werden, das den Request der Operation `Unlock` enthält. Hierzu wird die Hilfsfunktion `my:sessionID` verwendet, deren Realisierung in Abbildung 114 gezeigt wird.

```
define function my:sessionID($messageElement
                             as element tra:Message)
  as xsd:string
{
  opr:event-body-entry($messageElement)/nwst:SessionID
}
```

Abbildung 114: Benutzerdefinierte Hilfsfunktion `my:sessionID`

Abbildung 115 zeigt, wie die Funktion `my:lock-possibly-set` realisiert ist. Ihr Name bringt zum Ausdruck, dass sie auch dann den logischen Wert `true` liefert, wenn nur möglicherweise eine Sperre gesetzt ist. Sie verwendet einen Existenzquantor, der alle Requests der Operation `Get` daraufhin untersucht, ob sie die gesuchte Sperre setzen.

```
define function my:lock-possibly-set($messageID as xsd:string,
                                     $sessionID as xsd:string,
                                     $horizon
                                        as element tra:Message)
  as xsd:boolean
{
  some $lock
    in opr:restrict(my:tr-filtered(),
                    fn:QName-in-context("nwst:Get", fn:false()))
    satisfies
      ($lock << $horizon)
      and (my:sessionID($lock) eq $sessionID)
      and (opr:event-body-entry($lock)
                /nwst:MessageIDsSquence/nwst:MessageID
            = $messageID)
      and (opr:event-body-entry($lock)/nwst:Lock eq fn:true())
      and my:response-possibly-not-error($lock)
      and my:unlock-possibly-not-available($messageID,
                                           $sessionID,
                                           $lock,
                                           $horizon)
}
```

Abbildung 115: Benutzerdefinierte Hilfsfunktion `my:lock-possibly-set`

Hierzu wird der Trace zunächst auf alle Message-Elemente mit Requests der Operation `Get` beschränkt. Das ein Request die gesuchte Sperre setzt, wird daran erkannt, dass alle nachfolgend beschriebenen Teilausdrücken auf sein Message-Element zutreffen. Sie sind in Abbildung 115 hinter dem Schlüsselwort `satisfies` dargestellt und mit der Operation `and` verknüpft.

Der erste Teilausdruck `($lock << $horizon)` stellt sicher, dass die Sperre vor dem in `$horizon` übergebenen Request der Operation `Unlock` gesetzt wurde. Der zweite Teilausdruck `(my:sessionID($lock) eq $sessionID)` prüft, ob die Sperre in der gleichen Session gesetzt wurde. Um aus `$lock` die Session-ID zu ermitteln, wird dabei erneut die Hilfsfunktion `my:sessionID` aus Abbildung 114 verwendet.

Der nachfolgende Teilausdruck prüft, ob die übergebene Message-ID in `$messageID` im Request der Operation `Get` enthalten ist. Eine Sequenz der Message-IDs aus dem Request wird mit einem Pfadausdruck ermittelt. Sie wird mit der Operation `=` mit der übergebenen Message-ID verglichen, die prüft, ob die übergebene Message-ID in der Sequenz enthalten ist.

Die letzten drei Teilausdrücke prüfen, ob der Request der Operation `Get` das Element `nwst:Lock` mit Wert `true` enthält, möglicherweise der zum Request gehörende Response keine Fehlernachricht ist und ob es möglicherweise keinen Aufruf der Operation `Unlock` in der gleichen Session gibt, der die Sperre bereits wieder freigegeben hatte. Die letzten beiden Teilausdrücke werden dabei durch die benutzerdefinierten Hilfsfunktionen `my:response-possibly-not-error` und `my:unlock-possibly-not-available` realisiert, deren Realisierung in Abbildung 117 bzw. Abbildung 116 gezeigt wird.

Zunächst soll die Funktion `my:unlock-possibly-not-available` vorgestellt werden, die prüft, ob es in der gleichen Session möglicherweise keinen Aufruf der Operation `Unlock` gibt, der die Sperre bereits freigegeben hat. Gäbe es einen solchen Aufruf, müsste sich das Message-Element seines Requests im Trace zwischen dem des Requests der Operation `Get`, das die Sperre setzt, und dem des ursprünglichen Aufrufs der Ope-

ration `Unlock` befinden. Damit die Funktion `my:unlock-possibly-not-available` diese Grenzen kennt, werden ihr diese beiden Message-Elemente in den Parametern `$start` und `$horizon` übergeben. Zusätzlich sind die Parameter `$messageID` und `$sessionID` erforderlich, um den zu entsperrenden Artikel zu bezeichnen, sowie die Session, in der die Operation `Unlock` ausgeführt worden sein muss.

```
define function my:unlock-possibly-not-available
                                ($messageID as xsd:string,
                                 $sessionID as xsd:string,
                                 $start as element tra:message,
                                 $horizon as element tra:message)
  as xsd:boolean
{
  fn:not(
    some $unlock
      in opr:restrict(my:tr-filtered(),
                      fn:QName-in-context("nwst:Unlock",
                                          fn:false()))
      satisfies
        ($start << $unlock)
        and ($unlock << $horizon)
        and (my:sessionID($unlock) eq $sessionID)
        and (opr:event-body-entry($unlock)
                  /nwst:MessageIDsSquence/nwst:MessageID
              = $messageID)
        and (my:response-not-error($unlock)))
}
```

Abbildung 116: Benutzerdefinierte Hilfsfunktion `my:unlock-possibly-not-available`

Die Hilfsfunktion `my:unlock-possibly-not-available` in Abbildung 116 ist ähnlich konstruiert wie der Hilfsfunktion `my:lock-possibly-set` in Abbildung 115. Wieder wird ein Existenzquantor verwendet, der für Message-Elemente im Trace eine Reihe von Teilausdrücken prüft. Die untersuchten Message-Elemente enthalten hier jedoch Requests der Operation `Unlock`.

Für jedes dieser Message-Elemente wird mit den ersten beiden Teilausdrücken geprüft, ob es sich im Trace zwischen den Message-Elementen in `$start` und `$horizon` befindet. Die nächsten beiden stellen wie in Abbildung 115 sicher, dass der Request zur gleichen Session gehört und die Message-ID des zu entsperrenden Artikels enthält. Ebenso testet der letzte Teilausdruck wieder, ob der zugehörige Response keine Fehlernachricht ist. Dazu wird hier jedoch die Hilfsfunktion `my:response-not-error` verwendet, statt wie in Abbildung 115 `my:response-possibly-not-error`.

Beide Hilfsfunktionen prüfen, ob der Response zu einem Request keine Fehlernachricht ist. Der Request wird jeweils als Parameter übergeben. Die Hilfsfunktionen unterscheiden sich darin, was sie zurückliefern, wenn sich der Response nicht im Trace befindet. Dann liefert Hilfsfunktion `my:response-possibly-not-error` den logischen Wert `true`, weil der Response möglicherweise keine Fehlernachricht ist. Da aber auch eine Fehlernachricht möglich ist, liefert die Hilfsfunktion `my:response-not-error` den logischen Wert `false`. Dieses unterschiedliche Verhalten wird in den Hilfsfunktionen `my:lock-possibly-set` bzw. `my:unlock-possibly-not-available` benötigt um sicherzustellen, dass der SXQT-Ausdruck in Abbildung 113 im Zweifelsfall den logischen Wert `true` liefert.

```
define function my:response-possibly-not-error
                                ($request as element tra:Message)
  as xsd:boolean
{
  let $response := opr:associated-request($request)
  return
    fn:empty($response)
    or (opr:event-name($response)
        ne fn:QName-in-context("env:Fault", fn:false()))
}
```

Abbildung 117: Benutzerdefinierte Hilfsfunktion `my:response-possibly-not-error`

```
define function my:response-not-error
                             ($request as element tra:Message)
  as xsd:boolean
{
  let $response := opr:associated-request($request)
  return
    fn:exists($response)
    and (opr:event-name($response)
         ne fn:QName-in-context("env:Fault", fn:false()))
}
```

Abbildung 118: Benutzerdefinierte Hilfsfunktion `my:response-not-error`

Abbildung 117 und Abbildung 118 zeigen die Realisierungen der Hilfsfunktionen `my:response-possibly-not-error` und `my:response-not-error`. Beide verwenden die Funktion `opr:associated-request`, um zum Message-Element des Requests dasjenige des zugehörigen Responses zu ermitteln. Um zu prüfen, ob der Response eine Fehlernachricht ist, wird der Ereignisname des Responses mit dem qualifizierten Namen `env:Fault` verglichen, den alle Fehlernachrichten haben. Beide Hilfsfunktionen prüfen vorher, ob die Fehlernachricht vorhanden ist. Hierzu verwenden sie die Funktionen `fn:empty` bzw. `fn:exists`, die bei einer leeren Sequenz gerade den umgekehrten Wahrheitswert liefern.

Auch für den SXQT-Ausdruck in Abbildung 113 mit seinen Hilfsfunktionen muss untersucht werden, in welchen Sichten er gültig ist. Für ihn muss sichergestellt sein, dass zumindest alle die Request-/Response-Paare beobachtet werden, mit denen in einer Session Sperren gesetzt oder freigegeben werden. Nur so kann aus ihnen geschlossen werden, ob in der Session eine Sperre bereits gesetzt ist und damit durch Aufruf der Operation `Unlock` freigegeben werden darf.

Die Beobachtung dieser SOAP-Nachrichten ist sichergestellt, wenn der Webservice beobachtet wird. Wird der Webclient beobachtet, muss das nicht der Fall sein. Es wäre jedoch erfüllt, wenn alle SOAP-Nachrichten einer Session nur zwischen einem Webclient und dem Webservice ausgetauscht werden. Dann wäre der SXQT-Ausdruck wieder von der beobachteten Entität unabhängig. Ansonsten muss die Webservicesicht verwendet werden.

Vom Startzustand wäre der SXQT-Ausdruck abhängig, wenn nicht die spezifikationsspezifische Filteroperation `my:tr-filtered` verwendet worden wäre. Es wurde vorausgesetzt, dass sich zu jeder gesetzten Sperre auch der Request der Operation `Get` im Trace befindet, der sie setzt. Würde die Beobachtung nach dem Setzen einer Sperre begonnen, wäre das nicht der Fall.

Ein Startzustand, in dem keine Sperren gesetzt sind, würde die Spezifikation des Sperrmechanismusses stark vereinfachen. Dann würde sich auch zu jeder gesetzten Sperre der Request der Operation `Get` im Trace befinden, der sie setzt. Dieser Startzustand wird durch die in Kapitel 7.6.6 vorgestellte spezifikationsspezifische Filteroperation `my:tr-filtered` simuliert. Sie liefert nur die Message-Elemente aus dem Trace nach einem Aufruf der Operation `UnlockAll`, die alle Sperren freigibt und damit Startzustand herstellt. Durch Verwendung der Filteroperation `my:tr-filtered` wird der SXQT-Ausdruck in Abbildung 113 vom benötigten Startzustand unabhängig.

7.7.6 Klassen CD und SD

Anforderungen der Klassen CD und SD setzen einen Wert in einer SOAP-Nachricht mit einer Menge von gültigen Werten in Beziehung. Ist ihr SOAP-Sender ein Webclient, gehört die Anforderung zur Klasse CD. Ist es der Webservice, gehört sie zur Klasse SD. Welche Werte gültig sind, wird mit einer Operation ermittelt, die eine Sequenz aller gültigen Werte liefert. Diese Operation, die im Folgenden als Werteoperation bezeichnete werden soll, muss jedoch nie aufgerufen werden. Schon durch die Tatsache, dass die Werteoperation einen Wert liefern würde, wenn sie den aufgerufen werden würde, reicht, damit der Wert in der SOAP-Nachricht gültig ist.

Bereits im Kapitel 6.2.3 wurde festgestellt, dass solche Anforderungen nicht als Beziehung zwischen SOAP-Nachrichten beschrieben werden können. Da die Werteoperation nie aufgerufen worden sein muss, muss ihr Response, der die Sequenz der gültigen Werte enthält, auch nicht beobachtet worden sein. Er muss sich also auch nicht im Trace befinden. Könnte das dagegen sichergestellt werden, könnte mit einem SXQT-Ausdruck der Klasse CC bzw. SC der Wert in einer SOAP-Nachricht mit der Menge der gültigen Werte im Response der Werteoperation in Beziehung gesetzt werden. Da der Aufruf der Werteoperation jedoch optional ist, würde ein solcher SXQT-Ausdruck etwas anders beschreiben als die Anforderung der Klasse CD bzw. SD.

Damit lassen sich Anforderungen der Klassen CD und SD mit SXQT, dem Spezifikationsverfahren dieser Arbeit, streng genommen nicht formulieren. Nachfolgend sollen jedoch zwei Ansätze diskutiert werden, wie mit solchen Anforderungen umgegangen werden könnte.

Der erste Ansatz greift die obige Idee wieder auf, Anforderungen der Klasse CD oder SD als Anforderungen der Klasse CC bzw. SC zu formulieren. Hierzu müssen die Anforderungen geeignet modifiziert werden. Eine solche Anforderung müsste fordern, dass sich der zu prüfende Wert in der SOAP-Nachricht im Response der Werteoperation befindet, wenn es einen solchen Response überhaupt im Trace gibt. Der SXQT-Ausdruck für die Anforderung müsste auch dann den logischen Wert `true` liefern, wenn sich ein solcher Response nicht im Trace befindet.

Diese Idee soll am Beispiel einer Anforderung der Klasse CD aus Kapitel 6.2.3 veranschaulicht werden. Dort wurde in Abbildung 62 auf Seite 107 ein Request der Operation `ChangeCategory` der Webkomponente für eine Internetzeitung gezeigt. Ein solcher Request enthält im Bodyeintrag im Element `nwst:Category` eine Rubrik der Internetzeitung. Der Wert ist nur gültig, wenn ihn die Werteoperation `Categories` im Response liefert, ebenfalls in einem Element `nwst:Category`. Ein solcher Response wurde bereits in Abbildung 64 auf Seite 108 gezeigt.

```
every $m
  in opr:restrict(opr:tr(),
                  fn:QName-in-context("nwst:ChangeCategory",
                                      fn:false()))
  satisfies
    let $seq
      := opr:restrict
           (opr:tr(),
            fn:QName-in-context("nwst:CategoriesResponse",
                                fn:false()))
    return
      fn:empty($seq)
      or opr:event-body-entry(opr:head(opr:reverse($seq)))
               /nwst:CategoriesSequence/nwst:Category
         = opr:event-body-entry($m)/nwst:Category
```

Abbildung 119: Anforderung der Klasse CD behandelt wie in Klasse CC als SXQT-Ausdruck

Abbildung 119 zeigt, wie sich mit den genannten Annahmen die Anforderung als SXQT-Ausdruck der Klasse CC formulieren lässt. Er prüft mit einem Allquantor für jedes Message-Element eines Requests der Operation `ChangeCategory`, ob die Anforderung zutrifft. Für jedes dieser Message-Elemente wird der Trace erneut auf alle Message-Elemente mit Responses der Werteoperation `Categories` beschränkt und das Ergebnis an die Variable `$seq` gebunden. Wurde die Operation `Categories` nie aufgerufen, ist die Sequenz in `$seq` leer. Dann sei die Anforderung erfüllt und es muss der logische Wert `true` geliefert werden. Diese Prüfung findet im Teilausdruck `fn:empty($seq)` statt. Nur wenn die Sequenz nicht leer ist, muss die Rubrik im Request der Operation `ChangeCategory` im Response des letzten Aufrufes der Operation `Categories` enthalten sein, was am Ende des Ausdrucks geprüft wird.

Der eben beschriebene, erste Ansatz erforderte, dass Anforderungen in solche der Klasse CC bzw. SC umformuliert werden. Mit dem zweiten nun beschriebenen Ansatz ist das nicht erforderlich. Mit ihm können direkt Anforderungen der Klasse CD und SD als SXQT-Ausdrücke formuliert werden. Dazu ist es aber erforderlich, dass die Werteoperation statt mit dem Request-/Response-Kommunikationsmuster mit dem SOAP-/Response-Kommunikationsmuster realisiert werden, das Kapitel 3.8 beschrieben wurde. Durch Einsatz der Standardfunktion `fn:document` können dann im SXQT-Ausdruck die Werteoperation direkt aufgerufen werden.

Grundsätzlich wird die in [Boag 02], [Malhotra 02] und Kapitel 7.4.5 beschriebene Standardfunktion `fn:document` verwendet, um auf mit URIs bezeichnete XML-Dokumente oder XML-Fragmente zu verweisen. So können deren Inhalte in XQuery-Ausdrücken berücksichtigt werden. Im einfachsten Fall wird der Funktion eine Zeichenkette als Parameter übergeben, die die URI eines XML-Dokumentes enthält. Die Funktion liefert dann den Dokumentknoten des bezeichneten XML-Dokumentes.

Handelt es sich bei der angegebenen URI um eine URL, die mit `http:` beginnt, wird das HTTP-Protokoll verwendet, um das XML-Dokument von einem Webserver zu kopieren. Dabei wird die Webmethode `GET` verwendet, die bereits in Kapitel 3.10 vorgestellt wurde. Im Request wird die URI (genauer: ein Teil von ihr) übertragen und im Response das XML-Dokument.

Solche Zugriffe auf XML-Dokumente entsprechen Aufrufen von Operationen nach dem SOAP-Response-Kommunikationsmuster, wenn als Protokollbindung die HTTP-Bindung verwendet wird. (Siehe Kapitel 3.8 und 3.10.) Die Operationen können daher in XQuery mit der Funktion `fn:document` aufgerufen werden. Die im Response zurückgelieferten SOAP-Nachrichten sind XML-Dokument und können damit in XQuery verarbeitet werden.

Das SOAP-Response-Kommunikationsmuster darf nur verwendet werden, wenn die Operation seiteneffektfrei ist. Bei den Werteoperationen, wie der oben betrachteten Operation `Categories`, mit denen gültige Werte abgefragt werden, ist das jedoch der Fall. Daher können solche mit dem SOAP-Response-Kommunikationsmuster realisiert werden.

Für einen Präzisierungsstandard und diese Arbeit wurde in Kapitel 5.2.1 jedoch festgelegt, dass das SOAP-Response-Kommunikationsmuster nicht verwendet werden soll, sondern nur das Request-/Response-Kommunikationsmuster. Damit sollte vermieden werden, dass für SOAP mehr Optionen als notwendig zugelassen werden. Das Request-/Response-Kommunikationsmuster ist universeller. Mit ihm können alle Operation realisiert werden, die auch mit dem SOAP-Response-Kommunikationsmuster realisiert werden können. Umgekehrt ist das nicht der Fall.

Leider lassen sich Operationen, die mit dem Request-/Response-Kommunikationsmuster realisiert sind, in XQuery nicht mit der Funktion `fn:document` aufrufen. Die Funktion erlaubt nicht die Verwendung der Webmethode `POST`, die stets mit dem Request-/Response-Kommunikationsmodell verwendet wird. Außerdem gibt es keine Möglichkeit, den benötigten Request als SOAP-Nachricht zu übergeben. Soll daher aus einem SXQT-Ausdruck eine Werteoperation aufgerufen werden, muss diese mit dem SOAP-Response-Kommunikationsmuster realisiert sein.

Soll die bereits oben betrachtete Anforderung der Klasse CD nach diesem Ansatz als SXQT-Ausdruck formuliert werden, muss die Operation `Categories` also mit dem SOAP-Response-Kommunikationsmuster realisiert werden. Sie habe die URI `http://www.newsservice.example.org/Categories`. Wird diese URI im Request von HTTP an den Webservice geschickt, antwortet dieser mit dem Response der Operation `Categories`, der bereits im in Abbildung 64 auf Seite 108 abgebildet wurde.

Um die Anforderung auszudrücken, müssen die in dieser SOAP-Nachricht enthaltenen, erlaubten Rubriken mit den Rubriken in Beziehung gesetzt werden, die in Requests der Operation `ChangeCategory` im Element `nwst:Category` angegeben sind. Ein Beispiel für einen solchen Request wurde bereits in Abbildung 62 auf Seite 107 gezeigt. Die Beziehung lässt sich als SXQT-Ausdruck formulieren, wie in Abbildung 120 gezeigt.

```
every $m
  in opr:restrict(opr:tr(),
                 fn:QName-in-context("nwst:ChangeCategory",
                                     fn:false()))
  satisfies
    opr:event-body-entry($m)/nwst:Category
    = fn:document("http://www.newsservice.example.org/Categories")
      /env:Envelope/env:Body/nwst:CategoriesResponse
      /nwst:CategoriesSequence/nwst:Category
```

Abbildung 120: Anforderung der Klasse CD als SXQT-Ausdruck mit Funktion `fn:document`

Zunächst wird der Trace auf alle Message-Elemente mit Requests der Operation `ChangeCategory` beschränkt und mit einem Allquantor für jedes die Anforderung geprüft. Für jedes wird ähnlich aus jedem Request das Element `nwst:Category` ermittelt, wobei die Funktion `opr:event-body-entry` verwendet wird. Neu ist, dass mit der Funktion `fn:document` die Operation `Categories` aufgerufen wird. Der von ihr gelieferte Response wird im SXQT-Ausdruck verarbeitet. Mit einem Pfadausdruck wird aus ihm die Sequenz der Elemente `nwst:Category` ermittelt, die die erlaubten Rubriken enthalten. Mit der Operation `=` wird danach geprüft, ob eine dieser erlaubten Rubriken gerade die im Request der Operation `ChangeCategory` ist.

Offenbar ist es also möglich, Anforderungen der Klassen CD und SD mit Hilfe der Funktion `fn:document` zu formulieren, wenn die Werteoperation mit dem SOAP-Response-Kommunikationsmuster realisiert ist. Dazu muss die Werteoperation seiteneffektfrei sein, was in der Regel der Fall ist.

Ein Nachteil dieses Ansatzes ist zunächst, dass das SOAP-Response-Kommunikationsmuster verwendet werden muss. Diese sollte nach Kapitel 5.2.1 nicht verwendet werden.

Darüber hinaus ändert die Verwendung der Funktion `fn:document` die Semantik von Spezifikationen wesentlich. Ohne sie kann zunächst ein Beobachter nur Ereignisse beobachten und als Trace aufzeichnen. Danach kann für den Trace entschieden werden, ob er der Spezifikation entspricht oder nicht. Dabei sind neben dem Trace und der Spezifikation keine weiteren Informationen nötig. Wird in einem SXQT-Ausdruck jedoch die Funktion `fn:document` verwendet, wird bei dessen Auswertung zusätzlich „noch schnell“ eine Operation des Webservices aufgerufen, um noch benötigte Informationen zu ermitteln, nämlich die Menge der gültigen Werte.

Abgesehen von der Änderung der Semantik der Spezifikation, muss auch die Frage gestellt werden, ob die Menge der gültigen Werte überhaupt zum richtigen Zeitpunkt ermittelt wird. Richtig wäre der Zeitpunkt der Beobachtung der SOAP-Nachricht, die den Wert enthält, dessen Gültigkeit geprüft werden soll. Mit der Funktion `fn:document` wird die Menge jedoch bei der Auswertung des SXQT-Ausdrucks ermittelt. Dazu müsste vorausgesetzt werden, dass sich die Menge der erlaubten Werte seit der Beobachtung der SOAP-Nachricht nicht geändert hat. Ansonsten könnte es sogar geschehen, dass die Auswertung eines SXQT-Ausdrucks für denselben Trace zu unterschiedlichen Zeitpunkten zu unterschiedlichen Ergebnissen führt.

Schließlich muss auch gefragt werden, wozu SXQT-Ausdrücke überhaupt verwendet werden sollen. Wird die Funktion `fn:document` verwendet, ist es möglich, sie auszuwerten. Sollen SXQT-Ausdrücke aber anders verwendet werden, z. B. um Eigenschaften eines Webservices zu beweisen, ist die Semantik der Funktion `fn:document` erneut zu klären.

Aus diesen Argumenten folgt, dass in SXQT-Ausdrücken für Anforderungen der Klassen CD und SD die Verwendung der Funktion `fn:document` sowie das Request-/Response-Kommunikationsmuster für die Werteoperation problematisch ist. Sie ist bestenfalls als Notbehelf zu betrachten, um Anforderungen der Klassen CD und SD überhaupt formulieren zu können. Als zweckmäßige Vorgehensweise soll sie in dieser Arbeit jedoch nicht betrachtet werden. Daher sei wie bereits am Anfang dieses Kapitels 7.7.6 festgestellt, dass sich Anforderungen der Klassen CD und SD mit SXQT nicht formulieren lassen.

8 Ablage von SXQT-Ausdrücken

Die in Kapitel 7.7 gezeigten Beispiele haben veranschaulicht, wie Anforderungen als SXQT-Ausdrücke formuliert werden können. Es wurde stets eine Anforderungen untersucht und für sie ein SXQT-Ausdruck angegeben. Jeder SXQT-Ausdruck einzeln beschreibt also nur einen Aspekt der Schnittstelle. Erst eine Menge von ihnen bildet zusammen mit einem WSDL-Dokument die vollständige SXQT-Spezifikation, wie in Kapitel 7.3.1 festgestellt.

Die vollständige SXQT-Spezifikation muss Entwicklern zur Verfügung gestellt werden, damit diese Webclients bzw. Webservices ihr entsprechend implementieren können. Bei WSDL wird das typischerweise erreicht, indem der Webservice ein WSDL-Dokument im Internet bereitstellt und es der Webclient von ihm abfragen kann. Dieser verwendet dazu in der Regel das Protokoll HTTP, um benötigte WSDL-Dokumente zu kopieren. Da der Webservice ohnehin das WSDL-Dokument verbreitet und damit die SXQT-Spezifikation als Einheit angesehen werden kann, sollte die SXQT-Spezifikation vollständig vom Webservice zur Verfügung gestellt werden.

Es stellt sich die Frage, wie die SXQT-Spezifikation in einer Form gespeichert werden kann, dass eine Bereitstellung durch den Webservice möglich ist. Dazu muss insbesondere untersucht werden, wo und wie die SXQT-Ausdrücke ergänzend zur Beschreibung in WSDL abgelegt werden können. Unterschiedliche Möglichkeiten werden in diesem Kapitel diskutiert.

Die Untersuchung wird so angelegt, dass sie auch auf andere Spezifikationsverfahren übertragbar ist. Dazu werden Details von SXQT zunächst nicht betrachtet. Es wird nur angenommen, dass es eine Menge von Zusätzen gibt, die in textueller Form kodierbar sind und die die Schnittstellenbeschreibung des Webservice in WSDL ergänzen. Welche Bedeutung ein einzelner Zusatz hat und in welcher Form er notiert wird, spielt keine Rolle. Erst nach dieser generellen Untersuchung sollen deren Ergebnisse in Kapitel 8.3 auf SXQT angewendet werden. Die Zusätze sind dann SXQT-Ausdrücke.

8.1 Mögliche Ablageorte für Anforderungen

8.1.1 Eigenen Dateien

Wird von den SXQT-Ausdrücken in den Beispielen des Kapitels 7.7 ausgegangen, mit denen jeweils eine Anforderung formuliert wurde, ist es eine naheliegende Idee einen oder mehrere SXQT-Ausdrücke gemeinsam in einer Datei abzulegen. Solche Dateien könnten später direkt zur Auswertung an einen XQuery-Prozessor übergeben werden.

Das lässt sich auf andere Spezifikationsverfahren übertragen. Auch ihre Zusätze können einzeln oder gemeinsam mit andern in Dateien abgelegt werden. Wird für jeden Zusatz eine Datei verwendet, besteht die Spezifikation aus einer Vielzahl solcher Dateien zusätzlich zum WSDL-Dokument. Alternativ ließen sich alle Zusätze zu einer Datei zusammenfassen. Dann würde nur diese das WSDL-Dokument ergänzen.

Im letzteren Fall stellt sich die Frage, wie sich die Zusätze zusammenfassen lassen. Hierzu müsste ein geeignetes Dateiformat definiert werden. Lassen sich alle Zusätze einzeln als XML-Dokumente formulieren, können sie leicht zu einem XML-Dokument zusammengefasst werden, das sie als XML-Fragmente enthält.

Handelt es sich bei den Zusätzen, wie bei SXQT-Ausdrücken, um logische Ausdrücke, die alle für das Erfüllen der Spezifikation zutreffen müssen, gibt es einen anderen Weg sie zusammenzufassen. Sie können zu einem logischen Ausdruck zusammengefasst werden, indem von ihnen die Konjunktion gebildet wird. Dann liefert der zusammengefasste Ausdruck genau dann den Wert `true`, wenn alle konjugierten Ausdrücke den Wert `true` liefern.

Zusätze separat vom WSDL-Dokument in Dateien zu speichern, hat den Vorteil, dass das WSDL-Dokument unverändert bleibt. Werkzeuge können es ohne Kenntnis der Zusätze weiterhin interpretieren. Die Zusätze werden einfach ignoriert. Auf der anderen Seite muss trotz der Verteilung der Spezifikation auf Dateien und WSDL-Dokument sichergestellt werden können, dass die Spezifikation konsistent ist und auch bei Änderungen bleibt. Wird das WSDL-Dokument geändert, muss auf jeden Fall geprüft werden, dass auch die Zusätze weiterhin gültig sind.

Darüber hinaus muss sichergestellt werden, dass die Spezifikation als ein Ganzes wahrgenommen werden kann, obwohl sie auf mehrere Dateien verteilt ist. Das ist der Fall, wenn sie sich mit einer URI bezeichnen lässt, wie es mit WSDL-Dokumenten geschieht. Die URI eines WSDL-Dokumentes kann auch verwendet werden, um seinen Inhalt zu ermitteln. Die vollständige Spezifikation ließe sich mit ihrer URI ermitteln, wenn diese ein Dokument bezeichnet, dass seinerseits auf das WSDL-Dokument und alle Dateien der Zusätze verweist. Das Dokument könnte ein XML-Dokument sein. In jedem Fall müsste ein geeignetes Dateiformat definiert werden.

8.1.2 WSDL-Dokument

Alternativ zur Speicherung der Zusätze in eigenen Dateien kann auch WSDL so erweitert werden, dass die Zusätze direkt im WSDL-Dokument gespeichert werden. Da dann die gesamte Spezifikation im WSDL-Dokument enthalten ist, wird sie als ein Ganzes wahrgenommen. Sie wird durch die URI des WSDL-Dokumentes bezeichnet. Gleichzeitig wird dadurch vereinfacht, die Konsistenz der Spezifikation langfristig sicherzustellen, weil bei Änderungen offensichtlich ist, welche Teile zur Spezifikation gehören.

Nachteilig erscheint, dass für diesen Ansatz die Sprache WSDL erweitert wird. Das birgt die Gefahr, dass Werkzeuge, die WSDL-Dokumente lesen, diese mit unbekannten Erweiterungen nicht interpretieren können. Es ist jedoch möglich, WSDL so zu erweitern, dass Werkzeuge, die WSDL 1.2 korrekt interpretieren, gezielt solche Erweiterungen ignorieren können. Hierzu stellt WSDL 1.2 Mechanismen zu seiner Erweiterung zur Verfügung, die im 4. Kapitel des WSDL-Standards [Chinnici 02] vorgestellt werden.

Die Mechanismen beruhen auf der Tatsache, dass sich im WSDL-Standard definierte Elemente im Namensraum von WSDL (`http://www.w3.org/2002/07/wsdl`) befinden und verwendete Attribute im gleichen Namensraum oder gar keinem. Den Elementen dürfen beliebige Kindelemente und Attribute in anderen Namensräumen zugefügt werden. Leser des WSDL-Dokumentes können diese sogenannten Erweiterungselemente bzw. Erweiterungsattribute daran erkennen und dürfen sie ignorieren. So können sie es auch dann korrekt interpretieren, wenn er Erweiterungen nicht kennen.

Bei einem Erweiterungselement kann jedoch deklariert werden, dass es nicht ignoriert werden darf. Hierzu wird ihm das Attribut `wsdl:required` mit dem Wert `true` zuge-

fügt. Damit wird festgelegt, dass ein Leser das Erweiterungselement nicht ignorieren darf, wenn er dessen Elternelement aus dem Namensraum von WSDL verarbeitet. Verarbeitet er das Elternelement nicht, darf er das Erweiterungselement trotzdem ignorieren.

Erweiterungselemente und -attribute können verwendet werden, um Zusätze eines Spezifikationsverfahrens für Webservices im WSDL-Dokument zu speichern. Dazu muss zunächst entschieden werden, ob Erweiterungselemente oder -attribute verwendet werden sollen. Erweiterungsattribute haben den Vorteil, dass ihre Schreibweise häufig kompakter ist. Sie können jedoch nur Zeichenketten aufnehmen. Diese sollten kurz sein, damit die Notation übersichtlich bleibt. Zweckmäßig ist es zumindest, dass Werte im Attribut einzeilig sind.

Erweiterungselemente haben dagegen den Vorteil, dass sie große, auch mehrzeilige Zeichenketten enthalten können. Sie können auch selbst Kindelemente und Attribute haben. Werden für die Zusätze des Spezifikationsverfahrens Attribute oder Kindelemente benötigt oder sind es lange, mehrzeilige Zeichenketten, sollten also Erweiterungselemente verwendet werden. Sind es dagegen kurze Zeichenketten ist auch die Verwendung von Erweiterungsattributen möglich.

Werden Erweiterungselemente verwendet, muss entschieden werden, ob das Attribut `wsdl:required` mit dem Wert `true` zugefügt werden soll. Das wäre nur zweckmäßig, wenn eine Interpretation des Elternelementes ohne Kenntnis der Erweiterung nicht korrekt möglich ist. Da Leser ein WSDL-Dokument jedoch auch ohne die Zusätze des Spezifikationsverfahrens sinnvoll verwenden können, sollte das Attribut `wsdl:required` mit dem Wert `true` nicht zugefügt werden.

8.1.3 XML-Schemadokument

Zusätzen von Spezifikationen können auch in XML-Schema eingebettet werden. Wie in Kapitel 4.3.3 vorgestellt, enthält ein WSDL-Dokument ein XML-Schemadokument oder es verweist auf ein solches. Damit würden Zusätze im XML-Schemadokument auch automatisch Teil der Beschreibung des Webservices. Die Spezifikation würde wie bei der Einbettung der Zusätze in WSDL als ein Ganzes wahrgenommen und durch die URI des WSDL-Dokumentes bezeichnet. Ebenso ließe sich die Konsistenz der Spezifikation ähnlich gut langfristig sicherstellen.

Wieder bestünde die Gefahr, dass Leser erweiterte XML-Schemadokumente nicht interpretieren können. XML-Schema stellt aber wie WSDL Mechanismen zu seiner Erweiterung zur Verfügung, die es erlaubten, zusätzliche Attribute und Elemente hinzuzufügen, die nicht im Standard von XML-Schema definiert sind.

Das Zufügen von Attributen geschieht ähnlich wie bei WSDL. Die Elemente von XML-Schema befinden sich in dessen Namensraum `http://www.w3.org/2001/XMLSchema` und seine Attribute in keinem oder dem gleichen. Den meisten dieser Elemente können zusätzliche Erweiterungsattribute in einem anderen Namensraum zugefügt werden, wie im Standard von XML-Schema [Thompson 01] festgestellt wird[54].

Erweiterungselemente werden jedoch anders behandelt als in WSDL. In XML-Schema ist es nicht erlaubt, Elementen direkt Erweiterungselemente als Kindelemente zuzufügen. Sie dürfen nur bestimmten, dafür vorgesehen Elementen zugefügt werden. Hierzu

[54] In [Thompson 01] wird das im Text in Kapitel 3.13.1 festgestellt. Dabei ist dort missverständlich, auf welche Elemente es sich bezieht. Klar folgt die Feststellung jedoch aus dem XML-Schemadokument für XML-Schema in Anhang A.

haben die meisten Elemente von XML-Schema optional das erste Kindelement `xsd:annotation`. Dieses kann wiederum die Kindelemente `xsd:documentation` und `xsd:appinfo` beliebig oft und in beliebiger Reihenfolge haben.

Diese beiden Elemente dürfen einen beliebigen Inhalt enthalten, also Text und auch Kindelemente und damit auch beliebige Erweiterungselemente. Das Element `xsd:documentation` ist dabei für Zusatzinformationen gedacht, die für menschliche Leser bestimmt sind. Die Zusatzinformationen im Element `xsd:appinfo` sind dagegen für maschinelle Verarbeitung bestimmt.

Abbildung 121 zeigt einen Ausschnitt aus einem XML-Schemadokument mit Erweiterungselementen. Es stammt aus dem Standard von XML-Schema [Brion 01] und enthält die Definition des Datentyps `xsd:boolean`. Im Element `xsd:appinfo` werden mit Erweiterungselementen Eigenschaften dieses Typs deklariert, die sich in XML-Schema sonst nicht ausdrücken lassen. Analog könnten auch Zusätze von Spezifikationen in XML-Schema als Erweiterungselemente zugefügt werden. Sind diese für eine maschinelle Verarbeitung bestimmt, sollte das im Element `xsd:appinfo` geschehen.

```
<xsd:simpleType name="boolean" id="boolean">
  <xsd:annotation>
    <xsd:appinfo>
      <hfp:hasFacet name="pattern"/>
      <hfp:hasFacet name="whiteSpace"/>
      <hfp:hasProperty name="ordered" value="false"/>
      <hfp:hasProperty name="bounded" value="false"/>
      <hfp:hasProperty name="cardinality" value="finite"/>
      <hfp:hasProperty name="numeric" value="false"/>
    </xsd:appinfo>
    <xsd:documentation
      source="http://www.w3.org/TR/xmlschema-2/#boolean"/>
  </xsd:annotation>
  <xsd:restriction base="xsd:anySimpleType">
    <xsd:whiteSpace value="collapse" fixed="true"
      id="boolean.whiteSpace"/>
  </xsd:restriction>
</xsd:simpleType>
```

Abbildung 121: Definition des Typs `xsd:boolean` mit Element `xsd:appinfo` aus [Brion 01][55]

Damit sind auch in XML-Schema sowohl Erweiterungselemente als auch Erweiterungsattribute möglich. Die Argumente, ob Erweiterungselemente oder -attribute verwendet werden sollten, sind die gleichen wie bei WSDL und wurden in Kapitel 8.1.2 diskutiert. Bei XML-Schema gibt es jedoch kein Analogon zum Attribut `wsdl:required`, für das in Kapitel 8.1.2 aber ohnehin festgestellt wurde, dass es für Zusätze von Spezifikationsverfahren nicht verwendet werden sollte.

8.1.4 Vergleich

Damit stehen drei grundsätzlich unterschiedliche Möglichkeiten zur Verfügung, die Zusätze von Spezifikationsverfahren abzulegen. Sie können in eigenen Dateien oder als Erweiterungselemente oder -attribute in WSDL oder XML-Schema abgelegt werden.

---

[55] Das gezeigte Beispiel ist ein Ausschnitt aus dem XML-Schemadokument für Datentypendefinitionen aus Anhang A von [Brion 01]. Dort wird statt dem Präfix `xsd` jedoch das Präfix `xs` verwendet. Für die Konsistenz dieser Arbeit wurde das Präfix angepasst. Das Präfix `hfp` steht für den Namensraum `http://www.w3.org/2001/XMLSchema-hasFacetAndProperty`.

Für die Verwendung eigener Dateien spricht lediglich, dass die WSDL-Dokumente unverändert bleiben, einschließlich enthaltener Beschreibungen in XML-Schema. Leser der WSDL-Dokumente können diese auf jeden Fall weiter interpretieren. Das ist allerdings auch möglich, wenn WSDL oder XML-Schema mit Erweiterungselementen oder -attributen erweitert wird und Leser solche Erweiterungen ignorieren, wie in den Standards von WSDL und XML-Schema vorgeschrieben.

Dann ergibt sich der Vorteil, dass sich die Spezifikation als ganzes im WSDL-Dokument befindet. Nur wenn es gewünscht ist, kann sie mit den Mechanismen von WSDL oder XML-Schema auf mehrere Dokumente aufgeteilt werden. Da die Spezifikation als Ganzes gesehen wird, kann sie leichter konsistent geändert werden, weil die Vorgehensweise das „Vergessen“ einzelner Teilen verhindert.

Daher sollten für Zusätze von Spezifikationen Erweiterungselemente oder -attribute in WSDL oder XML-Schema verwendet werden. Ob Erweiterungselemente oder -attribute zweckmäßiger sind, hängt von der Art ab, wie umfangreich die Zusätze sind und wie sie kodiert werden. Für kleine Zusätze, kodiert als einzeilige, kurze Zeichenketten, sind Erweiterungsattribute zweckmäßig. Dann kann die Darstellung kompakter und übersichtlicher sein. Sind die Zeichenketten dagegen lang und mehrzeilig oder wird der Zusatz selbst als XML-Fragment mit Kindelementen kodiert, müssen Erweiterungselemente verwendet werden.

8.2 Zweckmäßige Ablageorte in WSDL und XML-Schema

Es bleibt die Frage, ob WSDL oder XML-Schema erweitert werden sollte. Das hängt davon ab, auf welchen Teil der Schnittstelle eines Webservices sich die Zusätze des Spezifikationsverfahrens beziehen. Mit der Präzisierung von WSDL in Kapitel 5.3 wird XML-Schema in WSDL verwendet, um Body-, Header- und Detaileinträge von Fehlernachrichten einschließlich ihrer Nachfahren zu beschreiben. In WSDL selbst wird dagegen festgelegt, wie diese XML-Fragmente zu SOAP-Nachrichten, die SOAP-Nachrichten zu Request-/Response-Paaren und diese wiederum zu Schnittstellen zusammengefasst werden. Sowohl XML-Schema als auch WSDL stellen zur Beschreibung jedes dieser Teile spezielle Elemente bereit.

Ein Zusatz einer Spezifikation der sich auf einen dieser Teile bezieht, sollte mit dem entsprechenden Element von XML-Schema bzw. WSDL gespeichert werden. Bezieht sich ein Zusatz z. B. auf ein einzelnes XML-Element in einer SOAP-Nachricht, ist es sinnvoll ihn mit der Deklaration dieses XML-Elementes im XML-Schemadokument oder mit der Definition des zugehörigen Typs zu speichern. Bezieht sich andererseits ein Zusatz z. B. auf die Schnittstelle als ganzes, sollte sie mit der Beschreibung der Schnittstelle in WSDL abgelegt werden.

Welche Elemente für das Zufügen von Zusätzen in WSDL und XML-Schema zur Verfügung stehen und wann diesen ein Zusatz zugefügt werden sollte, wird nachfolgend genauer untersucht. Die Hierarchie eines WSDL-Dokumentes einschließlich der von XML-Schema besteht aus einer Vielzahl von Elementen. Zur Veranschaulichung ist sie in Abbildung 122 gezeigt. Dazu muss beachtet werden, dass Elemente in der Hierarchie z. T. fehlen oder mehrfach vorkommen dürfen. In Abbildung 122 wurden außerdem die Erweiterungselemente der SOAP-Bindung von SOAP 1.2 zugefügt. Sie sind am Präfix `soap12` zu erkennen. Schließlich wurde die Darstellung von XML-Schema auf vier wesentliche Elemente beschränkt und auch deren Kindelemente nicht mit gezeigt. Eine vollständige Darstellung von XML-Schema ist wegen seiner rekursiver Strukturen in dieser Form nicht möglich.

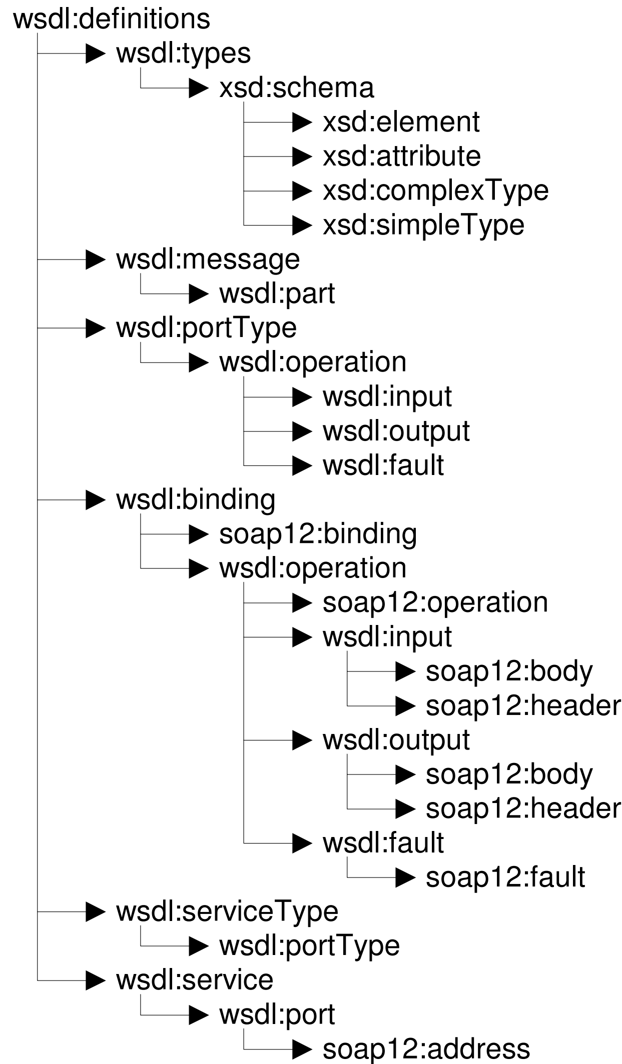

Abbildung 122: Hierarchie der Elemente in WSDL-Dokumenten

8.2.1 XML-Element oder Attribut

Im ersten, eben gegebenen Beispiel wurde festgestellt, dass sich ein Zusatz auf ein XML-Element beziehen kann und er dann der Deklaration eines XML-Elementes oder dessen Typ zugeordnet wird. Hier soll zunächst der Fall betrachtet werden, dass der Zusatz der Deklaration des XML-Elementes zugeordnet wird.

Tatsächlich beziehen sich Zuätze in der Regeln nicht nur auf ein einzelnes XML-Element. Damit ein Zusatz etwas miteinander in Beziehung setzen kann, muss er sich zumindest noch auf dessen Attribute beziehen. Analog zu den Anforderungen in den Kapiteln 6.1.1, 6.2.1 und 6.2.2 werden Zusätze häufig neben Attributen auch Kindelemente und andere Nachfahren miteinander in Beziehung setzen.

Es wäre sogar möglich, dass ein Zusatz Vorfahren des XML-Elementes und deren Nachfahren mit betrachtet. Dann könnte der Zusatz aber ebenso gut dem Vorfahren zugeordnet werden. Die Zuordnung ist also nicht unbedingt eindeutig. Gegebenenfalls muss eines der beteiligten XML-Elemente gewählt werden. Gibt es ein XML-Element, dass offensichtlich für den Zusatz eine herausragende Rolle spielt, sollte dieses verwendet werden. Ansonsten erscheint es naheliegend das XML-Element zu verwenden, das in der Hierarchie am höchsten steht, so dass es der Zusatz mit seinen Nachfahren in Beziehung gesetzt.

Der Deklaration dieses XML-Elementes sollte der Zusatz als Erweiterungselement oder -attribut zugefügt werden. Das ist unabhängig davon möglich, ob die Deklarationen lokal oder global ist. Lokale und globale Deklarationen wurden im Kapitel 4.2.3 vorgestellt. In beiden Fällen bezieht sich der Zusatz zu einer Deklaration auf alle XML-Elemente im Instanzdokument, die mit der Deklaration beschrieben werden.

Bezieht sich ein Zusatz auf ein Attribut, sollte der Zusatz dessen Deklaration mit `xsd:attribute` zugefügt werden. Attribute haben lediglich einen qualifizierten Namen und einen Wert. Da sie keine Nachfahren haben, kann sich der Zusatz auch nicht auf solche beziehen. Möglich wäre es höchstens, dass sich der Zusatz auf Vorfahrenelemente bezieht, was nur in Ausnahmefällen sinnvoll erscheint.

8.2.2 Typ

Sowohl XML-Elemente als auch Attribute haben Typen, die bei ihrer Deklaration angegeben werden. Soll sich ein Zusatz auf alle XML-Elemente bzw. Attribute eines Typs beziehen, ist es zweckmäßig ihn der Definition des Typs zuzuordnen. Wie in den Kapiteln 4.2.5 und 4.2.6 vorgestellt, gibt es in XML-Schema zur Definition von Typen zwei Elemente: `xsd:complexType` und `xsd:simpleType`.

Das Element `xsd:complexType` wird verwendet, um komplexe Typen zu definieren. Komplexe Typen können nur für die Deklaration von XML-Elementen verwendet werden. Mit ihnen kann festgelegt werden, welche Kindelemente und Attribute ein XML-Element des Typs haben kann. Der Zusatz kann sich daher wie bei der Deklaration eines XML-Elementes auch auf Attribute und Nachfahren beziehen sowie in Ausnahmefällen auf Vorfahren.

Das Element `xsd:simpleType` wird dagegen für die Definition von einfachen Typen verwendet, die letztlich nur eine Menge von Zeichenketten beschreiben. Solche Typen können sowohl für die Deklaration von XML-Elementen als auch von Attributen verwendet werden. Diese können dann weder Kindelemente noch Attribute haben. Daher kann sich der Zusatz nicht auf diese beziehen. Möglich wäre lediglich eine Beziehung mit Vorfahren-Elementen. Die Definition eines einfachen Typen in Abbildung 121 ist ein Beispiel für einen einfachen Typen, dessen Definition Zusätze als Erweiterungselemente zugefügt wurden.

Typen können anonym oder benannt sein, wie in Kapitel 4.2.3 beschrieben. Ein Zusatz, der einem anonymen Typen zugefügt wird, bezieht sich nur auf die Deklaration des XML-Elementes bzw. Attributes, der er zugefügt wurde. Daher sollte er eher der Deklaration selbst zugefügt werden. Bei benannten Typen ist das anders, weil auf sie in mehreren Deklarationen von XML-Elementen oder Attributen verwiesen werden kann. Ein Zusatz ist dann für alle Deklarationen gültig, die auf den Typ mit dem Zusatz verweisen.

8.2.3 Andere Bezüge zu XML-Schema

Neben den Elementen `xsd:element` und `xsd:attribute` zur Deklaration von XML-Elementen und Attributen und den Elementen `xsd:complexType` und `xsd:simpleType` zur Definition von Typen stellt XML-Schema weitere Elemente bereit. Sie werden Deklarationen von Elementen als zusätzliche Einschränkungen beigefügt oder beschreiben einen Aspekt eines neu definierten Typen. In beiden Fällen kann es interessant sein, auch ihnen Zusätze zuzufügen, wenn sich ein Zusatz auf das durch sie beschriebene bezieht.

So kann z. B. in einer Deklaration eines XML-Elementes mit dem Element `xsd:unique` eine Eindeutigkeitsbeziehung ausgedrückt werden. Sie beschreibt, dass Werte innerhalb des XML-Elementes in einem bestimmten Kontext eindeutig sein müssen. Die Werte können in Attributen oder Nachfahren enthalten sein. Ähnlich dazu kann mit den Elementen `xsd:key` und `xsd:keyref` eine referenzielle Integrität im XML-Dokument ausgedrückt werden, mit der sichergestellt werden kann, dass Verweise in XML-Dokumenten stets gültig sein müssen. Details zu beidem finden sich in [Fallside 01], [Thompson 01] und [Skonnard 02]. Soll eine Eindeutigkeitsbeziehung oder eine Beziehung zur referenziellen Integrität durch einen Zusatz eines Spezifikationsverfahrens genauer beschrieben werden, sollte der Zusatz den Elementen `xsd:unique`, `xsd:key` bzw. `xsd:keyref` zugefügt werden.

Gleiches gilt auch für die Elemente mit denen Typen definiert werden. So wird mit den Elementen `xsd:sequence`, `xsd:choice` und `xsd:all` festgelegt, welche Kindelemente ein XML-Element haben darf. Ein ihnen zugefügter Zusatz würde sich also auf die Kombination von ihnen erlaubter Kindelementen beziehen.

Bei solchen Definitionen kann mit den Elementen `xsd:any` und `xsd:anyAttribute` festgelegt werden, dass ein beliebiges XML-Element als Kindelement bzw. beliebige Attribute zugelassen werden. In beiden Fällen können die Namensräume beschränkt werden, aus denen die Kindelemente bzw. Attribute stammen. Wird den Elementen `xsd:any` oder `xsd:anyAttribute` ein Zusatz eines Spezifikationsverfahrens zugefügt, könnten diese weitere Einschränkungen über erlaubte Kindelemente bzw. Attribute ausdrücken.

XML-Schema erlaubt zwei Arten, Typen aus anderen Typen abzuleiten: die Extension und die Restriction. Mit der Extension (`xsd:extension`) können einem Typ weitere Kindelemente oder Attribute zugefügt werden. Bei der Restriction (`xsd:restriction`) wird dagegen die Wertemenge eines Typs beschränkt. In beiden Fällen könnte ein Spezifikationsverfahren weitere Informationen als Zusätze hinzufügen. Bei der Restriction könnte z. B. mit einem Zusatz eines Spezifikationsverfahrens die Wertemenge auf eine Art eingeschränkt werden, die mit XML-Schema nicht möglich ist.

Zur Definition von Typen seien schließlich noch die Elemente `xsd:group` und `xsd:attributeGroup` angesprochen. Sie erlauben es, eine Menge von Kindelementen bzw. Attributen zu benennen, so dass diese bei der Definition von komplexen Typen angegeben werden kann. Werden solchen Gruppen Zusätze eines Spezifikationsverfahrens zugefügt, können sie die Kindelemente bzw. Attribute der Gruppe miteinander in Beziehung setzen. Diese Beziehung könnte dann auch für alle XML-Elemente gelten, deren komplexe Typen diese Gruppe enthalten.

Namen solcher Gruppen ebenso wie Namen von benannten Typen und global definierten XML-Elementen und Attributen, die in einem XML-Schemadokument definiert werden, bilden zusammen einen Namensraum. Dieser wird durch das XML-Schemadokument definiert. Spezifikationsverfahren könnten auch Zusätze enthalten, die den Namensraum als Ganzes genauer beschreiben und daher nicht den vorgenannten Elementen zugeordnet werden können. Solche Zusätze sollten direkt dem Element `xsd:schema` als Erweiterungselement oder -attribut zugeordnet werden.

8.2.4 SOAP-Nachricht

SOAP-Nachrichten sind im WSDL-Dokument nicht in XML-Schema deklariert. Der Rahmen der SOAP-Nachricht ist im SOAP-Standard beschrieben. In XML-Schema sind die Body-, Header und Detaileinträge beschrieben, die in der SOAP-Nachricht enthalten

sind. Welche von diesen zusammen eine SOAP-Nachricht bilden, wird aber direkt mit Elementen von WSDL festgelegt.

Für die Frage, welchen Elementen von WSDL in welchen Fällen Zusätze zuzufügen sind, ist zu beachten, dass WSDL nicht nur zur Beschreibung von SOAP-basierten Webservices verwendet werden kann. Ohne die Festlegungen für den Präzisierungsstandard aus Kapitel 5.3.1 können beliebige Protokollbindungen unterstützt werden, also beliebige Protokolle mit beliebigen Übertragungsformaten. Hierzu werden in WSDL die Schnittstellen mit einigen Elementen zunächst abstrakt beschrieben und dann in der Protokollbindung mit weiteren Elementen an Protokoll und Übertragungsformat gebunden.

Für Zusätze von Spezifikationen ist daher wichtig, ob sie die Schnittstelle des Webservices unabhängig von der Protokollbindung beschreiben oder nicht. Sind sie von ihr unabhängig, sollten sie den Elementen der abstrakten Schnittstelle zugefügt werden. Das sind die Elemente `wsdl:message` und `wsdl:portType` sowie ihre Nachfahrenelemente. Ansonsten sollten die Elemente der Protokollbindung oder des Webservices erweitert werden, also das Element `wsdl:binding` bzw. `wsdl:service` oder deren Nachfahrenelemente. Von der Protokollbindung abhängig ist ein Zusatz schon dann, wenn vorausgesetzt wird, dass XML- oder SOAP-Nachrichten ausgetauscht werden.

Bezieht sich ein Zusatz eines Spezifikationsverfahrens unabhängig von der Protokollbindung auf eine einzelne Nachricht, kann dieser entweder dem Element `wsdl:message` oder Kindelementen des Elementes `wsdl:portType/wsdl:operation` zugefügt werden. Letztere werden verwendet, um die Nachrichten zu Operationen zusammenzufassen. Welche der beiden Möglichkeiten gewählt wird, spielt prinzipiell keine Rolle. Die Verwendung des Elementes `wsdl:message` erscheint jedoch naheliegender, da es schon von seinem Namen her offensichtlich eine Nachricht beschreibt. Außerdem ist es weniger tief in der Hierarchie des WSDL-Dokumentes „verborgen“.

Ist der Zusatz dagegen von der Protokollbindung abhängig, muss er innerhalb von deren Element `wsdl:binding` beschrieben werden. Einzelne SOAP-Nachrichten werden in ihm durch die Elemente `wsdl:binding/wsdl:operation/wsdl:input` sowie `wsdl:binding/wsdl:operation/wsdl:output` beschrieben, je nachdem ob es sich um einen Request oder Response handelt. Diesen Elementen sollte also ein von der Protokollbindung abhängiger, sich auf eine einzelne SOAP-Nachricht beziehender Zusatz zugefügt werden.

Besonderer Beachtung gebühren Fehlernachrichten, die in WSDL von der Protokollbindung unabhängig oder von ihr abhängig beschrieben werden können. Unabhängig von der Protokollbindung werden Detaileinträge zunächst in XML-Schema definiert und mit einem Element `wsdl:message` zusammengefasst. Dieses wird mit einem Element `wsdl:portType/wsdl:operation/wsdl:fault` einer Operation zugeordnet, wodurch explizit wird, dass es sich um eine Fehlernachricht handelt. Daher sollten diesen Elementen Zusätze zugefügt werden, die sich unabhängig von der Protokollbindung auf einzelne Fehlernachrichten beziehen. Ist der Zusatz dagegen von der Protokollbindung abhängig, sollte das Element `wsdl:binding/wsdl:operation/wsdl:fault` erweitert werden, mit dem Fehlernachrichten in dieser Art beschrieben werden.

8.2.5 Request-/Response-Paar oder Schnittstelle

Ebenso wie für einzelne Nachrichten muss auch für Zusätze, die sich auf ein Request-/Response-Paar oder eine gesamte Schnittstelle beziehen, beachtet werden, ob sie von der Protokollbindung abhängig sind oder nicht. Beides wird in WSDL zunächst von ihr

unabhängig definiert. Die Beschreibung der abstrakten Schnittstelle ist in einem Element `wsdl:portType` enthalten und die der Request-/Response-Paare in Elementen `wsdl:portType/wsdl:operation`. Die abstrakte Schnittstelle wird mit dem Element `wsdl:binding` an ein Protokoll gebunden und so die gebundene Schnittstelle definiert. Auch dieses Element enthält Kindelemente `wsdl:binding/wsdl:operation` zur Beschreibung der Request-/Response-Paare, in von der Protokollbindung abhängiger Weise.

Damit ist offensichtlich, dass Zusätze, die sich auf eine abstrakte Schnittstelle beziehen, dem Element `wsdl:portType` zugefügt werden müssen. Dem Element `wsdl:binding` werden analog Zusätze mit Bezug auf eine gebundene Schnittstelle zugefügt. Dazu sei darauf hingewiesen, dass eine gebundene Schnittstelle zwar das Protokoll und Nachrichtenformat enthält, jedoch keine Details über die Instanz des Webservices. Es kann mehrere solche Instanzen geben, die dieselbe gebundene Schnittstelle implementieren. Ein hier zugefügter Zusatz darf sich daher nicht auf Details einer Instanz beziehen, wie z. B. auf seine Netzwerkadresse.

Offensichtlich ist auch, dass Zusätze, die sich auf Request-/Response-Paare beziehen, Elementen `wsdl:portType/wsdl:operation` oder `wsdl:binding/wsdl:operation` zugefügt werden sollten. Ist der Zusatz von der Protokollbindung unabhängig muss das Element `wsdl:portType/wsdl:operation` erweitert werden, ansonsten das Element `wsdl:binding/wsdl:operation`. Solche Zusätze beschreiben auch das Verhalten einer Operation bei Fehlern, also in welchen Fällen welche Fehlernachricht als Response geliefert wird.

8.2.6 Mehrere Schnittstellen oder Instanz des Webservices

In den bisherigen Kapiteln 8.2.1 bis 8.2.5 wurden die Elemente von WSDL und XML-Schema untersucht, mit denen direkt oder indirekt einzelne Schnittstellen eines Webservices beschrieben werden. Neben der Möglichkeit einzelne Schnittstellen zu beschreiben, erlaubt WSDL jedoch auch die Beschreibung von Webservicetypen, also Mengen von abstrakten Schnittstellen, sowie die Beschreibung von Instanzen von Webservices. Für eine Instanz eines Webservices wird vor allem beschrieben, welche gebundenen Schnittstellen er hat.

Der Fokus dieser Arbeit liegt in der Spezifikation einzelner Schnittstellen von Webservices. Es könnte jedoch auch Spezifikationsverfahren geben, die Zusätze verwenden, um Mengen von abstrakten Schnittstellen oder Instanzen von Webservices genauer zu beschreiben. Den Elementen, die zu deren Beschreibung in WSDL enthalten sind, müssten die Zusätze dann als Erweiterungselemente oder -attribute zugefügt werden. Für eine Menge abstrakter Schnittstellen wäre das also das Element `wsdl:serviceType` für Webservicetypen.

Analog müsste für einen Zusatz, der sich auf eine Instanz eines Webservices bezieht das Element `wsdl:service` erweitert werden. Da hier auch die Menge von Instanzen von gebunden Schnittstellen angegeben wird, die der Webservice unterstützt, müssten diesem Element auch Zusätze zugefügt werden, sie sich auf eine solche Menge beziehen. WSDL stellt jedoch keine Möglichkeit zur Verfügung, Mengen gebundener Schnittstellen zu beschreiben. Daher können Zusätze, die sich auf solche Mengen beziehen, auch nur dem Element `wsdl:service` zugeordnet werden.

Schließlich könnte es noch Zusätze geben, die sich auf eine einzelne Instanz einer Schnittstelle eines Webservices beziehen. In WSDL wird innerhalb des Elementes

`wsdl:service/wsdl:port` zu ihrer Beschreibung z. B. ihre Netzwerkadresse angegeben. Zusätze, die einer solchen Beschreibung weitere Informationen zufügen, sollten diesem Element zugeordnet werden.

8.3 Anwendung auf SXQT-Ausdrücke

In den Kapiteln 8.1 und 8.2 wurde gezeigt, wie XML-Schema und WSDL für Zusätze von Spezifikationsverfahren erweitert werden sollte. Diese generelle Untersuchung soll nachfolgend auf SXQT, das Spezifikationsverfahren dieser Arbeit, angewendet werden. Die Zusätze sind dann die in Kapitel 7 eingeführten SXQT-Ausdrücke.

8.3.1 Ablageort für SXQT-Ausdrücke

In Kapitel 8.1.4 wurde allgemein für Zusätze festgestellt, dass sie nicht in eigenen Dateien gespeichert werden sollten. Statt dessen sollten Zusätze als Erweiterungselemente oder -attribute in XML-Schema oder WSDL abgelegt werden. Das gilt selbstverständlich auch für SXQT-Ausdrücke. Ob XML-Schema oder WSDL erweitert werden soll und welches Element, hängt davon ab, worauf sich ein Zusatz bezieht. Für das SXQT muss also geklärt werden, worauf sich SXQT-Ausdrücke beziehen.

Im Kapitel 6.1 sowie in der Klassifikation in Kapitel 7.7 wurden Anforderungen danach unterschieden, ob sie sich auf einzelne SOAP-Nachrichten, einzelne Request-/Response-Paare oder mehrere Operationen beziehen. Eine ähnliche Unterscheidung fand für Zusätze auch im Kapitel 8.2 statt. Dabei wurde der Bezug auf einzelne SOAP-Nachrichten jedoch noch weiter unterschieden. Außerdem wurden statt mehreren Operationen Schnittstellen, Webservicetypen und Instanzen von Webservices betrachtet.

Es wäre naheliegend, den Bezug der Anforderung auf den Bezug des SXQT-Ausdrucks zu übertragen, weil SXQT-Ausdrücke die Formulierungen von Anforderungen sind. Bezieht sich also z. B. eine Anforderung auf einzelne Request-/Response-Paare, könnte man den SXQT-Ausdruck, der sie formuliert auch dem Request-/Response-Paar zuordnen. Bezieht sich eine Anforderung auf einzelne SOAP-Nachrichten müsste genauer untersucht werden, ob sie sich tatsächlich nur z. B. auf ein XML-Element bezieht. Davon abhängig müsste der SXQT-Ausdruck entsprechend der Diskussion in Kapitel 8.2 zugeordnet werden. Ebenso müsste eine Zuordnung von SXQT-Ausdrücken für Anforderungen an mehrere Operationen gefunden werden.

Betrachtet man aber auf der anderen Seite, wie SXQT-Ausdrücke konstruiert werden, stellt man fest, dass sie stets den gesamten Trace auf einen der logischen Werte `true` oder `false` abbilden. Sie prüfen stets für alle Beobachtungen im Trace, ob diese der jeweiligen Anforderung entsprechen. Damit bezieht sich ein SXQT-Ausdruck tatsächlich immer auf den gesamten Trace, mit Ereignissen unterschiedlicher Operationsaufrufe, auch wenn sich die Anforderung selbst nur auf einzelne SOAP-Nachrichten oder Request-/Response-Paare bezogen hat. Damit können SXQT-Ausdrücke nur noch logischen oder abstrakten Schnittstellen, Webservicetypen und Instanzen von Webservices bzw. von gebundenen Schnittstellen zugeordnet werden.

SXQT beruht jedoch auf der Kenntnis, dass Webservices SOAP-Nachrichten austauschen. Die SOAP-Nachrichten sind im Trace als Kindelemente der Message-Elemente enthalten. Die SXQT-Ausdrücke greifen direkt auf Werte in ihnen zu. Damit sind die SXQT-Ausdrücke von der Protokollbindung abhängig. Nach Kapitel 8.2.5 und 8.2.6 sollten abstrakte Schnittstellen und Webservicetypen nur für Zusätze verwendet werden, die von der Protokollbindung unabhängig sind.

Somit bleibt die Frage, ob SXQT-Ausdrücke der gebundenen Schnittstelle oder einer Instanz des Webservices bzw. der gebundener Schnittstelle zugeordnet werden sollten. Würde ein SXQT-Ausdruck einer Instanz zugeordnet, würde er nur diese Instanz beschreiben. Ziel der Arbeit ist es jedoch, Schnittstellen zu beschreiben, die von mehreren Instanzen von Webservices implementiert werden können. Dann können Webclients über dieselben Schnittstellen mit unterschiedlichen Webservices kommunizieren. Daher müssen SXQT-Ausdrücke gebundenen Schnittstellen zugeordnet werden.

Das Element `wsdl:binding` muss also mit Erweiterungselementen oder -attributen erweitert werden. Für die Frage, welches von beiden verwendet werden sollte, wurde in Kapitel 8.1.2 festgestellt, dass Erweiterungsattribute nur verwendet werden können, wenn die Zusätze kurze Zeichenketten sind. Sind es dagegen große, mehrzeilige Zeichenketten sind Erweiterungselemente zweckmäßiger. Diese müssen verwendet werden, um Kindelemente oder Attribute zufügen zu können.

Die Beispiele für SXQT-Ausdrücke aus Kapitel 7.7 zeigen, dass es sich bei ihnen nicht um kurze Zeichenketten handelt. Es sind mehrzeilige Zeichenketten. Wie im folgenden Kapitel 8.3.2 im Detail vorgestellt wird, können sie auch in einer auf XML basierenden Syntax dargestellt werden. In beiden Fällen sind also Erweiterungselemente zweckmäßig bzw. notwendig. Daher sollten die SXQT-Ausdrücke dem Element `wsdl:binding` als Erweiterungselemente zugefügt werden.

8.3.2 XQuery- versus XQueryX-Syntax

SXQT-Ausdrücke sollten im Erweiterungselement in einer standardisierten Form enthalten sein. Das W3C hat zwei Notationen für XQuery-Ausdrücke standardisiert. Die bisher in dieser Arbeit verwendete XQuery-Syntax wurde entwickelt, um von Menschen gelesen und geschrieben zu werden. Ihre Grammatik wird in Anhang A von [Boag 02] beschrieben. Zusätzlich wird für XQuery, wie in Kapitel 8.3.1 erwähnt, eine Syntax zu Verfügung, die auf XML basiert. Sie wird als XQueryX bezeichnet und ist in einem eigenen Standard-Dokument [Malhotra 01] beschrieben, von dem zur Zeit jedoch nur der erste Arbeitsentwurf vom Juni 2001 existiert. Daher orientiert er sich an älteren Arbeitsentwürfen des Standards von XQuery, als sie für diese Arbeit verwendet wurden. Trotzdem kann schon an diesem Arbeitsentwurf erkannt werden, wie XQueryX später grundsätzlich aufgebaut sein wird.

Die Struktur von XQueryX ist an der XQuery-Syntax orientiert. Nach [Malhotra 01] wurde sie durch Abbildung der Produktionen der XQuery-Syntax auf komplexe Typen von XML-Elementen konstruiert. Auch Pfadausdrücke wurden auf eine Hierarchie von XML-Elementen abgebildet. In [Malhotra 01] wird festgestellt, dass XQuery-Ausdrücke in dieser Form von Menschen nicht besonders „angenehm" zu lesen oder zu schreiben sind. Durch die Verwendung von XML sind sie mit Programmen jedoch einfach zu parsen, zu erzeugen und zu interpretieren.

Ein wesentlicher Teil des Arbeitsentwurfes von XQueryX besteht aus Beispielen, in denen für Ausdrücke in XQuery-Syntax die Notation in XQueryX gezeigt wird. Auch in dieser Arbeit soll an einem Beispiel der Aufbau von XQueryX erläutert werden. Verwendet wird der einfache SXQT-Ausdruck aus Abbildung 104. Um die Darstellung zu verkürzen, wurde in Abbildung 104 der Query-Prolog nicht mit abgebildet. Abbildung 123 zeigt den SXQT-Ausdruck erneut mit Query-Prolog, in dem die beiden benötigten Namensräume an Präfixe gebunden werden.

```
declare namespace my="http://example.org/MultirefSample"
declare namespace opr="http://ti5.tu-harburg.de/venzke/20021015/operations"
every $a in opr:tr()//my:a
  satisfies fn:not(fn:exists($a/@ref) eq fn:exists($a/my:str))
```

Abbildung 123: Beispiel für SXQT-Ausdruck in XQuery-Syntax

Neben dem Query-Prolog besteht der SXQT-Ausdruck in Abbildung 123 aus nur zwei Zeilen. Mit einem Allquantor wird für die vom Teilausdruck `opr:tr()//my:a` gelieferten Elemente geprüft, ob für sie der Teilausdruck nach dem Schlüsselwort `satisfies` zutrifft. Beide Teilausdrücke verwenden Pfadausdrücke und Funktionen, der zweite außerdem die Operation `eq`.

```
<?xml version="1.0" encoding="UTF-8"?>
<q:query xmlns:q="http://www.w3.org/2001/06/xqueryx"
   xmlns:opr="http://ti5.tu-harburg.de/venzke/20021015/operations"
   xmlns:op="http://www.w3.org/2002/11/xquery-operators"
   xmlns:fn="http://www.w3.org/2002/11/xquery-functions"
   xmlns:my="http://example.org/MultirefSample"
  <q:quantifier type="EVERY">
    <q:quantifierAssignment variable="$a">
      <q:step axis="SLASHSLASH">
        <q:function name="opr:tr" />
        <q:identifier>my:a</q:identifier>
      </q:step>
    </q:quantifierAssignment>
    <q:function name="op:eq">
      <q:function name="fn:not">
        <q:function name="fn:exists">
          <q:step axis="ATTRIBUTE">
            <q:variable>$a</q:variable>
            <q:identifier>ref</q:identifier>
          </q:step>
        </q:function>
      </q:function>
      <q:function name="fn:exists">
        <q:step axis="CHILD">
          <q:variable>$a</q:variable>
          <q:identifier>my:str</q:identifier>
        </q:step>
      </q:function>
    </q:function>
  </q:quantifier>
</q:query>
```

Abbildung 124: Beispiel für SXQT-Ausdruck in XQueryX-Syntax

Abbildung 124 zeigt den gleichen SXQT-Ausdruck in XQueryX[56]. Benötigte Namensräume werden hier nicht im Query-Prolog, sondern mit den Mechanismen von XML an Präfixe gebunden. Im Beispiel geschieht das im Element `q:query`[57], das zusammen mit seinen Nachfahren den XQuery-Ausdruck repräsentiert.

[56] Zu Abbildung 124 sei angemerkt, dass zur Struktur des XQueryX-Arbeitsentwurfes vom Juni 2001 die gleichen Operationen und Funktionen wie in Abbildung 123 verwendet wurden. Diese gehören jedoch zum XQuery-Arbeitsentwurf vom November 2002. Für die Verwendung mit XQueryX ist das jedoch kein Problem. Außerdem wurde das Beispiel in Abbildung 124 gegen ein XML-Schemadokument validiert, in dem offensichtliche Fehler in XML-Schemadokument von XQueryX aus [Malhotra 01] korrigiert wurden.

[57] Das Präfix `q` stehe für den Namensraum `http://www.w3.org/2001/06/xqueryx`.

Der Allquantor des SXQT-Ausdrucks wird zum Element `q:quantifier`. In dessen Attribut `type` wird angegeben, um welchen der beiden Quantoren es sich handelt. Seine beiden Kindelemente stehen für der beiden Teilausdrücke, mit denen angegeben wird, welche Elemente der Allquantor untersuchen soll und was für diese prüfen ist. Als erstes Kindelement wird stets das Element `q:quantifierAssignment` verwendet. In dessen Attribut `variable` wird der Name der Variable des Allquantors angegeben.

Aufrufe von Funktionen und Operationen werden als Element `q:function` notiert. Im Attribut `name` wird der Name der Funktion bzw. Operation angegeben, die Parameter bzw. Operanden als Kindelemente.

Schließlich werden im Beispiel noch Pfadausdrücke verwendet, die vereinfacht in Kapitel 7.4.6 vorgestellt wurden. Pfadausdrücke bestehen aus Ausführungsschritten, die in der XQuery-Syntax in der Regel durch das Zeichen `/` voneinander getrennt werden. In XQueryX wird jeder dieser Ausführungsschritte einzeln als ein Element `q:step` notiert. Die Achse des Ausführungsschritts wird im Attribut `axis` angegeben. Der Ausdruck, auf den er angewendet werden soll, ist das erste Kindelement. Details des Ausführungsschritts sind im zweiten Kindelement enthalten.

Die Tatsache, dass XQuery-Ausdrücke auf zwei Arten notiert werden können, führt zur Frage, in welchen Fällen welche Art verwendet werden sollte. Auch für SXQT-Ausdrücke muss das geklärt werden. Sollen sie im Erweiterungselement in XQuery-Syntax oder in XQueryX enthalten sein?

Bereits am Anfang dieses Kapitels 8.3.2 wurde auf die Aussage aus [Malhotra 01] hingewiesen, dass in XQueryX notierte XQuery-Ausdrücke von Menschen nicht besonders „angenehm“ zu lesen oder zu schreiben sind. In der XQuery-Syntax fällt das leichter. Das ist anhand des Beispiels in Abbildung 123 und Abbildung 124 leicht nachzuvollziehen. Statt einem XQuery-Ausdruck von vier Zeilen in XQuery-Syntax in Abbildung 123 werden in XQueryX in Abbildung 124 31 Zeilen benötigt. Die XQuery-Syntax fasst den XQuery-Ausdruck also viel prägnanter zusammen.

XQueryX hat durch die Verwendung von XML jedoch Vorteile, wenn es um eine maschinelle Verarbeitung geht. Zwar können XQuery-Ausdrücke in beiden Fällen maschinell verarbeitet und generiert werden. Bei Verwendung von XQueryX ist das jedoch direkt mit Standardwerkzeugen für XML möglich, um z. B. XQuery-Ausdrücke zu parsen, zu interpretieren oder auch zu generieren. In [Malhotra 01] wird z. B. kurz die Idee vorgestellt, XQuery-Ausdrücke in XQueryX selbst mit XQuery zu verarbeiten und so neue XQuery-Ausdrücke zu generieren. Gleiches könnte auch mit anderen XML-basierten Sprachen, wie z. B. XSLT geschehen.

Beide Arten XQuery-Ausdrücke zu notieren haben also ihre Vorteile. Das gilt auch für SXQT-Ausdrücke. Für sie ist es aber vor allem wichtig, dass sie möglichst gut von Menschen gelesen und geschrieben werden können. Ansonsten kann es zu Missverständnissen über die Spezifikation kommen, die zu Inkompatibilitäten zwischen Webclient und Webservice führen können. Daher sollten die SXQT-Ausdrücke in der XQuery-Syntax in WSDL eingebettet werden. Eine maschinelle Verarbeitung ist trotzdem weiterhin möglich, nur nicht mit Standardwerkzeugen für XML.

8.3.3 Vorschlag für Erweiterung von WSDL für SXQT-Ausdrücke

Somit ist das Ergebnis der Diskussion in den Kapiteln 8.3.1 und 8.3.2, dass SXQT-Ausdrücke in der XQuery-Syntax, also als mehrzeilige Zeichenketten, mit Hilfe von Erweiterungselementen in WSDL eingebettet werden sollten. Das sollte in der Defini-

tion der gebunden Schnittstelle geschehen, also im Element `wsdl:binding`. Von dieser Feststellung ausgehend, soll nachfolgend vorgeschlagen werden, wie das Erweiterungselement heißen und welche Struktur es haben soll.

Der Name des Erweitungselementes sei `wex:assert`. Der lokale Name `assert` legt nahe, dass zusätzlich zur Beschreibung in WSDL etwas zugesichert werden soll. SXQT-Ausdrücke stellen solche Zusicherungen dar. Das Präfix `wex` sei an den Namensraum `http://ti5.tu-harburg.de/venzke/20021015/wsdl-extension` gebunden.

Es wäre vorteilhaft, wenn die Bedeutung des SXQT-Ausdrucks natürlichsprachlich kommentiert werden könnte. Z. B. ließe sich die als SXQT-Ausdruck formulierte Anforderung natürlichsprachlich angegeben. Das vereinfacht es menschlichen Lesern, sich einen Überblick über die Spezifikation zu verschaffen.

In WSDL wird für natürlichsprachliche Kommentare stets das Element `wsdl:documentation` verwendet. Es kann jedem Element von WSDL als Kindelement zugefügt werden. Als Inhalt erlaubt es neben Text auch beliebige Kindelemente. Da das Element `wex:assert` selbst WSDL erweitert, ist es sinnvoll auch in diesem optional das Element `wsdl:documentation` als Kindelement für natürlichsprachliche Kommentare zuzulassen. Das Element `wex:assert` habe daher optional das erste Kindelement `wsdl:documentation`.

Als zweites Kindelement habe das Element `wex:assert` das Element `wex:xqueryExpression`. Dieses enthält den SXQT-Ausdruck in XQuery-Syntax, also als Zeichenkette. Die Struktur des Elementes `wex:assert` ist zusammenfassend in Abbildung 125 zusammen mit der Einbettung in das Element `wsdl:binding` veranschaulicht.

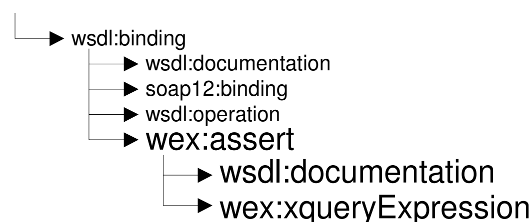

Abbildung 125: Struktur der Ereiterungselemente für WSDL, eingebettet in gebundene Schnittstelle

Der Struktur könnten weitere Informationen über den SXQT-Ausdruck zugefügt werden. Erforderlich ist das für die Sicht, in der der SXQT-Ausdruck gültig ist. In Kapitel 7.3.4 wurde festgestellt, dass die benötigt Sicht für einen logischen Ausdruck bekannt sein muss, wenn er nicht von ihr unabhängig ist. Daher wird die für einen SXQT-Ausdruck benötigte Sicht im Attribut `wex:xqueryExpression/@viewEntity` angegeben, für das drei Werte erlaubt sind. Der Wert `Client` zeigt an, dass der SXQT-Ausdruck nur für die Webclientsicht gültig ist. Beim Wert `Service` ist er nur für die Webservicesicht gültig. Spielt die beobachtete Entität keine Rolle, wird der Wert `Any` angegeben.

Die Elemente `wex:assert`, `wex:xqueryExpression` sowie das Attribut `viewEntity` sind im XML-Schemadokument in Abbildung 126 deklariert. Für die Deklarationen der beiden Elemente enthält es jeweils einen benannten, komplexen Typ. Auch für das Attribut `viewEntity` ist ein Typ definiert, in dem die drei gültigen Werte aufgezählt werden.

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:wsdl="http://www.w3.org/2002/07/wsdl"
  xmlns:wex="http://ti5.tu-harburg.de/venzke/20021015/wsdl-extension"
  targetNamespace="http://ti5.tu-harburg.de/venzke/20021015/wsdl-extension">

  <xsd:import namespace="http://www.w3.org/2002/07/wsdl" />

  <xsd:simpleType name="viewEntityType">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="Client" />
      <xsd:enumeration value="Service" />
      <xsd:enumeration value="Any" />
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:complexType name="xqueryExpressionType">
    <xsd:simpleContent>
      <xsd:extension base="xsd:string">
        <xsd:attribute name="viewEntity"
                       type="wex:viewEntityType"
                       default="Any" />
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>

  <xsd:element name="xqueryExpression"
               type="wex:xqueryExpressionType" />

  <xsd:complexType name="assertType">
    <xsd:sequence>
      <xsd:element ref="wsdl:documentation" minOccurs="0" />
      <xsd:element ref="wex:xqueryExpression" />
    </xsd:sequence>
  </xsd:complexType>

  <xsd:element name="assert" type="wex:assertType" />
</xsd:schema>
```

Abbildung 126: XML-Schemadokument der Erweiterungselemente für WSDL

Um die Einbettung von SXQT-Ausdrücken in WSDL zu veranschaulichen, soll abschließend ein Ausschnitt eines WSDL-Dokumentes beispielhaft um SXQT-Ausdrücke erweitert werden. Der in Abbildung 127 gezeigte Ausschnitt zeigt die Definition der gebundenen Schnittstelle der in Kapitel 2.6 vorgestellten Webkomponente für eine Internetzeitung. Zur Verkürzung ausgelassene Teile wurden durch drei Punkte („...“) ersetzt.

Von den vorhandenen Operationen ist in Abbildung 127 nur die Operation `Login` mit einem Element `wsdl:operation` angedeutet. Dahinter wurden Erweiterungselemente `wex:assert` zugefügt. Eines von ihnen ist vollständig dargestellt, ein weiteres angedeutet. Da jeder SXQT-Ausdruck in einem eigenen Erweiterungselement eingefügt wird, kann die Deklaration der gebundenen Schnittstelle viele von ihnen enthalten. Auch die Auslassung der Erweiterungselemente ist in Abbildung 127 mit drei Punkten („...“) angedeutet.

Das vollständig gezeigte Element `wex:assert` beschreibt eine Anforderung, die bereits im Kapitel 6.2.1 vorgestellt und für die im Kapitel 7.7.3.1 in Abbildung 103 der SXQT-Ausdruck angegeben wurde. Die Anforderung besagt, dass einerseits im Request der

Operation `Login` ein Passwort angegeben sein muss, wenn der Benutzername ein anderer als „Gast“ ist. Ist andererseits der Benutzername „Gast“ angegeben, muss das Passwort fehlen. In natürlicher Sprache ist diese Anforderung in Abbildung 127 im Element `wex:assert/wsdl:documentation` enthalten.

```
<wsdl:binding name="NewsServiceBinding"
              type="nwsi:NewsServicePortType" >
  <soap12:binding
      style="document"
      transport="http://www.w3.org/2002/06/soap/bindings/HTTP/" />

  <wsdl:operation name="Login" > ... </wsdl:operation>
  ...

  <wex:assert>
    <wsdl:documentation>
      Im Request der Operation Login muss ein Passwort angegeben
      sein, wenn der Benutzername ein anderer als "Gast" ist.
      Wird als Benutzername "Gast" angegeben, darf kein
      Passwort angegeben werden.
    </wsdl:documentation>

    <wex:xqueryExpression viewEntity="Any">
      declare namespace
              xsd="http://www.w3.org/2001/XMLSchema"
      declare namespace
              opr="http://ti5.tu-harburg.de/venzke/20021015/operations"
      declare namespace
              nwst="http://example.org/NewsService020917/Types"

      every $m
       in opr:restrict(opr:tr(),
                       fn:QName-in-context ("nwst:Login",
                                            fn:false()))
       satisfies
         (opr:event-body-entry($m)/nwst:Name eq "Gast")
         eq fn:empty(opr:event-body-entry($m)/nwst:Password)
    </wex:xqueryExpression>
  </wex:assert>

  <wex:assert> ... </wex:assert>
  ...
</wsdl:binding>
```

Abbildung 127: Ausschnitt aus WSDL-Dokument mit Erweiterungselementen

Das Element `wex:assert/wex:xqueryExpression` enthält die Anforderung als SXQT-Ausdruck in XQuery-Syntax. In Abbildung 103 wurde sie zur Verkürzung ohne Query-Prolog dargestellt. Dieser ist in Abbildung 127 ergänzt worden. Da die Anforderung zur Klasse CA gehört und sich daher nur auf einzelne SOAP-Nachrichten bezieht, spielt für den SXQT-Ausdruck die Sicht keine Rolle, wie in Kapitel 7.7.2 festgestellt. Daher hat das Attribut `viewEntity` den Wert `Any`.

9 Automatische Validation

Ein um SXQT-Ausdrücke erweitertes WSDL-Dokument, als SXQT-Spezifikation eines Webservices, kann für unterschiedliche Zwecke verwendet werden. Es kann von Entwicklern gelesen werden, die Webclient oder Webservice entwickeln. So erhalten sie Kenntnis über die Schnittstelle, so dass sie den Webclient bzw. Webservice nach ihr implementieren können.

Hat der Entwickler die Spezifikation nicht korrekt verstanden oder sind Programmierfehler aufgetreten, kann es geschehen, dass eine Webkomponente von der Spezifikation abweicht. Solche Abweichungen sollten nach Möglichkeit erkannt werden, da sie zu Interoperabilitätsproblemen zwischen den Webkomponenten führen können.

Anweichungen von der Spezifikation können auch auftreten, wenn der Webservice geändert wird. Um neue Funktionalitäten des Webservices zu unterstützen, können Änderungen der Schnittstelle notwendig werden. Aus Sicht des Webclients erfüllt der Webservice dann nicht mehr die Spezifikation, gegen die der Webclient implementiert wurde. Das kann für den Webclient relevant sein, muss aber nicht. Daher wird ein Weg benötigt zu erkennen, ob eine Änderung der Schnittstelle des Webservices stattgefunden hat, die für den Webclient relevant ist.

Beide Herausforderungen sollen mit der gleichen Vorgehensweise gelöst werden. Es ist die automatische Validation von Webkomponenten, die in diesem Kapitel entwickelt wird. Für jeden beobachteten Trace wird geprüft, ob er der Spezifikation entspricht.

9.1 Programmchecker

Die automatische Validation von Webkomponenten wurde aus der Idee der Programmchecker entwickelt, die in der Literatur auch als „Checker“, „Simple Checker“, „Self-Testing Programs“ oder „Program Result Checker“ bezeichnet werden. Vorgestellt wurden Programmchecker zuerst in [Blum 89] sowie von den gleichen Autoren mit gleichem Titel in überarbeiteter Form in [Blum 95].

Die Idee von Programmcheckern ist wie folgt: Sei `P` ein Programm das eine Funktion π berechnen soll. Für eine Eingabe `x` liefert es eine Ausgabe `y`. Ein Programmchecker für das Programm `P` ist ein weiteres Programm `C`, das nach `P` ausgeführt wird und prüft, ob die von `P` gelieferte Ausgabe `y` für die Eingabe `x` korrekt ist, ob also $y = \pi(x)$ gilt. Ist das der Fall, bestätigt es `C`. Sonst stellt `C` fest, dass `P` fehlerhaft ist, wie in Abbildung 128 dargestellt. Formaler wird das in [Blum 95] mit Hilfe von Funktionen definiert, so dass `P` und `C` Funktionen sind.

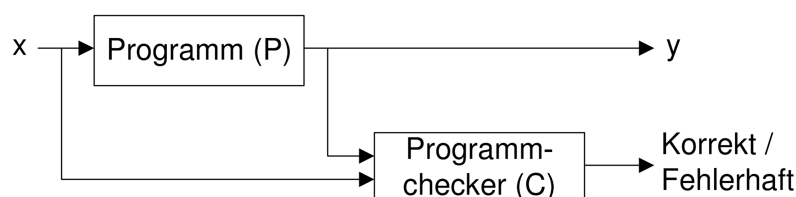

Abbildung 128: Funktionsweise eines Programmcheckers

Auch der Programmchecker `C` kann fehlerhaft sein. Daher wird in [Blum 95] festgelegt, dass seine Entscheidungen nur mit einer bestimmten Wahrscheinlichkeit korrekt sein muss. Um das Ergebnis zu testen, darf der Programmchecker `C` grundsätzlich auch das Programm `P` mehrfach aufrufen. Es muss jedoch sichergestellt sein, dass der Programmchecker `C` anders arbeitet als `P` selbst. Ansonsten besteht die Gefahr, dass er die Fehler von `P` nicht erkennt, weil bei ihm die gleichen Fehler auftreten. Um das zu erzwingen wird vorgeschlagen, dass die Ausführungszeit von `C` weniger stark wachsen darf als die von `P`[58].

Für das Modell der Programmchecker wurden in der Literatur unterschiedliche Erweiterungen vorgestellt. So wird in [Rubinfeld 96] die Idee zugefügt, dass Programmchecker parallele Programme sein können. Das ist erforderlich, weil sequentielle Programmchecker für parallele Algorithmen als nicht effizient genug angesehen werden.

In [Goerigk 98] werden Programmchecker in Kombination mit der Programmverifikation verwendet. Wenn möglich wird die Korrektheit von Programmen mit Hilfe der Programmverifikation sichergestellt. Das spart die Ausführungszeit für Programmchecker. Für Programme komplexer Probleme kann die Programmverifikation jedoch zu aufwendig sein. Dann wird es nicht verifiziert, sondern statt dessen seine Ergebnisse mit einem Programmchecker geprüft und der Programmchecker verifiziert. Das führt zu partiell korrekten Programmen, die also stets korrekte Ergebnisse liefern oder überhaupt keines.

Angewendet wurde diese Vorgehensweise z. B. in [Gaul 99] auf das Back-End eines Compilers und in [Bartsch 01] auf ein Expertensystem. In beiden Fällen erschwert die Verwendung von Heuristiken zur Optimierung die Programmverifikation. Die Ergebnisse lassen sich jedoch einfach prüfen, weil sie durch Regeln beschrieben sind, die überprüft werden können. So kann sichergestellt werden, dass gelieferte Ergebnisse korrekt sind.

In [Blum 93] wird das Modell aus [Blum 89] auf drei Arten erweitert. Zunächst werden Programmchecker so definiert, dass sie vom geprüften Programm unabhängig sind. Sie sind nur von der zu berechnenden Funktion π abhängig und können daher zum Prüfen aller Programme verwendet werden, die diese Funktion implementieren.

Neben Programmcheckern wird außerdem die Idee selbstkorrigierender Programme `Cf` vorgestellt. Auch diese sind nur von der zu berechnenden Funktion π abhängig. Sei `P` ein Programm, das π berechnet und potentiell fehlerhaft ist, wobei jedoch die Fehlerwahrscheinlichkeit „niedrig genug" sei. Dann berechnet das Programm `Cf` mit Hilfe von `P` stets korrekt die Funktion π. `P` kann dazu mehrfach aufgerufen werden. Wie selbstkorrigierende Programme implementiert werden, wird in [Blum 93] unter anderem an Beispielen gezeigt.

Die Idee einzelner Programmchecker und selbstkorrigierender Programme wird in [Blum 93] auch auf Bibliotheken verallgemeinert. Für eine Bibliothek potentiell fehlerhafter Funktionen lässt sich eine Bibliothek von Programmcheckern und selbstkorrigierenden Funktionen entwickeln. Der Vorteil gegenüber einzelnen liegt hier darin, dass sich Programmchecker und selbstkorrigierende Funktionen gegenseitig verwenden können, was ihre Implementierung vereinfacht.

---

[58] In [Blum 95] wird hierzu festgelegt, dass die Ausführungszeit von `C` „little oh" von der von `P` sein muss.

9.2 Zuverlässigkeit mit Programmcheckern

[Wasserman 97] enthält eine interessante Untersuchung, wie sich die Zuverlässigkeit von Software mit Hilfe von Programmcheckern verbessern lässt. Für diese Arbeit wichtige Aspekte sollen nachfolgend zusammengefasst werden.

9.2.1 Alternativen

Zunächst geht [Wasserman 97] auf drei Alternativen zu Programmcheckern ein. Es wird festgestellt, dass das Finden von Fehlern in Software schwierig ist. Typischerweise wird Software hierzu getestet. Das zu testende Programm wird für sorgsam gewählte Eingaben ausgeführt und durch Menschen geprüft, ob seine Ausgaben dem Erwarteten entsprechen. Neben der Frage, ob die Performanz des Programms ausreicht, ist beim Testen das Hauptproblem, dass die Software nur für einige Eingaben getestet werden kann. Ob die Ausgaben für nicht getestete Eingaben korrekt sind, bleibt ungewiss.

Ob eine Software für beliebige Eingaben korrekt ist, lässt sich dagegen mit der Programmverifikation feststellen. Bei ihr wird eine mathematische Anforderung an das Verhalten der Software formal bewiesen. Die Konstruktion solcher Beweise ist jedoch sehr aufwendig.

Neben dem Testen und der Programmverifikation wird als dritte Alternative die Fehlertoleranz angeführt, bei der mehrere Softwareteams ein Programm für die gleiche Aufgabe implementieren. Für eine Eingabe werden alle diese Programme gestartet und ihre Ergebnisse verglichen. Nachteil ist die Ineffizienz, die sich durch die mehrfache Softwareentwicklung und Ausführung ergibt. Alle drei Alternativen haben also Nachteile, was die Untersuchung von Programmcheckern motiviert.

9.2.2 Fehlersuche

Programmchecker können für die Fehlersuche in Softwareprodukten eingesetzt werden und so zur Steigerung von deren Zuverlässigkeit beitragen. Dazu werden sie in die Softwareprodukte eingebettet, was das Erkennen fehlerhafter Ausgaben während des Testens erlaubt.

Wird der Programmchecker auch nach dem Testen, also im produktiven Betrieb, im Softwareprodukt belassen, ermöglicht das auch dann auftretende Fehler sofort zu erkennen. Gefundene Fehler können in Dateien gespeichert und so periodisch von Software-Wartungspersonal ausgewertet werden. In kritischen Systemen ist es darüber hinaus möglich, den Benutzern eine Warnmeldung anzuzeigen. Auch kann mit Programmcheckern ein Fehler in einer Komponente abgefangen werden, bevor er eine katastrophale Fehlersituation verursacht.

Überhaupt erlauben Programmchecker, Missverständnisse über Schnittstellen von Komponenten[59] aufzudecken. Hierzu könnte z. B. eine Gruppe von Entwicklern, die eine Komponente `A` einer anderen Gruppe verwendet, für diese einen Programmchecker erstellen. Für ihre eigene Komponente `B` können sie so sicherstellen, dass deren Funktionalität nicht durch ein fehlerhaftes Verhalten der Komponente `A` gefährdet wird. Alternativ könnte auch der Entwickler, der für eine Komponente die Spezifikation erstellt, zusätzlich einen Programmchecker für die Komponente entwickeln. Mit diesem kann geprüft werden, ob die Programmierer der Komponente die Spezifikation verstanden haben.

[59] Statt von Schnittstellen von Komponenten wird in [Wasserman 97] vom „Rand von Programmen" („'seams' of programs") gesprochen. Gemeint ist jedoch im Wesentlichen das Gleiche.

Da Programmchecker nur untersuchen, welche Ein- und Ausgaben eine Komponente erhält, kann die Komponente geändert werden, ohne dass auch der Programmchecker geändert werden muss. Wird ein neuer Algorithmus eingesetzt, können auch bei diesem sofort Fehler erkannt werden. Wird die Komponente nach und nach neuen Erfordernissen angepasst, möglicherweise auch ohne erneutes Testen, stellt der Programmchecker immer weiter die Korrektheit ihrer Ergebnisse sicher.

9.2.3 Korrektheit von Programmcheckern

Damit Programmchecker zuverlässig Ausgaben von Programmen testen, ist es erforderlich, dass sie selbst korrekt sind. Daher wurden sie in [Goerigk 98], [Gaul 99] und [Bartsch 01] verifiziert. In [Wasserman 97] wird dazu zumindest festgestellt, dass Programmchecker gründlich getestet werden müssen. Für die Tests bietet es sich an, die Programme, die der Programmchecker testen soll, zu modifizieren, so dass sie fehlerhafte Ausgaben liefern. Dann muss der Programmchecker einen Fehler anzeigen.

In [Wasserman 97] wird darüber hinaus untersucht, was geschieht, wenn ein Programmchecker fehlerhaft ist. Unterschieden werden vier Möglichkeiten, je nachdem ob für eine Eingabe `x` das zu testende Programm `P` die korrekte oder falsche Ausgabe liefert und ob der Programmchecker `C` das erkennt oder nicht. Die vier Möglichkeiten sind in Tabelle 6 zusammengefasst.

	`C` stellt fest, dass `P(x)` korrekt	`C` stellt fest, dass `P(x)` inkorrekt
`P(x)` ist korrekt	OK	Falscher Alarm
`P(x)` ist inkorrekt	Fehler nicht erkannt	OK

Tabelle 6: Mögliche Fehlersituationen mit Programmchecker

Der gewünschte Fall ist, dass `P` die korrekte Ausgabe liefert und `C` diese als korrekt erkennt. Liefert `P` dagegen eine inkorrekte Ausgabe, sollte `C` das auch erkennen. Durch einen Fehler in `C` kann es jedoch einerseits dazu kommen, dass `P` eine korrekte Ausgabe liefert, die jedoch von `C` als inkorrekt erkannt wird („Falscher Alarm“) oder dass `P` eine falsche Ausgabe liefert, die von `C` als korrekt erkannt wird („Fehler nicht erkannt“). Nur der letzte Fall ist wirklich schlimm, denn durch einen falschen Alarm wird zumindest die Aufmerksamkeit auf den Fehler im Programmchecker gezogen.

In [Wasserman 97] wird weiterhin ausgeführt, dass die Anforderung, dass `C` eine andere Berechnung durchführen muss als `P`, die Wahrscheinlichkeit für nicht erkannte Fehler reduziert. Diese Anforderung ist in der Definition von Programmcheckern in Kapitel 9.1 enthalten. Durch die unterschiedlichen Berechnungen ist die Wahrscheinlichkeit hoch, dass `P` und `C` unterschiedliche Fehler haben, so dass `C` die von `P` erkennt.

9.2.4 Abweichungen vom Paradigma

In der Literatur behandelte Probleme sind häufig mathematische, die sich besonders für Programmchecker eignen. Das Prinzip der Programmchecker kann jedoch auch für andere Probleme interessant sein. [Wasserman 97] enthält eine Liste von Ideen, wie es verallgemeinert werden kann. Einige von ihnen sollen nachfolgend wiedergegeben werden.

Zunächst können Programmchecker so implementiert werden, dass sie die Korrektheit der Ausgaben nur unvollständig prüfen. Tests können z. B. nur Fehler betreffen, die besonders wahrscheinlich auftreten oder solche, die für den Erfolg des Systems besonders kritisch sind. Ein Programmchecker könnte auch nur prüfen, ob Ein- und Ausgaben

im erlaubten Bereich liegen. Es könnte sogar soweit gegangen werden, dass nur die Ausführungszeit geprüft wird. Ist diese ungewöhnlich lang, deutet das auf Softwarefehler hin.

Ist die zu lange Ausführungszeit des Programmcheckers selbst ein Problem, können die Ausgaben des Programms auch nur stichprobenartig geprüft werden. Dann werden Fehler zwar nicht in jedem Fall gefunden. Tritt ein Fehler aber häufiger auf, ist es wahrscheinlich, dass er irgendwann erkannt wird. Um die Antwortzeiten des Systems nicht durch den Programmchecker zu verlängern, könnte dieser auch erst im nachhinein die Korrektheit der Ausgaben des Programms prüfen. Dazu müssen alle Ein- und Ausgaben des Programms paarweise gespeichert werden. Diese gespeicherten Paare können dann zu einem späteren Zeitpunkt durch den Programmchecker geprüft werden.

Es gibt auch Fälle, in denen die Ausführungszeit von Programmcheckern deutlich verbessert werden könnte, wenn er vom Programm zusätzliche Informationen bekommen würde, die dieses leicht ermitteln könnte. So wie das Programm spezifiziert ist, liefert es diese aber nicht, so dass der Programmchecker diese Informationen selbst ermitteln muss. Dann ist es möglich, das Programm so zu modifizieren, dass es diese zusätzlichen Informationen liefert. Als Beispiel wird in [Wasserman 97] ein Programmchecker für ein Sortierprogramm gegeben, das zusätzlich zur sortierten Ausgabe liefert, durch welche Permutation sich die Ausgabe aus der Eingabe ergibt. Diese Permutation zu bestimmen, ist für den Programmchecker sehr aufwendig. Sie zu prüfen ist jedoch sehr viel einfacher und das Sortierprogramm muss sie ohnehin bestimmen.

9.3 Automatische Validation von Webkomponenten

Meist stammen Anwendungen, für die Webkomponenten verwendet werden, nicht aus dem Bereich der Mathematik. Sie bilden nicht einfach eine Eingabe auf eine Ausgabe ab, wie die Programme in Kapitel 9.1. Daher können Programmchecker für sie nicht direkt angewendet werden. Die Idee der Programmchecker kann aber trotzdem übertragen werden, was nachfolgend erläutert werden soll.

9.3.1 Programmchecker für Webkomponenten

Am einfachsten lassen sich Programmchecker auf Webkomponenten anwenden, wenn einzelne Operationsaufrufe als Aufrufe von Programmen aufgefasst werden. Die Eingabe ist dann im Request der Operation enthalten, die Ausgabe im Response. Für jede Operation könnte ein Programmchecker implementiert werden, der sicherstellt, dass ihre Ergebnisse korrekt sind. Da ein Webservice mehrere Operationen enthält, könnten die Programmchecker als Bibliothek implementiert werden, wie in [Blum 93] und Kapitel 9.1 vorgestellt.

In Kapitel 6.1.3 wurde aber bereits ausgeführt, dass es auch Abhängigkeiten zwischen unterschiedlichen Operationsaufrufen gibt. Solche Abhängigkeiten gehören zur Schnittstelle zwischen Webclient und Webservice. Sie können durch Programmchecker, die nur einzelne Operationsaufrufe prüfen, aber nicht geprüft werden.

Daher muss die Idee der Programmchecker erweitert werden. Statt einzelner Operationsaufrufe, mit ihren Ein- und Ausgaben, muss der gesamte Kommunikationsprozess zwischen einem Webservice und seinen Webclients vom Programmchecker geprüft werden. Er enthält alle Ein- und Ausgaben aller Operationsaufrufe. Kommunikationsprozesse wurden bereits in den Kapiteln 7.1.3 bzw. 7.2.2 als Traces formalisiert. Auf ihnen beruht SXQT, das Spezifikationsverfahren dieser Arbeit. Damit müsste ein Programmchecker für einen Trace prüfen, ob dieser korrekt ist.

Für Programmchecker dieser Art soll nachfolgend der Begriff Validator verwendet werden. Validatoren sind unabhängig von einer bestimmten Implementierung eines Webservices oder Webclients. Sie beobachten je nach Sicht (siehe Kapitel 7.2.4) einen Webclient oder Webservice, zeichnen den Trace auf und prüfen, ob dieser bezogen auf die Spezifikation korrekt ist. Der Vorgang der Prüfung soll als automatische Validation bezeichnet werden. Das Attribut „automatisch“ weist dabei darauf hin, dass eine Validation auch von Menschen geleistet werden könnte, was dem Testen eines Webservices bzw. Webclients entspricht.

9.3.2 Automatische Validation von Prozessen

Der Begriff der Validation wird auch in [Cook 99] verwendet. Dort bezeichnet er den Vorgang der Prüfung, inwieweit ein Prozess einem Prozessmodell entspricht. Die Validation von Webservices kann davon als Spezialfall angesehen werden, bei dem geprüft wird, inwieweit der Kommunikationsprozess zwischen einem Webservice und seinen Webclients einem Prozessmodell entspricht.

In [Cook 99] wird nicht nur untersucht, ob der Prozess dem Prozessmodell entspricht. Ist das nicht der Fall, wird darüber hinaus ein quantitatives Maß für die Übereinstimmung angegeben. Das ist z. B. für Prozesse sinnvoll, an denen Menschen beteiligt sind, die sich nicht unbedingt an ein Prozessmodell halten. Ein Kommunikationsprozess zwischen einem Webservice und seinen Webclients sollte sich jedoch stets an das Prozessmodell halten, das die Schnittstelle zwischen ihnen im Sinne eines Vertrages beschreibt. Jede Abweichung von diesem kann zu schwerwiegenden Fehlern führen. Daher sind es auch „geringe“ Abweichungen nicht zu akzeptieren und es reicht festzustellen, ob überhaupt eine Abweichung auftritt. Daher soll in dieser Arbeit nur das betrachtet werden.

Ein grundsätzlicher Unterschied zwischen Programmcheckern und der Validation in [Cook 99] besteht darin, dass nicht für eine spezielle Funktion bzw. ein Prozessmodell ein spezieller Validator implementiert werden muss. Bei der Validation wird der Prozess mit einem Prozessmodell verglichen. Der Validator kann daher ein universelles Programm sein, das das Prozessmodell in einer geeigneten Repräsentation liest und dagegen den Vergleich durchführt.

Damit Validatoren und die zu prüfenden Programme nicht dieselben Fehler haben, muss für Validatoren sichergestellt sein, dass sie anders arbeiten als das zu prüfende Programm selbst. In Kapitel 9.1 wurde das analog für Programmchecker gefordert. Da Validatoren das Prozessmodell verwenden, dürfen die Programme dieses nicht ebenfalls ausführen. Das wäre grundsätzlich möglich, wenn Prozessmodelle wie in [Cook 99] zustandsbasierte Modelle, Petrinetze oder prozedurale oder regelbasierte Sprachen sind.

9.3.3 Automatische Validation von Webkomponenten

Auch das in Kapitel 7 entwickelte Spezifikationsverfahren SXQT kann als Prozessmodell angesehen werden, mit dem Kommunikationsprozesse zwischen einem Webservice und seinen Webclients beschrieben werden. Mit einer SXQT-Spezifikation wird festgelegt, welche SOAP-Nachrichten zwischen Webservice und Webclient ausgetauscht werden dürfen. Damit werden die erlaubten Kommunikationsprozesse beschrieben.

Daher soll SXQT als Prozessmodell für Kommunikationsprozesse von Webservices verwendet werden. Validatoren prüfen für beobachtete Traces, ob sie der SXQT-Spezifikation entsprechen. Hierzu lesen sie WSDL-Dokumente, die um SXQT-Ausdrücke erweitert wurden, wie in Kapitel 8 beschrieben.

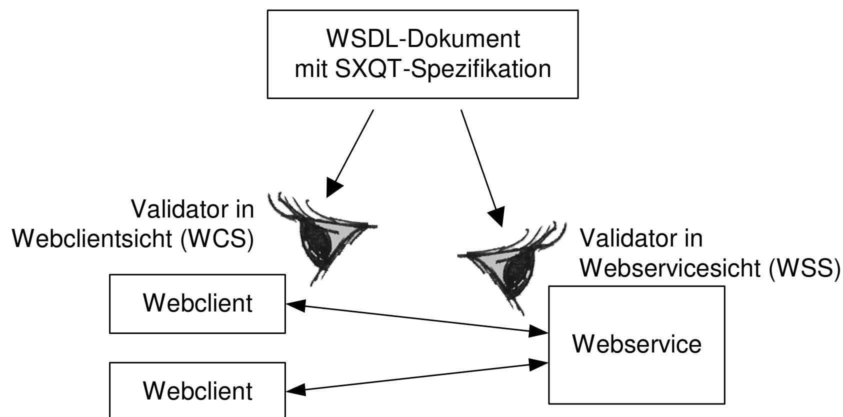

Abbildung 129: Validatoren für Webkomponenten

Abbildung 129 veranschaulicht, was ein Validator für Webkomponenten leistet. Er beobachtet je nach seiner Sicht (siehe Kapitel 7.2.4) welche SOAP-Nachrichten der Webservice bzw. einer seiner Webclients sendet oder empfängt. Er zeichnet die SOAP-Nachrichten als Trace auf und prüft, ob der Trace der SXQT-Spezifikation entspricht. Ist das nicht der Fall zeigt er einen Fehler an. Es sei festgestellt, dass ein Validator SOAP-Nachrichten weder verändert noch löscht oder eigene SOAP-Nachrichten sendet. Später in Kapitel 9.4.4 wird jedoch ein Validator mit Filter vorgestellt, der SOAP-Nachrichten löschen und durch neue SOAP-Nachrichten ersetzen kann.

Interessant ist die Beobachtung, dass Validatoren den Trace unvollständig prüfen können, wie es für Programmchecker in Kapitel 9.2.4 vorgeschlagen wurde. Mit SXQT-Ausdrücken werden nur die Anforderungen ausgedrückt, die spezifiziert werden sollen. Ein Validator, der eine SXQT-Spezifikation liest, kann nur diese Anforderungen prüfen. Je nachdem, wie vollständig die Spezifikation ist, prüft daher der Validator mehr oder weniger vollständig den Trace.

9.3.4 Korrektheit der Validation

Bei Programmcheckern war eine wichtige Frage, wie ihre Korrektheit sichergestellt werden kann. In Kapitel 9.2.3 wurde festgestellt, dass Programmchecker hierzu verifiziert oder getestet werden können. Da ein Validator Traces gegen eine gelesene Spezifikation prüft, hängt die Korrektheit der automatische Validation von der Korrektheit der Spezifikation ab. Hier könnte man sich auf den Standpunkt stellen, dass die Spezifikation per Definition korrekt ist. Kann angenommen werden, dass die Implementierung des Validators korrekt ist, würde dann ein Validator stets zuverlässig die Korrektheit eines Traces bestimmen können.

Auch wenn eine Spezifikation per Definition als korrekt angesehen wird, kann jedoch nicht sichergestellt werden, dass sie wirklich das gewünschte Verhalten beschreibt. Der Validator prüft somit nur, ob ein beobachteter Trace der Spezifikation entspricht, jedoch nicht, ob er im Sinne des gewünschten Verhaltens korrekt ist. Es kann daher auch zu den beiden in Kapitel 9.2.3 beschriebenen Fehlersituationen kommen, bei dem ein Fehler im Trace nicht erkannt wird oder bei dem mit einem falschen Alarm ein Fehler angezeigt wird, der nicht vorliegt.

Bei einem nicht erkannten Fehler entspricht der Trace also der Spezifikation, aber nicht dem erwarteten Verhalten. Diese Situation kann auch nicht durch Verifikation beseitigt werden, weil schon die Spezifikation fehlerhaft ist. Zu solchen Fehlern kann es auch durch die Unvollständigkeit einer Spezifikation kommen, wenn der fehlerhafte Aspekt im beobachteten Trace nicht als Anforderung beschrieben wurde.

Ein falscher Alarm ist wenig problematisch, wie schon in Kapitel 9.2.3 festgestellt. Durch ihn wird sichergestellt, dass der Fehler von Entwicklern untersucht wird. Dabei können nicht nur Fehler in den Implementierungen von Webservice bzw. Webclient gefunden werden, sondern auch solche in der Spezifikation. Sie können dann korrigiert werden. Ein falscher Alarm kann also darauf hinweisen, dass die Spezifikation nicht dem erwarteten Verhalten entspricht. Umgekehrt kann die Validation daher auch verwendet werden, um beim Ausbleiben von falschen Alarmen das Vertrauen zu erhöhen, dass die Spezifikation dem erwarteten Verhalten entspricht, wie in [Cook 99] festgestellt wird.

9.4 Anwendungen

Außer dass mit Hilfe der automatischen Validation von Webkomponenten das Vertrauen in die Spezifikation erhöht werden kann, gibt es weitere interessante Anwendungen. Diese sollen im Folgenden kurz angesprochen werden. Einige wurden ähnlich bereits für Programmchecker erwähnt und sollen hier auf die automatische Validation von Webkomponenten übertragen werden.

9.4.1 Finden von Fehlern in Implementierungen

So wurde in Kapitel 9.2.2 aus [Wasserman 97] wiedergegeben, dass Programmchecker für das Testen von Softwareprodukten verwendet werden können. Analog kann ein Validator auch für das Testen von Implementierungen von Webkomponenten eingesetzt werden. Hierzu werden unterschiedliche Kommunikationsprozesse zwischen einem Webservice sowie einem oder mehreren Webclients untersucht. Der Validator beobachtet welche Nachrichten der Webservice bzw. ein Webclient sendet oder empfängt. Für jeden beobachteten Trace wird dann geprüft, ob er der Spezifikation entspricht.

Ohne Validator wäre es die Aufgabe des menschlichen Testers zu entscheiden, ob sich die Komponenten korrekt verhalten. Bei komplexen Szenarien kann diese Entscheidung schwierig sein. Der Validator entlastet den Tester um diese Arbeit, so dass Fehler in Webservices, aber auch Webclients beim Testen leichter gefunden werden können.

Es ist aber nur möglich beim Testen eine begrenzte Zahl von Kommunikationsprozessen zu erproben. Daher werden in der Regel nicht alle in den Implementierungen enthaltenen Fehler gefunden und können im produktiven Betrieb der Webkomponente auftreten. Um sie zu finden, kann der Validator auch im produktiven Betrieb verwendet werden, wie es auch für Programmchecker in Kapitel 9.2.2 vorgeschlagen wurde.

Tritt dann im produktiven Betrieb eine Abweichung zwischen dem Kommunikationsprozess und der Spezifikation auf, kann das protokolliert werden. Da der Trace eine Formalisierung des Kommunikationsprozesses ist, sollte dieser hierzu gespeichert werden. In Kapitel 7.2.2 wurde vorgestellt, wie sich ein Trace zu einem XML-Dokument vervollständigen lässt. In dieser Form kann er gespeichert und später von Software-Wartungspersonal analysiert werden.

Der gespeicherte Trace kann nicht nur vom Wartungspersonal gelesen werden, um daraus die Fehlersituation zu erkennen. Der Trace könnte auch „wiederabgespielt" werden. Kann angenommen werden, dass sich in der Implementierung des Webservices ein Fehler befindet, können alle Requests aus dem Trace erneut an den Webservice gesendet werden. Die Implementierung des Webservices kann dabei im Debugger ausgeführt und so im Programmcode der Fehler gesucht werden. Nach einer Korrektur kann auf diese Art auch untersucht werden, ob der Verstoß gegen die Spezifikation nicht mehr auftritt.

Tritt eine Abweichung zwischen dem Kommunikationsprozess und der Spezifikation auf, kann neben dem Abspeichern des Traces auch dem Benutzer angezeigt werden, dass ein Fehler aufgetreten ist. Damit ist er gewarnt, dass es im System ein Problem gibt. Für Programmchecker wurde das in [Wasserman 97] und Kapitel 9.2.2 beschrieben.

Es muss beachtet werden, dass die Prüfung, ob der beobachtete Trace der Spezifikation entspricht, Rechenzeit kostet. Daher kann die Antwortzeit der Webkomponente reduzieren werden. Um das für Programmchecker zu vermeiden, wurde in Kapitel 9.2.4 die Idee von [Wasserman 97] wiedergegeben, dass zunächst nur die Ein- und Ausgaben gespeichert werden und diese erst später durch einen Programmchecker geprüft werden. Diese Vorgehensweise lässt sich analog auch für Validatoren anwenden. Während des produktiven Betriebs wird dazu nur der Trace aufgezeichnet und als XML-Dokument gespeichert werden. Erst später prüft dann der Validator, ob der Trace der Spezifikation entspricht.

9.4.2 Prüfen des Verständnisses über Schnittstelle

Hauptaufgabe beim Testen von Webkomponenten ist zu prüfen, ob deren Implementierungen fehlerhaft sind. Abweichungen von der Spezifikation können jedoch nicht nur durch Implementierungsfehler auftreten, sondern auch durch Missverständnisse über die Schnittstelle, wie bereits in Kapitel 9.2.2 aus [Wasserman 97] wiedergegeben. Auch in solchen Fällen findet der Validator die Abweichung von der Spezifikation. Entwickler müssen der Abweichung dann nachgehen und gegebenenfalls ihr Verständnis über die Schnittstelle anpassen[60].

Interessant ist, dass es zum Erkennen von Missverständnissen über die Schnittstelle nicht erforderlich ist, dass die Spezifikation vom Webservice bereitgestellt wird. Typischerweise stellt der Webservice ein WSDL-Dokument zur Verfügung, das der Webclient über das Internet kopieren kann. Es kann, wie in Kapitel 8 beschrieben, um SXQT-Ausdrücke erweitert sein. Ist das nicht der Fall, können auch die Entwickler von Webclients ihr Verständnis über die Schnittstelle in dieser Form formulieren.

Hierzu kopieren sie das nicht erweiterte WSDL-Dokument vom Webservice oder erstellen es selbst. Ihr Verständnis über die Schnittstelle drücken sie mit Hilfe von SXQT-Ausdrücken aus und fügen diese dem WSDL-Dokument zu. Gegen die so erstellte SXQT-Spezifikation kann der Validator beobachtete Traces validieren. Tritt eine Abweichung zwischen Spezifikation und Trace auf, kann das auf ein Missverständnis über die Schnittstelle des Webservices hindeuten. Diese Vorgehensweise ist der in Kapitel 9.2.2 aus [Wasserman 97] wiedergegeben ähnlich, bei der Entwickler, die eine Komponente anderer Entwickler verwenden, für diese einen Programmchecker erstellen.

Wird auf diese Art validiert, wird der Validator häufig nicht die Möglichkeit haben, den Webservice zu beobachten. Dieser läuft möglicherweise in einem anderen Unternehmen, die Entwickler des Webclients können auf ihn nur über das Internet zugreifen. Dann kann der Validator nur den Webclient beobachten und dessen gesendete und empfangene SOAP-Nachrichten im Trace aufzeichnen. Der Validator hat also die Client-

[60] Neben Implementierungsfehlern und Missverständnissen über die Schnittstelle kann natürlich auch ein Fehler in der Spezifikation zu einer Abweichung zwischen Trace und Spezifikation führen, wie bereits in Kapitel 9.3.4 erwähnt. Außerdem kommen auch Hardware- und Betriebssystemfehler in Betracht.

sicht. In Kapitel 7.3.4 wurde festgestellt, dass nicht jeder SXQT-Ausdruck in der Clientsicht verwendet werden kann. Da der Validator hier jedoch die Clientsicht hat, dürfen die Entwickler des Webclients ihr Verständnis nur mit solchen SXQT-Ausdrücken formalisieren.

9.4.3 Erkennen relevanter Änderungen der Schnittstelle

In Kapitel 9.2.2 wurde aus [Wasserman 97] wiedergegeben, dass Programmchecker nur von der Ein- und Ausgabe des zu prüfenden Programms abhängen. Daher konnten sie auch für neue Versionen des Programms sofort eingesetzt werden, auch wenn dieses andere Algorithmen verwendet. Auch der Validator und die Spezifikation sind von der Implementierung der Webkomponenten unabhängig. Daher können auch sie ohne Änderungen zum Testen neuer Implementierungen eingesetzt werden.

Interessant ist, dass diese Tatsache auch verwendet werden kann, um Schnittstellen mit relevanten Änderungen zu erkennen. Sie zu erkennen ist wichtig, weil die Entwicklung verschiedener Webkomponenten unabhängig voneinander in unterschiedlichen Unternehmen stattfinden kann, wie in Kapitel 2.7 festgestellt. Entwickler eines Webclients können daher nicht sicher sein, dass ein verwendeter Webservice nicht nach einiger Zeit geändert wird. Solche Änderungen sind jedoch wichtig, wenn sie seine Schnittstelle bzw. deren Spezifikation betreffen.

Es wäre nicht schwierig zu erkennen, dass sich die Spezifikation eines Webservices geändert hat. Die aktuell gültige Spezifikation wird stets vom Webservice zur Verfügung gestellt und kann von diesem über das Internet kopiert werden. Bei der Entwicklung eines Webclients kann daher die Spezifikation kopiert und später mit der aktuell gültigen Version byteweise verglichen werden. Jede Änderung der Spezifikation könnte so erkannt werden.

Ein Webclient kann jedoch auch mit einer neuen Version des Webservices interoperabel sein, wenn sich die Spezifikation geändert hat. So stört es z. B. wenig, wenn dem Webservice eine Operation zugefügt, eine Anforderung an einen Request gelockert oder die an einen Response verschärft wurde. Leider ist es jedoch schwierig zu entscheiden, ob ein Webclient, der für eine ältere Spezifikation des Webservices entwickelt wurde, auch noch mit einer neuen verwendbar ist. Wie das bewiesen werden kann, wird allgemein für Schnittstellen z. B. in [Kreuz 99] untersucht.

Obwohl die Entscheidung allgemein schwierig ist, kann jedoch leicht mit Hilfe der automatischen Validation erkannt werden, ob ausgetauschte SOAP-Nachrichten noch den Annahmen entsprechen, unter denen der Webclient entwickelt wurde. Diese Annahmen sind in der Spezifikation enthalten, die zum Zeitpunkt der Erstellung des Webclients vom Webservice kopiert werden konnte. Alternativ dazu können die Entwickler des Webclients ihre Annahmen auch selbst in Form einer Spezifikation ausdrücken, wie in Kapitel 9.4.2 vorgeschlagen.

In beiden Fällen wird die Spezifikation auf Seite des Webclients gespeichert und gegen sie validiert, um die Entscheidung zu treffen. Wird dann die Implementierung des Webservices geändert, kann sich auch dessen Spezifikation ändern. Das ändert aber nicht die auf Seite des Webclients gespeicherte Version der Spezifikation, gegen die der Webclient implementiert wurde und gegen die validiert wird. Nur wenn SOAP-Nachrichten ausgetauscht werden, die dieser Version widersprechen, wird das als Verstoß gegen die Spezifikation erkannt. Nur dann gibt es aus Sicht des Webclients eine für ihn relevante Änderung der Schnittstelle. Ansonsten können Webclient und Webservice trotz geänderter Spezifikation weiter zusammen verwendet werden.

9.4.4 Schutz der Webkomponenten vor böswilligen Zugriffen

Für die bisherigen Anwendungen reicht ein Validator, der gesendete und empfangene SOAP-Nachrichten nur beobachtet aber nicht verändert, wie in Kapitel 9.3.3 beschrieben. Zwei weitere Anwendungen ergeben sich, wenn der Validator zusätzlich eine Filterfunktion hat, mit der eine SOAP-Nachricht, die nicht der Spezifikation entspricht, vernichtet und gegebenenfalls durch eine andere SOAP-Nachricht ersetzt werden kann. Dann kann sichergestellt werden, dass einen Webservice bzw. Webclient nur der Spezifikation entsprechende SOAP-Nachrichten erreichen.

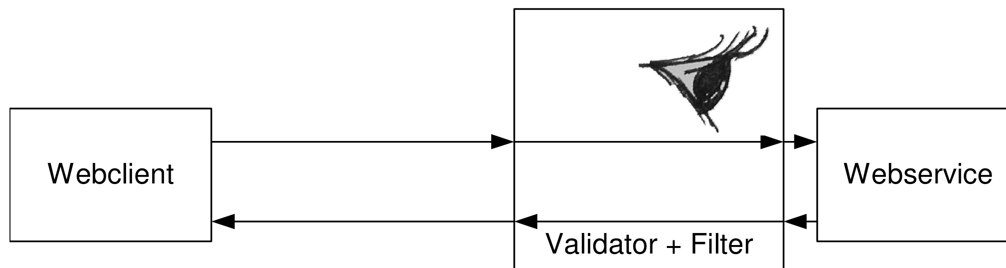

Abbildung 130: Validator mit Filter

Abbildung 130 veranschaulicht einen solchen Validator mit Filter. Im Unterschied zu einem Validator ohne Filter wird nicht nur eine Möglichkeit benötigt, die SOAP-Nachrichten zu beobachten, so dass von ihnen eine Kopie im Trace gespeichert werden kann. Vielmehr ist es erforderlich, dass der Validator die SOAP-Nachricht zunächst empfängt und validiert, aber nicht an den SOAP-Empfänger weiterleitet. Das darf erst geschehen, wenn erkannt ist, dass die SOAP-Nachricht der Spezifikation entspricht.

Als erste Anwendung für einen Validator mit Filter soll betrachtet werden, wie ein Webservice oder Webclient vor böswilligen Zugriffen geschützt werden kann. Eine Möglichkeit, über das Internet in einen Server einzudringen, besteht darin, zu ihm eine Verbindung aufzubauen und ihm ungültige Nachrichten zu senden. Die Implementierung des Servers sollte prüfen, ob Nachrichten gültig sind und diese sonst ablehnen. Wird diese Prüfung jedoch für eine bestimmte Art fehlerhafter Nachrichten vergessen, kann es geschehen, dass sich der Server auf eine ungewollte Art verhält oder auch „abstürzt“. In einigen Fällen lässt sich die Nachricht auch so konstruieren, dass in ihr enthaltener Programmcode ausgeführt wird, was zum „Einbruch“ in den Server verwendet werden kann.

Diese Art des Einbruchs könnte auch bei Webservices oder Webclients versucht werden. Insbesondere Webservices sind gefährdet, weil sie im Internet frei zugänglich sind. Webclients sind weniger gefährdet, weil sie es sind, die Verbindungen zum Webservice aufbauen. Ein böswilliger Nutzer müsste also erst einen Webclient dazu bringen, zu ihm eine Verbindung aufzubauen, bevor er als Response eine ungültige SOAP-Nachricht senden kann. Daher sollen nachfolgend vor allem Webservices betrachtet werden.

Die Idee, einen Validator mit Filter zum Schutz von Webservices zu verwenden, besteht darin, ihn die Prüfung auf ungültige SOAP-Nachrichten durchführen zu lassen. Alle an den Webservice gesendeten SOAP-Nachrichten werden zunächst vom Validator geprüft. Nur wenn sie der Spezifikation entsprechen, werden sie an den Webservice weitergeleitet. Entspricht eine SOAP-Nachricht nicht der Spezifikation, wird statt dessen eine Fehlernachricht an den Webclient zurückgesendet. Das ist erforderlich, weil sonst der Response des Request-/Response-Paares fehlen würde.

Die Spezifikation muss so konstruiert sein, dass sie nur SOAP-Nachrichten erlaubt, die den Webservice nicht gefährden können. Z. B. muss in der Spezifikation angegeben sein, wie lang Zeichenketten sein dürfen, damit in der Implementierung des Webservices für diese genügend Speicherplatz reserviert werden kann.

Werden in der Spezifikation alle solchen Anforderungen ausgedrückt, können fehlerhafte SOAP-Nachrichten den Webservice nicht erreichen. Damit kann der Webservice zuverlässiger vor böswilligen Zugriffen geschützt werden als durch Prüfungen in der Implementierung des Webservices selbst, weil in kompakter Form explizit deklariert werden kann, welche SOAP-Nachrichten erlaubt sind, statt das implizit im Programmcode festzulegen. Gleichzeitig wird durch diese Vorgehensweise die Entwicklung von Webservices vereinfacht, weil solche Prüfungen dann nicht mehr implementiert werden müssen.

9.4.5 Verbergen von Spezifikationsverstößen von Webkomponenten

Als zweite Anwendung des Validators mit Filter soll hier diskutiert werden, wie mit ihm das Verhalten eines Webclients oder Webservices so korrigiert werden kann, dass der Kommunikationspartner nur ein der Spezifikation entsprechendes Verhalten beobachten kann. Das hat Ähnlichkeiten zur Idee der selbstkorrigierenden Programme aus [Blum 93], die bereits in Kapitel 9.1 vorgestellt wurde.

Anders als selbstkorrigierende Programme dürfen Validatoren keine Operationen im Webservice aufrufen. Ein selbstkorrigierendes Programm tut das, um doch noch ein korrektes Ergebnis berechnen zu können. Dabei wird vorausgesetzt, dass die Programme seiteneffektfrei sind, also neben dem Berechnen der Ausgabe nichts weiteres bewirkt wird. Diese Annahme kann bei Webservices oder Webclients nicht gelten. Z. B. werden bei der Webkomponente für eine Internetzeitung, die in Kapitel 2.6 vorgestellt wurde, mit Hilfe von Operationsaufrufen Artikel gespeichert oder auch Sperren gesetzt. Das sind Seiteneffekte. Daher kann die Vorgehensweise selbstkorrigierender Programme nicht direkt auf Webservices übertragen werden[61].

Ein Validator mit Filter kann jedoch dafür sorgen, dass ein Kommunikationspartner nie eine nicht der Spezifikation entsprechende SOAP-Nachricht empfängt. Hierzu werden solche SOAP-Nachrichten vom Validator vernichtet und gegebenenfalls durch eine SOAP-Fehlernachricht ersetzt, die die Fehlersituation anzeigt.

Um einen Webservice auf diese Art „zu korrigieren“, prüft der Validator mit Filter alle Responses des Webservices. Weicht ein Response von der Spezifikation ab, wird er vernichtet und eine SOAP-Fehlernachricht an den Webclient geschickt, die diesem die Fehlersituation anzeigt. Zwar ist dann die Operation aus Sicht des Webclients fehlgeschlagen, der Webclient kann jedoch kein Verhalten beobachten, dass der Spezifikation widerspricht.

Analog lässt sich auch ein Webclient „korrigieren“. Versucht dieser einen Request an den Webservice zu senden, der nicht der Spezifikation entspricht, wird der Request vom Validator mit Filter vernichtet. So kann der Webservice nie einen nicht der Spezifikation entsprechenden Request beobachten. Auch hier sollte der Validator mit Filter dem Webclient eine SOAP-Fehlernachricht zurücksenden, damit dieser über die Fehlersituation informiert ist.

---

[61] In Kapitel 7.7.6 wurde eine Ausnahme vorgestellt, bei der der Validator Operationen eines Webservices aufruft. Hierzu wird in XQuery die Funktion `fn:document` verwendet. Mit ihr können jedoch nur Operationen aufgerufen werden, die mit dem SOAP-Response-Kommunikationsmuster realisiert sind. Für solche Operationen ist die Seiteneffektfreiheit garantiert.

9.5 Architektur mit Validatoren

Um die beschriebenen Anwendungen zu realisieren, müssen die Validatoren so in die Architektur der Webkomponenten eingebettet werden, dass sie die gesendeten und empfangenen SOAP-Nachrichten beobachten können. Außerdem muss festgelegt werden, auf welchem Rechner und in welchem Prozess sie ausgeführt werden. Verschiedene Möglichkeiten sollen nachfolgend vorgestellt werden.

9.5.1 Zuordnung zu Webclient oder Webservice

Zunächst stellt sich die Frage, ob der Validator dem Webclient oder dem Webservice zugeordnet wird, wie in Abbildung 131 veranschaulicht. Die Frage ist wichtig, weil Webclient und Webservice miteinander nur über das Internet verbunden sind und möglicherweise weit voneinander entfernt in unterschiedlichen Unternehmen ausgeführt werden. Ohne weitere Maßnahmen steht das Ergebnis der automatischen Validation nur auf der Seite zur Verfügung, auf der die Validation ausgeführt wird.

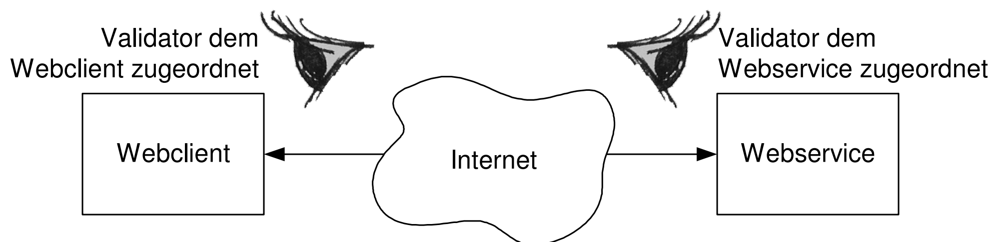

Abbildung 131: Zuordnung des Validators zu Webclient bzw. Webservice

Mit der Frage der Zuordnung eng verbunden ist auch die in Kapitel 7.2.4 vorgestellte Sicht. Der Validator enthält den Beobachter, der gesendete und empfangene SOAP-Nachrichten im Trace aufzeichnet. Ist er einem Webclient zugeordnet, kann er in der Regel nur die SOAP-Nachrichten beobachten, die mit diesem Webclient ausgetauscht wurden. Wird der Validator dagegen dem Webservice zugeordnet, ist es möglich alle mit dem Webservice ausgetauschten SOAP-Nachrichten zu beobachten, also die von allen Webclients. Nach dieser Überlegung wäre es naheliegend den Validator dem Webclient zuzuordnen, wenn die Webclientsicht benötigt wird und dem Webservice für die Webservicesicht.

Welche Zuordnung gewählt werden sollte, kann auch davon abhängen, was durch die automatische Validation erreicht werden soll. Zum Teil wurde das bereits im Kapitel 9.4 angesprochen. So wurde z. B. in Kapitel 9.4.2 vorgestellt, wie Entwickler von Webclients ihr Verständnis über einen Webservice als Spezifikation formulieren und durch die automatische Validation überprüfen. Dazu wird ein Validator benötigt, der dem Webclient zugeordnet ist. Ebenso musste in Kapitel 9.4.3 der Validator dem Webclient zugeordnet sein, um eine relevante Änderung der Schnittstelle durch den Webservice zu erkennen.

Um einen Webservice vor böswilligen Zugriffen zu schützen, wie in Kapitel 9.4.4 beschrieben, muss dagegen offensichtlich der Validator dem Webservice zugeordnet sein. Der Seite des Webclients kann nicht vertraut werden, weil von ihr der böswillige Zugriff ausgeht.

Zum Verbergen von Spezifikationsverstößen aus Kapitel 9.4.5 hängt die Zuordnung davon ab, ob Spezifikationsverstöße des Webclients oder des Webservices verborgen werden sollen. Dieser Seite muss der Validator zugeordnet werden, weil der Kommunikationspartner den Verstoß nicht beobachten können soll.

9.5.2 Zuordnung zu Prozess

Auf der zugeordneten Seite muss der Validator in einem Prozess ausgeführt werden. Er kann entweder in einem eigenen Prozess oder im Prozesses der SOAP-Implementierung des Webclients bzw. Webservices enthalten sein. Beides soll nachfolgend erläutert werden.

9.5.2.1 Eigener Prozess

Befindet sich der Validator in einem eigenen Prozess, stellt sich die Frage, wie er die von der SOAP-Implementierung gesendeten und empfangenen SOAP-Nachrichten beobachten kann. Das ist ohne Änderung der SOAP-Implementierung möglich, wenn der Validator als HTTP-Proxy implementiert ist. Ein HTTP-Proxy ist ein Vermittler, der HTTP-Requests von HTTP-Clients an HTTP-Server weiterleitet und HTTP-Responses in umgekehrter Richtung. Dabei kann der HTTP-Proxy die HTTP-Nachrichten verändern oder vernichten. Mehr über HTTP-Proxies findet man im Standard von HTTP 1.1 [Fielding 99] .

Abbildung 132: HTTP-Proxy zwischen HTTP-Client und HTTP-Server

Da SOAP-Nachrichten über HTTP ausgetauscht werden, können sie auch von HTTP-Proxies weitergeleitet werden. Ist der Validator ein HTTP-Proxy, kann er beim Weiterleiten auf die SOAP-Nachrichten zugreifen. Aus Sicht des Webclients ist der Validator dann der Webservice. Daher baut der Webclient zu ihm die HTTP-Verbindung auf und sendet den Request. Der Validator speichert diesen im Trace und validiert ihn gegen die Spezifikation. Außerdem baut er selbst eine HTTP-Verbindung zum Webservice auf und sendet über diese den Request an den Webservice weiter. Der Webservice verarbeitet den Request und antwortet auf der gleichen HTTP-Verbindung mit einem Response. So erhält ihn der Validator, der ihn dem Trace zufügt, validiert und an den Webclient weiterleitet.

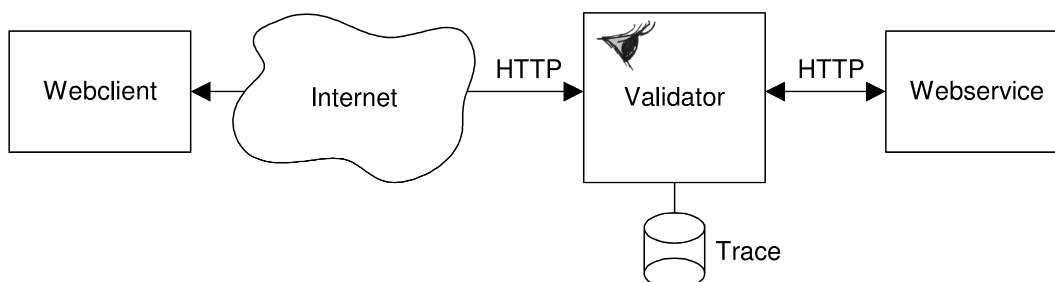

Abbildung 133: Validator als HTTP-Proxy, dem Webservice zugeordnet

Abbildung 133 zeigt schematisch, wie die Kommunikation über den Validator abläuft. Beispielhaft ist er dem Webservice zugeordnet. Dazu ist dargestellt, dass sich das Internet zwischen Webclient und Validator befindet. Validator und Webservice können sich auf demselben Rechner befinden. Sie können aber auch auf unterschiedlichen Rechnern installiert sein, da sie miteinander über HTTP kommunizieren.

Realisiert als HTTP-Proxy ist der Validator von der SOAP-Implementierung von Webservice und Webclient völlig unabhängig. Ohne Änderung kann er daher auch auf Seite des Webclients verwendet werden. Dann findet die Kommunikation über das Internet zwischen Validator und Webservice statt.

Als HTTP-Proxy realisierte Validatoren haben Ähnlichkeiten mit den in Kapitel 3.1 eingeführten SOAP-Intermediaries. Beide leiten SOAP-Nachrichten weiter. An einen SOAP-Intermediary können jedoch Headereinträge in der SOAP-Nachricht adressiert werden. Diese werden von ihm verarbeitet. Aufgabe des Validators ist jedoch nicht die Verarbeitung von Teilen der SOAP-Nachrichten. Er soll die vollständige SOAP-Nachricht prüfen und aus Sicht von Webclient und Webservice transparent sein. Daher soll der Validator in dieser Arbeit nicht als SOAP-Intermediary angesehen werden.

9.5.2.2 Prozess der SOAP-Implementierung

Statt den Validator in einem eigenen Prozess auszuführen, kann er auch direkt der SOAP-Implementierung von Webclient oder Webservice zugefügt werden, so dass beide im selben Prozess ausgeführt werden. In Abbildung 134 ist das wieder für den Fall veranschaulicht, in dem er dem Webservice zugeordnet ist.

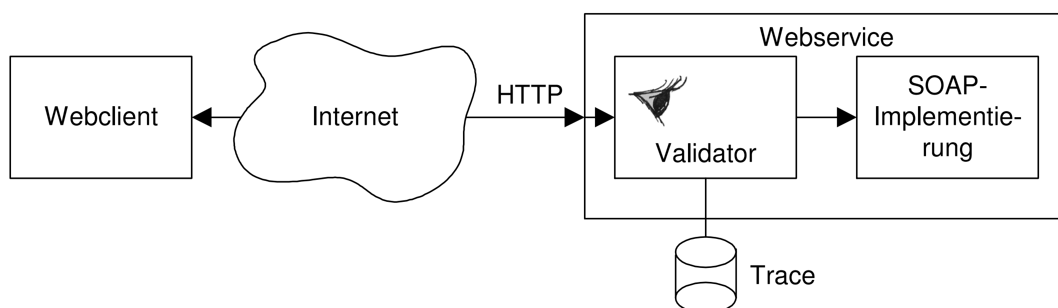

Abbildung 134: Validator in SOAP-Implementierung, dem Webservice zugeordnet

Die Tatsache, dass sich Validator und SOAP-Implementierung im selben Prozess befinden, vereinfacht es, die SOAP-Nachrichten zu beobachten. SOAP-Nachrichten können direkt von der SOAP-Implementierung empfangen werden. Innerhalb des Prozesses lässt sich dann leicht eine interne Repräsentation der empfangenen SOAP-Nachricht an den Validator weiterreichen, der sie dem Trace zufügt und validiert. Die gleiche interne Repräsentation kann außerdem von der SOAP-Implementierung selbst verarbeitet werden. Ebenso können zu sendende SOAP-Nachrichten neben dem Senden auch an den Validator weitergegeben werden. Hierzu muss die Implementierung des Webservice bzw. des Webclients selbstverständlich geändert werden.

9.5.2.3 Vergleich

Den Validator als HTTP-Proxy oder im Prozess der SOAP-Implementierung zu realisieren, hat unterschiedliche Eigenschaften. Zunächst fällt auf, dass eine Realisierung als HTTP-Proxy von der SOAP-Implementierung vollständig unabhängig ist. Validator und SOAP-Implementierung kommunizieren miteinander nur über das Internet. Das ist unabhängig von der Art ihrer Implementierungen möglich. Wird der Validator dagegen im Prozess der SOAP-Implementierung realisiert, ist er viel stärker von dieser abhängig. Zumindest wird ein Mechanismus benötigt, mit dem die SOAP-Nachrichten im Prozess weitergereicht werden. Darüber hinaus ist es zumindest zweckmäßig, dass Validator und SOAP-Implementierung auf die gleiche Art implementiert sind, z. B. mit der gleichen Komponentenarchitektur.

Die Unabhängigkeit ist ein Vorteil der Realisierung als HTTP-Proxy. Der Validator kann zusammen mit beliebigen SOAP-Implementierungen verwendet werden. Diese Flexibilität fehlt Realisierungen im Prozess der SOAP-Implementierung, bei der für unterschiedliche SOAP-Implementierungen unterschiedliche Validatoren implementiert werden müssen.

Auf der anderen Seite verspricht eine Realisierung im Prozess der SOAP-Implementierung eine bessere Effizienz. SOAP-Nachrichten werden nur einmal empfangen und durch Parsen in eine interne Repräsentation gebracht. Innerhalb eines Prozesses kann dann die interne Repräsentation sowohl an den Validator als auch an die SOAP-Implementierung gegeben werden. Bei Verwendung eines HTTP-Proxy wird dagegen die SOAP-Nachricht sowohl im Validator als auch in der SOAP-Implementierung durch Parsen in eine interne Repräsentation gebracht, was zusätzlichen Aufwand verursacht. Außerdem muss für einen Operationsaufruf eine zusätzliche HTTP-Verbindung zwischen Validator und SOAP-Implementierung aufgebaut werden. Der Aufbau und die Übertragung der SOAP-Nachrichten über diese Verbindung vergrößert die Antwortzeit der Webkomponente.

Um die Antwortzeiten bei einer Realisierung als HTTP-Proxy zu verbessern, kann dieser auf einem anderen Rechner als die SOAP-Implementierung ausgeführt werden, da beide ohnehin über eine Netzwerkverbindung miteinander kommunizieren. Bei einer Realisierung im Prozess der SOAP-Implementierung würde eine solche Verteilung zusätzliche Maßnahmen erfordern.

Schließlich hat eine Realisierung als HTTP-Proxy noch den Vorteil, dass dann Validator und SOAP-Implementierung gegeneinander geschützt sind, weil sie sich in unterschiedlichen Prozessen befinden. So kann z. B. vermieden werden, dass die SOAP-Implementierung durch Softwarefehler auf Datenstrukturen des Validators zugreift und dadurch dessen korrektes Funktionieren gefährdet. Das wäre bei einer Realisierung des Validators im Prozess der SOAP-Implementierung möglich.

Zusammenfassend sei festgestellt, dass eine Realisierung des Validators innerhalb der SOAP-Implementierung vorzuziehen ist, wenn größere Effizienz und gute Antwortzeiten der Webkomponenten im Vordergrund stehen. Für eine flexible Realisierung, die zusammen mit beliebigen SOAP-Implementierungen verwendet werden kann, ist jedoch die Realisierung als HTTP-Proxy vorzuziehen. Dabei ist vor allem der Schutz des Validators gegen die SOAP-Implementierung ein weiterer Vorteil.

9.5.3 Zeitpunkt der Validation

Für die bisher gezeigten Realisierungen wurde angenommen, dass die automatische Validation sofort nach der Beobachtung einer SOAP-Nachricht stattfindet. Die hierzu erforderliche Rechenzeit kann die Antwortzeit der Webkomponente verschlechtern, vor allem, wenn sich Validator und SOAP-Implementierung auf dem gleichen Rechner befinden. Dann wird beides von den gleichen Prozessoren ausgeführt. Daher sollen nachfolgend zwei Möglichkeiten untersucht werden, wie durch eine spätere Validation bessere Antwortzeiten erreicht werden können.

9.5.3.1 Manueller Start

In Kapitel 9.4.1 wurde festgestellt, dass die Antwortzeiten verbessert werden können, wenn der Trace zunächst nur aufgezeichnet und erst später validiert wird. So kann die Validation z. B. für einen im produktiven Betrieb aufgezeichneten Trace erst manuell durch Wartungspersonal gestartet werden, wenn mit dem System ein Problem aufgetreten ist.

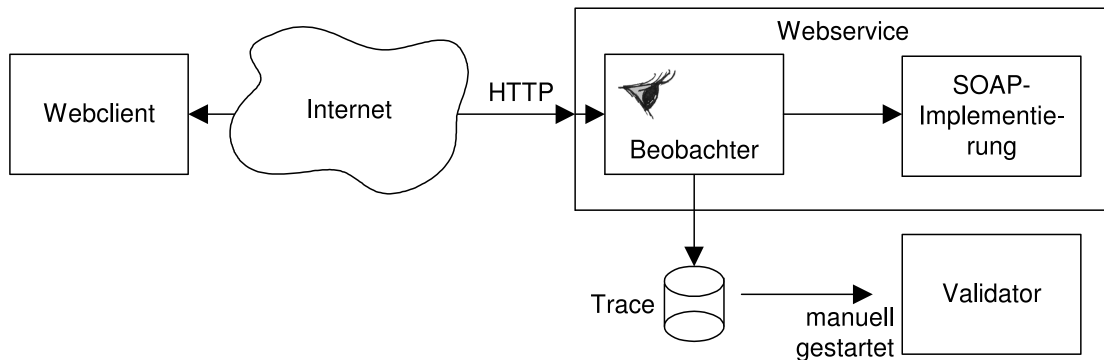

Abbildung 135: Manuell gestarteter Validator mit Beobachter in Implementierung des Webservices

Abbildung 135 zeigt, was hierzu an der Architektur geändert werden muss. Statt dem Validator muss im produktiven Betrieb nur ein Beobachter die SOAP-Nachrichten beobachten und dem Trace zufügen. Eine Validation gegen die Spezifikation findet zunächst nicht statt. Das geschieht erst, wenn der Validator für den Trace manuell gestartet wird. In Abbildung 135 ist der Beobachter innerhalb des Prozesses der SOAP-Implementierung realisiert. Ebenso könnte er in einem eigenen Prozess als HTTP-Proxy realisiert sein.

Wie die anderen Realisierungsarten kann auch diese analog für Validatoren verwendet werden, die dem Webclient zugeordnet sind. Dann befindet sich der Beobachter auf dessen Seite. Eine manuell gestartete Validation erlaubt aber keine Validatoren mit Filter, weil dazu eine SOAP-Nachricht validiert werden muss, bevor sie weitergeleitet wird.

9.5.3.2 Warteschlange für SOAP-Nachrichten

Ein Validator mit Filter lässt sich auch mit der folgenden Realisierung nicht erstellen, weil auch hier jede SOAP-Nachricht zunächst nur von einem Beobachter gespeichert und erst später validiert wird. Gespeichert wird sie hier jedoch in einer Warteschlange, aus der sie der Validator entnimmt. Erst der Validator fügt sie dem Trace zu und validiert sie. Das ist in Abbildung 136 veranschaulicht, wobei exemplarisch der Validator wieder dem Webservice zugeordnet ist und der Beobachter im Prozess der SOAP-Implementierung ausgeführt wird.

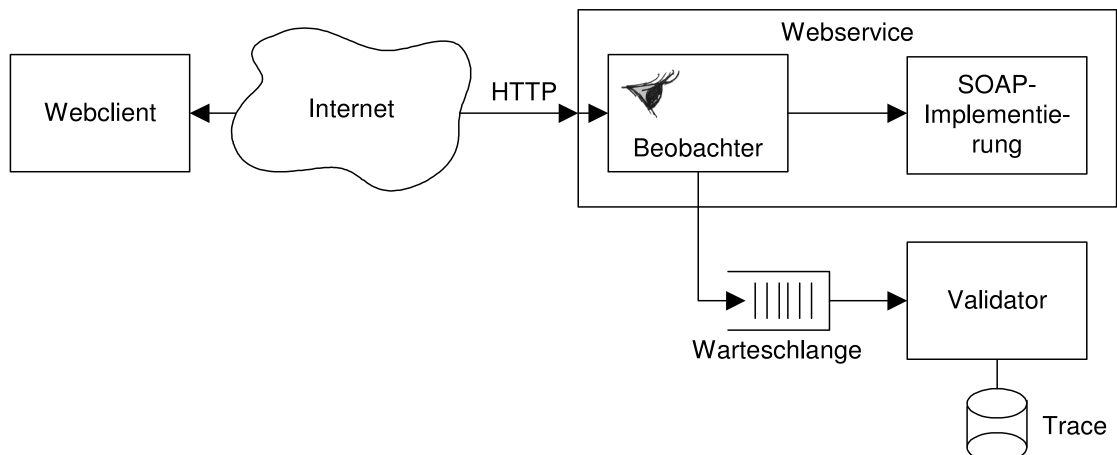

Abbildung 136: Validator mit Warteschlange und Beobachter in Implementierung des Webservices

Mit dieser Realisierung muss die Validation nicht manuell gestartet werden. Verstöße gegen die Spezifikation können also automatisch angezeigt werden. Trotzdem wird die Antwortzeit des Webservices nicht deutlich verschlechtert, insbesondere wenn sich der Validator auf einem anderen Rechner befindet als die SOAP-Implementierung.

9.5.3.3 Vergleich

Je nachdem, wann die Validation ausgeführt wird, ergeben sich unterschiedliche Eigenschaften. Ob die automatische Validation stets sofort ausgeführt werden muss oder erst durch eine Warteschlange gekoppelt später oder ob sie manuell gestartet werden kann, hängt auch davon ab, welche der Kapitel 9.4 vorgestellten Anwendung realisiert werden soll.

So lässt sich ein Validator mit Filter nur realisieren, wenn sofort beim Empfang einer SOAP-Nachricht validiert wird. Erst wenn für die SOAP-Nachricht erkannt wurde, dass sie der Spezifikation entspricht, darf sie weitergeleitet werden. Damit können Anwendungen, die einen Validator mit Filter benötigen, nicht realisiert werden, wenn erst durch eine Warteschlange gekoppelt validiert wird oder die Validation später manuell gestartet wird. Das trifft auf die Anwendung aus Kapitel 9.4.4 und 9.4.5 zu, bei der Webkomponenten mit einem Validator vor böswilligen Zugriffen geschützt bzw. Spezifikationsverstöße von Webkomponenten verborgen werden.

Die drei Realisierungsmöglichkeiten unterscheiden sich auch in der Frage, inwieweit die Antwortzeiten von Webkomponenten durch Zufügen des Validators erhöht werden. Die Antwortzeiten erhöhen sich vor allem, wenn jede SOAP-Nachricht sofort validiert wird, weil dann durch die Validation das System belastet wird, dass die SOAP-Nachricht verarbeitet oder in Falle des HTTP-Proxy zumindest weiterleitet. Insbesondere wird selbstverständlich die Antwortzeit erhöht, wenn beim Validator mit Filter die SOAP-Nachricht erst nach Abschluss der Validation weitergeleitet wird.

Wird dagegen eine SOAP-Nachricht zunächst nur beobachtet und in eine Warteschlange gestellt, werden die Antwortzeiten weniger beeinflusst. Der Validator kann die SOAP-Nachrichten nach und nach validieren. Sollten Lastspitzen auftreten, bei denen mehr SOAP-Nachrichten beobachtet werden, als er validieren kann, wächst lediglich die Länge der Warteschlange. Eine lange Warteschlange verschlechtert jedoch nicht die Antwortzeit der Webkomponente. Ob diese überhaupt durch den Aufwand für die Validation beeinflusst wird, hängt dann davon ab, ob Validator und Webkomponente auf demselben Rechner ausgeführt werden. Das muss nicht der Fall sein. Zugriff auf Warteschlangen ist vielfach auch über Rechnergrenzen hinweg möglich.

Die beste Effizienz jedoch kann erreicht werden, wenn die Validation manuell gestartet wird. Offensichtlich lässt sich dabei gezielt steuern, wann die Validation stattfindet und auf welchem Rechner. Während eines produktiven Betriebes wird überhaupt nicht validiert und kostet daher auch keine Rechenzeit. Die Validation kann später im Labor des Wartungspersonals auf speziellen Rechnern stattfinden. So verschlechtert sie im produktiven Betrieb nicht die Antwortzeiten der Webkomponenten.

Hinzu kommt, dass dann nicht jede SOAP-Nachricht einzeln validiert wird, sondern nur der vollständige Trace. Wie noch in Kapitel 9.7.1 genauer erläutert, führt die Validation für jede SOAP-Nachricht einzeln ohne weitere Optimierungen dazu, dass die SXQT-Ausdrücke immer wieder für fast den gleichen Trace ausgewertet werden. Jeweils wird nur eine SOAP-Nachricht dem Trace angefügt und dann die SXQT-Ausdrücke erneut

für den gesamten Trace ausgewertet. Eine einmalige Prüfung der SXQT-Ausdrücke für den vollständigen Trace würde aber auch sicherstellen, dass alle enthaltenen SOAP-Nachrichten geprüft werden. Nur für die sofortige Entscheidung, ob eine beobachtete SOAP-Nachricht der Spezifikation entspricht, muss diese Auswertung für jede SOAP-Nachricht immer wieder erfolgen. Wird der Validator für einen Trace manuell gestartet, wird eine sofortige Entscheidung nicht verlangt. Dann reicht es, die Prüfung nur einmal für den vollständigen Trace durchzuführen.

Eine manuell gestartete Validation bietet sich damit vor allem für die Anwendungen an, bei denen Fehler gesucht werden, sei es in Implementierungen von Webkomponenten oder im Verständnis über die Schnittstelle, wie in den Kapitel 9.4.1 bzw. 9.4.2 beschrieben. Insbesondere wenn eine solche Fehlersuche im laufenden Betrieb stattfindet, ist die Effizienz wichtig. Dann sollte der manuell gestarteten Validation der Vorzug gegeben werden.

Spielt Effizienz dagegen eine untergeordnetere Rolle oder ist es wichtig einen aufgetretenen Fehler schnell anzeigen zu können, kann die Validation auch durch eine Warteschlange gekoppelt oder sofort stattfinden. Dabei werden bei sofortiger Validation Fehler am schnellsten angezeigt. Die Entkopplung durch die Warteschlange ermöglicht dagegen bessere Antwortzeiten der Webkomponenten.

Einen Fehler schnell anzuzeigen, ist auch wichtig, wenn mit der Validation relevante Änderungen der Schnittstelle erkannt werden sollen, wie in Kapitel 9.4.3 beschrieben. Änderungen können dann erkannt werden, bevor im System ein ernstzunehmendes Problem auftritt. Das verbietet eine manuell gestartete Validation. Um die Antwortzeiten der Webkomponenten möglichst wenig zu vergrößern, bietet es sich für diese Anwendung an, Beobachtung und Validation durch eine Warteschlange zu entkoppeln.

9.6 Vorgang der automatischen Validation

Unabhängig davon, wann und in welchem Prozess die automatische Validation stattfindet, ist der notwendige Vorgang grundsätzlich der gleiche. Für eine Folge von beobachteten SOAP-Nachrichten muss geprüft werden, ob sie der Spezifikation entsprechen. Lediglich, wenn der Validator manuell gestartet wird, gibt es geringfügige Abweichungen. Was für eine beobachtete SOAP-Nachricht zu prüfen ist und wie die Prüfung durchgeführt werden kann, soll nachfolgend untersucht werden. Die Untersuchung ist Grundlage für die Implementierung von Validatoren.

Für die automatische Validation sind unterschiedliche Prüfungen erforderlich. Eine SXQT-Spezifikation besteht selbst aus dem in WSDL beschriebenen Teil sowie aus einer Menge von SXQT-Ausdrücken. Gegen beides müssen die SOAP-Nachrichten validiert werden. Darüber hinaus sollte auch geprüft werden, ob die SOAP-Nachricht überhaupt dem SOAP-Standard und dem in Kapitel 5.2 vorgeschlagenen Präzisierungsstandard für SOAP entspricht.

Anders verhält es sich mit dem vorgeschlagenen Präzisierungsstandard für WSDL. Er beschreibt, wie mit WSDL Schnittstellen von Webservices spezifiziert werden sollten. Damit bestimmt er den Aufbau von WSDL-Dokumenten, nicht den von SOAP-Nachrichten. Verstöße sind als Fehler in der Spezifikation anzusehen, nicht als Verstoß gegen die Spezifikation. Daher muss ein Validator diese Regeln nicht prüfen. Möglich wäre es beim Laden der Spezifikation um sicherzustellen, dass die Spezifikation in diesem Sinne korrekt ist.

Ein Validator kann WSDL-Dokumente auch verwenden, wenn sie keine SXQT-Ausdrücke enthalten. Dann sind die WSDL-Dokumente vom SXQT unabhängig. Es werden keine SXQT-Ausdrücke geprüft. Trotzdem kann der Validator prüfen, ob SOAP-Nachrichten dem WSDL-Dokument, dem SOAP-Standard und dem Präzisierungsstandard für SOAP entsprechen. Damit sind die nachfolgenden Überlegungen auch unabhängig von SXQT anwendbar.

9.6.1 XML-Schemadokumente

Ein wesentlicher Teil der Prüfung kann durch die Validation[62] der SOAP-Nachrichten gegen XML-Schemadokumente erfolgen. In den XML-Schemadokumenten werden Namensräume beschrieben. Durch die Validation gegen sie wird geprüft, ob die Namen dieser Namensräume korrekt verwendet werden. Hierzu benötigte Validatoren stehen in großer Auswahl zur Verfügung. [Sperberg-McQueen 02] verweist auf eine Reihe von ihnen.

Die Frage, gegen welche XML-Schemadokumente validiert wird, bestimmt, welche Teile der SOAP-Nachricht geprüft werden. So ist [W3C 02b] das XML-Schemadokument für den wichtigsten Namensraum des SOAP-Standards. Wird eine SOAP-Nachricht gegen dieses validiert, wird sichergestellt, dass die XML-Elemente und Attribute von SOAP korrekt verwendet worden sind.

So wird z. B. geprüft, dass Elemente `env:Envelope` als erstes Kindelement optional das Element `env:Header` haben und danach zwingend das Kindelement `env:Body`. (Siehe Kapitel 3.2.) In Fehlernachrichten wird auch z. B. geprüft, dass nur erlaubte Fehlercodes verwendet wurden. (Siehe Kapitel 3.4.)

Nicht geprüft wird allerdings, ob es sich überhaupt um eine SOAP-Nachricht handelt, ob sie also das Rootelement `env:Envelope` hat. Nicht geprüft wird auch, ob als Body-, Header- und Detaileinträge erlaubte XML-Elemente verwendet wurden. Das XML-Schemadokument lässt beliebige zu. Erst in WSDL wird festgelegt, welche erlaubt sind.

Hierzu werden in Frage kommende, anwendungsspezifische XML-Elemente zunächst in einem im WSDL-Dokument enthaltenen XML-Schemadokument definiert. Auch gegen dieses sollte eine SOAP-Nachricht validiert werden. Dadurch wird sichergestellt, dass auch anwendungsspezifische XML-Elemente und Attribute in der SOAP-Nachricht korrekt verwendet werden. Das schließt jedoch wieder nicht ein, dass sie an der richtigen Stelle z. B. als Body-, Header- oder Detaileintrag verwendet werden.

Statt jede SOAP-Nachricht einzeln gegen die XML-Schemadokumente zu validieren, könnte auch der vollständige Trace gegen sie validiert werden. Alle SOAP-Nachrichten sind im Trace vorhanden und die Validation gegen XML-Schemadokumente umfasst alle XML-Elemente und Attribute im Instanzdokument, also auch die der SOAP-Nachrichten. Diese Vorgehensweise bietet sich insbesondere an, wenn die Validation eines Traces gegen die Spezifikation manuell gestartet wird, weil dann nicht SOAP-Nachricht für SOAP-Nachricht, sondern der bereits vollständige Trace gegen die Spezifikation validiert wird.

---

[62] Der Leser sei darauf hingewiesen, dass der Begriff Validation in zwei Bedeutungen verwendet wird. Ein Instanzdokument (XML-Dokument) wird gegen XML-Schemadokumente validiert, Webkomponenten bzw. Traces dagegen gegen Spezifikationen des Spezifikationsverfahrens dieser Arbeit. Da SOAP-Nachrichten und Traces auch als XML-Dokumente angesehen werden können, können auch diese gegen XML-Schemadokumente validiert werden.

9.6.2 SOAP-Standard und Präzisierungsstandard

Die Prüfung der SOAP-Nachrichten gegen das XML-Schemadokument des SOAP-Standards stellt einen wichtigen Teil der Prüfung dar, ob eine SOAP-Nachricht dem SOAP-Standard entspricht. Es gibt jedoch auch Regeln im SOAP-Standard, die nicht in XML-Schema ausdrückbar und daher nicht im XML-Schemadokument des SOAP-Standards enthalten sind. Auch diese müssen durch den Validator geprüft werden.

Vor allem muss geprüft werden, ob überhaupt eine SOAP-Nachricht übertragen wurde. Hierzu muss für ein beobachtetes XML-Dokument festgestellt werden, ob es das Rootelement `env:Envelope` hat, sein lokaler Name also `Envelope` im Namenraum des SOAP-Standards ist. Dass dieses Element dann seinerseits die festgelegten Kindelemente und Attribute hat, wurde bereits durch die Validation gegen das XML-Schemadokument des SOAP-Standards geprüft.

Eine solche Prüfung wird in der Regel für jede SOAP-Nachricht einzeln erfolgen. Findet die Validation einer SOAP-Nachricht direkt nach deren Beobachtung statt oder sind Beobachter und Validator über eine Warteschlange gekoppelt, wie in Kapitel 9.5.3.2 beschrieben, wird dem Validator ohnehin eine einzelne SOAP-Nachricht übergeben. Diese kann direkt geprüft werden, noch bevor sie dem Trace zugefügt wird. Wird die Validation dagegen manuell gestartet, befinden sich bereits alle SOAP-Nachrichten im Trace. Dann müssen sie für die Prüfung wieder entnommen werden oder die Prüfung muss mit den im Trace gespeicherten SOAP-Nachrichten stattfinden.

Da für SOAP-Nachrichten angenommen wird, dass sie dem in Kapitel 5.2 vorgeschlagenen Präzisierungsstandard für SOAP entsprechen, müssen auch dessen Regeln geprüft werden. Die Regeln können nicht als XML-Schemadokument formuliert werden, weil der Namensraum von SOAP-Nachrichten bereits im XML-Schemadokument des SOAP-Standards definiert ist. Es ist nicht möglich, denselben Namensraum erneut, präziser zu definieren. Daher muss der Validator diese Regeln selbst prüfen.

Dazu muss er SOAP-Nachrichten darauf untersuchten, ob sie das Attribut `xsi:type` (Kapitel 5.2.3) oder `rpc:result` (Kapitel 5.2.4) enthalten. Beides ist nicht erlaubt. In Headereinträgen darf außerdem das Role-Attribut nicht verwendet worden sein, ebenso wie in Fehlernachrichten die Elemente `env:Role` und `env:Node` (Kapitel 5.2.6). Die SOAP-Nachricht darf auch nur einen Bodyeintrag enthalten (Kapitel 5.2.4).

Für den Präzisierungsstandard wurde darüber hinaus in Kapitel 5.2.5 festgelegt, dass SOAP-Nachrichten nach XML 1.0 serialisiert und im Zeichensatz UTF-8 oder UTF-16 kodiert sein müssen. Zwar werden solche Details häufig von verwendeten XML-Parsern verborgen. Wenn möglich, sollte der Validator aber auch diese Details prüfen.

Zu prüfen ist auch, ob das für den Präzisierungsstandard in Kapitel 5.2.1 vorgeschriebene Request-/Response-Kommunikationsmuster korrekt verwendet wird. Z. B. darf ein Request keine Fehlernachricht sein und damit der Name seines Bodyeintrages nicht `env:Fault`. Bei Responses ist das dagegen erlaubt.

Der Validator sollte auch prüfen, ob zu einem Response auch der Request beobachtet wurde. Zwar wurde in Kapitel 7.3.4 festgestellt, dass der Beobachter die Aufzeichnung des Traces erst nach einem Request und vor dem zugehörigen Response beginnen kann. Dann wird nur der Response aber nicht der zugehörige Request beobachtet und der Validator kann dem Response nicht den zugehörigen Request zuordnen. Request und Response werden aber auf derselben HTTP-Verbindung übertragen. Beachtet man, wie nach Kapitel 9.5 der Validator beobachtet, stellt man fest, dass sein Start nicht bei einer bestehen den HTTP-Verbindung stattgefunden haben kann. Damit muss er abweichend von der Feststellung in Kapitel 7.3.4 für jeden Response auch den Request beobachtet haben.

Damit kann der Validator für jeden Response prüfen, ob der Request bereits beobachtet wurde. Zusätzlich sollte geprüft werden, ob die Namen der Bodyeinträge von Request und Response zusammenpassen, wie in Kapitel 5.2.4 gefordert. Der Name des Bodyeintrages im Response muss also die Konkatenation des Namens des Bodyeintrages im Request mit der Zeichenkette „`Response`" sein. Alternativ darf ein Response aber auch stets eine Fehlernachricht sein und damit sein Bodyeintrag den qualifizierten Namen `env:Fault` haben.

9.6.3 WSDL-Beschreibung

Befolgt eine SOAP-Nachricht den SOAP- und Präzisierungsstandard, muss der Validator prüfen, ob sie auch der Schnittstellenbeschreibung im WSDL-Dokument entspricht. Dazu ist zu beachten, dass in einem WSDL-Dokument mehrere Webservices beschrieben sein können. Jeder Webservice kann wiederum mehrere Instanzen von gebundenen Schnittstellen haben. Auch diese sind im WSDL-Dokument deklariert. Dem Validator muss angegeben werden, welche Instanz einer Schnittstelle von welchem Webservice er beobachten soll. Gegen die Definition dieser Schnittstelle muss er validieren.

Dazu stellt sich die Frage, in welcher Form dem Validator angegeben werden kann, welche Instanz einer Schnittstelle er beobachten soll. Im WSDL-Dokument werden hierzu sowohl jeder Deklaration eines Webservices als auch jeder Deklaration einer Instanz einer Schnittstelle Namen gegeben. Der Name einer Instanz einer gebundenen Schnittstelle ist nur innerhalb eines Webservices eindeutig. Daher wird zur eindeutigen Bezeichnung einer Instanz einer Schnittstelle neben ihrem Namen auch der des Webservices benötigt.

Diese beiden Namen sollten dem Validator angegeben werden. Er kann sie verwenden, um die Deklaration der Instanz der Schnittstelle im WSDL-Dokument zu ermitteln. In dieser wird auf die Definition der gebundenen Schnittstelle verwiesen und von dieser wiederum auf die Definition der abstrakten Schnittstelle. Daher kann der Validator mit diesen beiden Namen die für die Validation relevanten Teile im WSDL-Dokument ermitteln.

Das ist erforderlich, um zu prüfen, ob eine SOAP-Nachricht auch dem dort Deklarierten entspricht. Durch die Validation einer SOAP-Nachricht gegen das im WSDL-Dokument enthaltene XML-Schemadokument wurde bereits geprüft, dass dort definierte, anwendungsspezifische XML-Elemente und Attribute korrekt verwendet werden. Nicht geprüft wurde aber, ob diejenigen XML-Elemente als Body-, Header- und Detaileinträge verwendet wurden, die nach dem WSDL-Dokument erlaubt sind. Das muss der Validator zusätzlich prüfen.

Um das für den Bodyeintrag zu prüfen, muss für jede SOAP-Nachricht untersucht werden, ob ihr einziger Bodyeintrag den qualifizierten Namen `env:Fault` hat. Dann handelt es sich um eine Fehlernachricht, die stets als Response erlaubt und als Request verboten ist. Handelt es sich um keine Fehlernachricht, muss der Validator prüfen, ob der qualifizierte Name des Bodyeintrags als Request bzw. Response gültig ist. Hierzu muss er die Definition der abstrakten Schnittstelle und der Nachrichtendeklaration untersuchen. Requests und Responses sind dabei zu unterscheiden.

Damit ein qualifizierter Name eines Bodyeintrags eines Requests gültig ist, muss eine Nachrichtendeklaration auf ihn verweisen, auf die ihrerseits von der abstrakten Schnittstelle als Request verwiesen wird. In der abstrakten Schnittstelle wird jede Operation mit einem Kindelement `wsdl:operation` definiert. In dessen Kindelement `wsdl:input` wird im Attribut `message` auf den Namen der Nachrichtendeklaration für den Request verwiesen. Die Nachrichtendeklaration wird in Elementen `wsdl:message` definiert. Der

Bodyeintrag wird darin in einem Kindelement `wsdl:part` definiert. Da nur ein Bodyeintrag erlaubt ist, ist nur ein Kindelement `wsdl:part` erlaubt. Es hat das Attribut mit Namen `element`, das den qualifizierten Namen des Bodyeintrages enthält.

Mit Hilfe des in Abbildung 137 gezeigten Beispiels soll diese Prüfung illustriert werden. Es handelt sich um einen Ausschnitt des WSDL-Dokumentes der Webkomponente für eine Internetzeitung, die schon mehrfach in dieser Arbeit als Beispiel verwendet wurde. Gezeigt werden zwei Nachrichtendeklarationen und die Deklaration der Operation `Get` in der Definition der abstrakten Schnittstelle.

```
<wsdl:message name="GetRequest">
  <wsdl:part name="FirstBodyEntry" element="nwst:Get" />
</wsdl:message>

<wsdl:message name="FaultDetailsMessageIDs">
    <wsdl:part name="FaultDetailsMessageIDs"
               element="nwst:MessageIDsSequence" />
</wsdl:message>

<wsdl:portType name="NewsServicePortType">
  <!-- ... -->

  <wsdl:operation name="Get">
    <wsdl:input  message="nwsi:GetRequest" />
    <wsdl:output message="nwsi:GetResponse" />
    <wsdl:fault  name="InvalidSessionID"
                 message="nwsi:FaultDetailsEmpty" />
    <wsdl:fault  name="InvalidMessageID"
                 message="nwsi:FaultDetailsMessageIDs" />
    <wsdl:fault  name="MessageLocked"
                 message="nwsi:FaultDetailsMessageIDs" />
  </wsdl:operation>

  <!-- ... -->
</wsdl:portType>
```

Abbildung 137: Definition der Operation `Get` in abstrakter Schnittstelle

Der Ausschnitt des WSDL-Dokumentes in Abbildung 137 legt fest, dass ein Bodyeintrag im Request den qualifizierten Namen `nwst:Get` haben darf. Das gilt, weil einerseits in einer Nachrichtendeklaration das Attribut `wsdl:message/wsdl:part/@element` diesen qualifizierten Namen als Wert hat. Andererseits gibt es in der Definition der abstrakten Schnittstelle eine Operation (die Operation `Get`), die festlegt dass diese Nachrichtendeklaration ihren Request beschreibt. Das wird dadurch angezeigt, dass das Attribut `wsdl:portType/wsdl:operation/wsdl:input/@message` den Namen der Nachrichtendeklaration hat.

Die Prüfung, ob im Response der qualifizierte Name des Bodyeintrages gültig ist, erfolgt analog. Die Nachrichtendeklaration wird für den Response auf die gleiche Art formuliert. In der Definition der abstrakten Schnittstelle wird aber die Nachrichtendeklaration mit dem Element `wsdl:operation/wsdl:output` einer Operation als Response zugeordnet, statt mit dem Element `wsdl:operation/wsdl:input`, wie beim Request. Neben den so ermittelten qualifizierten Namen muss der Validator beim Response zusätzlich den Namen `env:Fault` zulassen, weil Responses auch Fehlernachrichten sein dürfen.

Ist das der Fall, muss der Validator in der Fehlernachricht prüfen, ob deren Detaileinträge der Deklaration in WSDL entsprechen. Dazu muss er dem Response zunächst den

Request zuordnen. Nur aus dessen Bodyeintrag kann er bestimmen, welche Operation aufgerufen wurde. Die Deklaration der Operation in der abstrakten Schnittstelle enthält neben der Beschreibung von Request und Response auch die Beschreibungen von Fehlernachrichten und zwar in Elementen `wsdl:operation/wsdl:fault`. In einem dieser Elemente müssen die auch die Detaileinträge der beobachteten SOAP-Nachricht beschrieben sein.

Das in Abbildung 137 gezeigte Beispiel legt somit fest, dass eine Fehlernachricht der Operation `Get` ein Detailelement `nwst:MessageIDsSequence` haben darf. In der Deklaration der Operation `Get` sind drei Fehlernachrichten deklariert. Zwei von ihnen verweisen auf die Nachrichtendeklaration mit dem Namen `FaultDetailsMessageIDs`. Die Nachrichtendeklaration enthält einen Part. Daher darf es in der Fehlernachricht nur ein Detailelement geben. Dessen qualifizierter Name muss `nwst:MessageIDsSequence` sein, weil dieser im Attribut `wsdl:message/wsdl:part/@element` als Wert angegeben ist.

Neben Body- und Detaileinträgen kann in WSDL auch deklariert werden, dass eine SOAP-Nachricht Headereinträge enthält. Es müssen aber nicht alle Headereinträge deklariert sein. Daher muss der Validator nicht prüfen, ob alle in SOAP-Nachrichten enthaltenen Headereinträge deklariert sind. Er sollte aber prüfen, ob deklarierte Headereinträge in der SOAP-Nachricht vorhanden sind.

Hierzu ermittelt er mit Hilfe des qualifizierten Namens des Bodyeintrages, zu welcher Operation die SOAP-Nachricht gehört. Operationen sind außer in der abstrakten Schnittstelle auch in der gebundenen Schnittstelle deklariert. In beiden Fällen wird die Operation mit einem Element `wsdl:operation` deklariert, das im Attribut `name` den Namen der Operation enthält. Durch die Gleichheit dieses Namens kann erkannt werden, welche Deklarationen dieselbe Operation beschreiben.

Die Beschreibung der Headereinträge ist in der gebundenen Schnittstelle enthalten. Dort sind Requests in Elementen `wsdl:operation/wsdl:input` und Responses in Elementen `wsdl:operation/wsdl:output` beschrieben. Diese Elemente können Kindelemente `soap12:header` haben, die jeweils einen Headereintrag im Request bzw. Response beschreiben. Dazu wird im Element `soap12:header` mit dem Attribut `message` auf eine Nachrichtendeklaration verwiesen und im Attribut `part` auf einen Part in der Nachrichtendeklaration. Der so bezeichnete Part muss als Headereintrag der SOAP-Nachricht zugefügt sein. Der Validator sollte prüfen, dass der Headereintrag vorhanden ist.

Um dies Prüfung an einem Beispiel zu illustrieren, zeigt Abbildung 138 die Deklaration der Operation `Get` aus der Definition der gebundenen Schnittstelle. Der zugehörige Teil der abstrakten Schnittstelle wurde bereits in Abbildung 137 gezeigt. Die gebundene Schnittstelle ist so erweitert, dass im Request der Operation `Get` ein Headereintrag `nwst:TipURL` enthalten sein muss.

In der Definition der gebundenen Schnittstelle wird das durch das Element `wsdl:binding/wsdl:operation/wsdl:input/soap12:header` angezeigt. Der Name `nwsi:TipURLHeader` im Attribut `message` verweist auf die Nachrichtendeklaration, die am Anfang von Abbildung 138 abgebildet ist. Das Attribut `part` enthält den Namen `TipURLHeader` des Parts innerhalb der Nachrichtendeklaration. Dessen Deklaration enthält wieder im Attribut mit Namen `element` den qualifizierten Namen `nwst:TipURL` des XML-Elementes, das in der SOAP-Nachricht als Headereintrag vorhanden sein muss.

```
<wsdl:message name="TipURLHeader">
  <wsdl:part name="TipURLHeader" element="nwst:TipURL" />
</wsdl:message>

<wsdl:binding name="NewsServiceBinding"
              type="nwsi:NewsServicePortType" >
  <soap12:binding style="document"
                  transport="http://www.w3.org/2002/06/soap/bindings/HTTP/" />
  <!-- ... -->

  <wsdl:operation name="Get" >
    <soap12:operation />
    <wsdl:input>
      <soap12:body use="literal" />
      <soap12:header use="literal"
                     message="nwsi:TipURLHeader"
                     part="TipURLHeader"  />
    </wsdl:input>
    <wsdl:output><soap12:body use="literal" /></wsdl:output>
    <wsdl:fault name="InvalidSessionID">
      <soap12:fault name="InvalidSessionID" use="literal" />
    </wsdl:fault>
    <!-- ... -->
  </wsdl:operation>

  <!-- ... -->
</wsdl:binding>
```

Abbildung 138: Definition der Operation `Get` in gebundener Schnittstelle mit Headereintrag

9.6.4 SXQT-Ausdrücke

Für die bisher beschriebenen Prüfungen verwendet der Validator Informationen aus dem SOAP-Standard und WSDL-Dokumenten ohne die Erweiterung um SXQT-Ausdrücke aus Kapitel 7. Daher kann dieser Teil der Validation auch unabhängig von SXQT erfolgen, wenn für einen Webservice nur ein geeignetes WSDL-Dokument zur Verfügung steht.

Ist das WSDL-Dokument jedoch um SXQT-Ausdrücke erweitert, muss der Validator für jeden[63] prüfen, ob er für die Beobachtung zutrifft. Geprüft wird dabei nicht die Beobachtung einer SOAP-Nachricht, sondern alle bisherigen, also der Trace. Der Trace muss durch den Validator aus den beobachteten SOAP-Nachrichten erstellt werden, wenn das nicht bereits vorher durch einen Beobachter geschehen ist.

Das ist jedoch der Fall, wenn der Validator manuell gestartet wird, wie in Kapitel 9.5.2.3 beschrieben. Dann zeichnet ein Beobachter den Trace auf. Der vollständige Trace wird dem Validator übergeben und die SXQT-Ausdrücke für ihn ausgewertet. Wird bei Auswertung nur eines SXQT-Ausdrucks der Wert `false` oder ein Fehler geliefert, zeigt das einen Verstoß gegen die Spezifikation an.

Werden dem Validator dagegen einzelne SOAP-Nachrichten übergeben oder von ihm beobachtet, muss er den Trace selbst erstellen und speichern. Jede SOAP-Nachricht wird am Ende des Traces angefügt. Danach werden die SXQT-Ausdrücke für den Trace ausgewertet. Wieder wird ein Verstoß gegen die Spezifikation dadurch angezeigt, dass bei der Auswertung der Wert `false` oder ein Fehler geliefert wird.

---

[63] Diese Aussage ist vereinfacht. Unten in Kapitel 9.6.4 wird festgestellt, dass nur die SXQT-Ausdrücke ausgewertet werden, die in der Sicht des Validators gültig sind.

Würde nach einem solchen Verstoß die letzte SOAP-Nachricht im Trace belassen, würden danach weitere SOAP-Nachrichten zugefügt und erneut die SXQT-Ausdrücke ausgewertet. Dann würde immer wieder ein Verstoß gegen die Spezifikation gefunden werden, weil sich die fehlerhafte SOAP-Nachricht noch im Trace befindet. Die SXQT-Ausdrücke prüfen immer wieder den gesamten Trace und erkennen daher die fehlerhafte SOAP-Nachricht auch immer wieder. Daher muss eine SOAP-Nachricht, die gegen die Spezifikation verstößt, wieder aus dem Trace entfernt werden. Handelt es sich bei ihr um einen Response, sollte das auch mit dem zugehörigen Request geschehen.

Welche SXQT-Ausdrücke ein Validator für einen beobachteten Trace auszuwerten hat, hängt von seiner Sicht ab. In Kapitel 7.3.4 wurde festgestellt, dass es SXQT-Ausdrücke gibt, die nur verwendbar sind, wenn der Beobachter die Webclientsicht hat bzw. die Webservicesicht. Daher wurde jedem SXQT-Ausdruck im WSDL-Dokument das Attribut `viewEntity` zugefügt, in dem angegeben ist, in welcher Sicht er verwendbar ist, wie in Kapitel 8.3.3 beschrieben. Mögliche Werte waren `Client`, `Service` und `Any`, wobei `Any` festlegt, dass es für den SXQT-Ausdruck keine Rolle spielt, ob in Webclient- oder Webservicesicht beobachtet wurde.

Ein Validator darf nur die SXQT-Ausdrücke für Traces auswerten, die seiner Sicht entsprechen. Für andere ist nicht sichergestellt, dass sie ein aussagekräftiges Ergebnis liefern. Daher muss dem Validator übergeben werden, ob er die Webclient- oder Webservicesicht hat. Hat er die Webclientsicht, wertet er nur die SXQT-Ausdrücke aus, deren Attribute `viewEntity` die Werte `Client` oder `Any` haben. In der Webservicesicht verwendet er diejenigen, bei denen das Attribut `viewEntity` die Werte `Service` oder `Any` hat.

9.7 Optimierungen

Durch die automatische Validation gegen XML-Schemadokumente, SOAP-Standard, WSDL-Dokument und SXQT-Ausdrücke kann also sichergestellt werden, dass eine beobachtete Nachricht eine SOAP-Nachricht ist und der SXQT-Spezifikation entspricht. Insbesondere wenn das im produktiven Betrieb geschieht, ist die Frage nach der Effizienz wichtig. Ansonsten kann die automatische Validation zu einer Verschlechterung der Antwortzeiten der Webkomponenten führen und damit die Effizienz des gesamten Systems verschlechtern. Nachfolgend sollen daher zunächst Optimierungen motiviert und danach konkrete vorgeschlagen werden.

9.7.1 Notwendigkeit von Optimierungen

Die Validation gegen XML-Schemadokumente, SOAP-Standard und WSDL-Dokumente ist weniger problematisch als die gegen SXQT-Ausdrücke. Jede beobachtete SOAP-Nachricht wird nur einmal untersucht. Durchzuführende Prüfungen sind einfach. Eine SXQT-Spezifikation kann dagegen sehr viele SXQT-Ausdrücke enthalten, die alle geprüft werden müssen. Außerdem wird ohne Optimierungen eine beobachtete SOAP-Nachricht immer und immer wieder mit allen SXQT-Ausdrücken geprüft. Beides soll nachfolgend erläutert werden.

Viele SXQT-Ausdrücke enthält die Spezifikation dann, wenn an den Webservice viele Anforderungen gestellt werden. Wie in Kapitel 7 beschrieben, beruht das Spezifikationsverfahren dieser Arbeit darauf, dass jede an einen Webservice gestellte Anforderung einzeln als SXQT-Ausdruck formalisiert wird. In Kapitel 6 bzw. 7.7 wurden z. B. eine

Reihe von Anforderungen vorgestellt, die an die Webkomponente für eine Internetzeitung gestellt werden. Das waren jedoch nur wenige Beispiele. Sie vollständiger zu beschreiben, würde wesentlich mehr Anforderungen und damit auch wesentlich mehr SXQT-Ausdrücke erfordern. Sie alle müssten aber stets für den Trace geprüft werden.

Ohne Optimierungen müsste das für jede beobachtete SOAP-Nachricht erneut geschehen, was in vielen Fällen unnötig ist, wie die folgende Überlegung zeigt. Der Validator beginnt mit einem leeren Trace. Beobachtet er die erste SOAP-Nachricht, fügt er sie dem Trace an und validiert ihn. Beobachtet er die zweite SOAP-Nachricht, fügt er diese wieder dem Trace an und validiert ihn erneut vollständig. Dabei wird neben der zweiten SOAP-Nachricht auch die erste noch einmal validiert. Wurde die `n-te` SOAP-Nachricht beobachtet, dem Trace zugefügt und dieser validiert, enthält der Trace `n` SOAP-Nachrichten. Die erste wurde dann n mal validiert, die zweite `n-1`-mal und so weiter.

Diese Vorgehensweise ist notwendig, weil einerseits nach jeder SOAP-Nachricht sofort entschieden werden soll, ob sie der Spezifikation entspricht. Daher muss die Prüfung für jede SOAP-Nachricht stattfinden. Andererseits gibt es Anforderungen, die mehrere SOAP-Nachrichten miteinander in Beziehung setzen. Prinzipiell können beliebig viele im Trace enthaltene SOAP-Nachrichten miteinander in Beziehung gesetzt werden. In solchen Fällen ist es notwendig, alle SOAP-Nachrichten im Trace zu testen, also müssen die SXQT-Ausdrücke so konstruiert werden. Sie sind so auch für Menschen besser verständlich, weil beschrieben ist, was für den gesamten Trace gelten muss, anstatt das nur für einzelne SOAP-Nachrichten zu beschreiben. Für die Validation ist dieser Vorgang aber häufig zu aufwendig.

Die beiden beschriebenen Probleme legen es also nahe zu versuchen, die Validation gegen SXQT-Ausdrücke zu optimieren. Nachfolgend sollen hierzu Ansätze vorgestellt werden.

9.7.2 Zusammenfassen von SXQT-Ausdrücken

Optimierengen sollten nach Möglichkeit dem verwendeten XQuery-Prozessor überlassen werden, der die SXQT-Ausdrücke auswertet. XQuery wurde, wie in Kapitel 7.4 beschrieben, vor allem zum Abfragen von Datenquellen entwickelt, z. B. von Datenbanken. Daher ist zu erwarten, dass bei wachsender Verbreitung XQuery-Prozessoren ohnehin ähnlich gut optimieren werden, wie es heute relationale Datenbanksysteme für die Sprache SQL leisten. Diese Fähigkeit sollte genutzt und nicht erneut im Validator realisiert werden.

Übergibt der Validator die SXQT-Ausdrücke einzeln an den XQuery-Prozessor, ist dieser auch nur in der Lage, deren Ausführungszeit einzeln zu minimieren. Für die Auswertung der vielen SXQT-Ausdrücke einer Spezifikation wäre es dagegen interessanter, deren Gesamtausführungszeit zu minimieren. Das kann erreicht werden, indem die einzelnen SXQT-Ausdrücke zu einem zusammengefasst werden. Der XQuery-Prozessor kann sie dann gemeinsam untersuchen und die gemeinsame Ausführung optimieren. Dabei kann er auch in ihnen enthaltene identische Teilausdrücke zusammenfassen und so vermeiden, dass sie mehrfach ausgeführt werden müssen.

In der Spezifikation sollten die Anforderungen weiterhin als einzelne SXQT-Ausdrücke enthalten sein. So können Menschen sie besser verstehen. Das Zusammenfassen zu einem SXQT-Ausdruck sollte automatisch geschehen, zweckmäßiger Weise durch den Validator. Das kann z. B. beim Laden der Spezifikation geschehen.

Hierzu ist keine aufwendige Umformung der SXQT-Ausdrücke notwendig. Damit die Spezifikation zutrifft, müssen alle SXQT-Ausdrücke den logischen Wert `true` liefern. Liefert nur ein SXQT-Ausdruck den logischen Wert `false` oder einen Fehler, zeigt das einen Verstoß gegen die Spezifikation an. Daher können sie mit Hilfe der Konjunktion (logisches Und) zu einem SXQT-Ausdruck zusammengefasst werden.

Abbildung 139 zeigt das an einem Beispiel, in dem zwei SXQT-Ausdrücke zu einem zusammengefasst wurden. Einzeln wurden sie bereits in Abbildung 103 auf Seite 181 bzw. Abbildung 109 auf Seite 186 gezeigt. Beide wurden direkt wiedergegeben, verbunden durch die Operation `and`, mit der in XQuery die Konjunktion notiert wird.

```
every $m
  in opr:restrict(opr:tr(),
                  fn:QName-in-context("nwst:Login", fn:false()))
  satisfies (opr:event-body-entry($m)/nwst:Name eq "Gast")
             eq fn:empty(opr:event-body-entry($m)/nwst:Password)
and
  every $m
    in opr:restrict(opr:tr(),
                    fn:QName-in-context("nwst:Login", fn:false()))
    satisfies
      let $resp := opr:associated-response($m)
      return
        fn:not(opr:event-body-entry($m)/nwst:Name ne "Gast"
               and fn:empty(opr:event-body-entry($m)
                            /nwst:Password))
        or fn:empty($resp)
        or opr:event-body-entry($resp)
                /env:Code/env:Subcode/env:Value
           eq fn:QName-in-context("nwse:AccessDenied", fn:false())
```

Abbildung 139: Zwei durch Konjunktion zusammengefasste SXQT-Ausdrücke

Auf diese Art lassen sich auch mehr als zwei SXQT-Ausdrücke zusammenfassen. Sie sind stets durch eine Zeile mit der Operation `and` zu verbinden. Dabei ist aber zu beachten, dass die SXQT-Ausdrücke in der Regel einen Query-Prolog haben. Die Query-Prologe müssen zunächst abgespalten und zu einem zusammengefasst werden. Mit der Konjunktion können nur die logischen Ausdrücke selbst verbunden werden.

Das Ergebnis ist ein SXQT-Ausdruck, der in vielen Fällen unnötig aufwendig erscheint. Da in den verbundenen SXQT-Ausdrücken häufig die gleichen Teilausdrücke verwendet werden, sind diese mehrfach enthalten. Der SXQT-Ausdruck in Abbildung 139 enthält z. B. zwei Allquantoren. Für beide wird die Sequenz der Requests der Operation `Login` gebildet. Der XQuery-Prozessor muss solche gemeinsamen Teilausdrücke erkennen und nur einmal auswerten. Z. B. könnten die beiden Allquantoren zu einem zusammengefasst werden, wie es in Abbildung 140 gezeigt wird. Auch die doppelt vorkommenden Teilausdrücke `fn:empty(opr:event-body-entry($m)/nwst:Password)` und `opr:event-body-entry($m)/nwst:Name` sollten nur einmal ausgewertet werden.

Das Ergebnis solcher Optimierungen ist, dass ein XQuery-Prozessor einen einzigen, großen SXQT-Ausdruck deutlich effizienter auswerten kann als die einzelnen SXQT-Ausdrücke, aus denen er zusammengesetzt ist. Das rechtfertigt den ohnehin geringen Aufwand des Validators sie zusammenzusetzen.

```
every $m
  in opr:restrict(opr:tr(),
                  fn:QName-in-context("nwst:Login", fn:false()))
  satisfies
    (opr:event-body-entry($m)/nwst:Name eq "Gast")
    eq fn:empty(opr:event-body-entry($m)/nwst:Password)
  and
    let $resp := opr:associated-response($m)
    return
      fn:not(opr:event-body-entry($m)/nwst:Name ne "Gast"
             and fn:empty(opr:event-body-entry($m)/nwst:Password))
      or fn:empty($resp)
      or opr:event-body-entry($resp)
              /env:Code/env:Subcode/env:Value
         eq fn:QName-in-context("nwse:AccessDenied", fn:false())
```

Abbildung 140: Zwei durch Konjunktion zusammengefasste SXQT-Ausdrücke, optimiert

9.7.3 Validation der letzten SOAP-Nachricht

Schwieriger ist zu verhindern, dass dieselben SOAP-Nachrichten immer wieder gegen die SXQT-Ausdrücke validiert werden. Soll das dem XQuery-Prozessor überlassen werden, müsste dieser „Wissen" darüber haben, was bereits geprüft wurde und was daher bei einer erneuten Prüfung nicht mehr geprüft werden muss. Insbesondere muss ihm bekannt sein, dass der Trace bereits für alle SOAP-Nachrichten bis auf die letzte geprüft wurde.

Ein SXQT-Ausdruck wie in Abbildung 103 auf Seite 181 könnte dann vom XQuery-Prozessor optimiert werden. Der SXQT-Ausdruck betrachtet im Allquantor nur SOAP-Nachrichten, die Requests der Operation `Login` sind. Ist die letzte beobachtete SOAP-Nachricht kein solcher Request, muss der XQuery-Prozessor den Ausdruck gar nicht auswerten. Er kann sofort den logischen Wert `true` liefern. Ist die letzte beobachtete SOAP-Nachricht ein Request der Operation `Login`, reicht es nur für diese den Ausdruck hinter dem Schlüsselwort `satisfies` zu prüfen. Für alle anderen Requests der Operation `Login`, die sich im Trace befinden, wurde dieser Ausdruck bereits vorher geprüft.

Der XQuery-Prozessor könnte den SXQT-Ausdruck aus Abbildung 103 daher so optimieren, wie in Abbildung 141 gezeigt. Statt einen Allquantor zu verwenden, wird die letzte SOAP-Nachricht im Trace mit dem Teilausdruck `opr:tr()[last()]` ermittelt. Für sie wird geprüft, ob sie ein Request der Operation `Login` ist. Ist das nicht der Fall, trifft der SXQT-Ausdruck zu. Nur bei einem Request der Operation `Login` ist, muss der Teilausdruck, der im Allquantor hinter dem Schlüsselwort `satisfies` angegeben war, zutreffen. Er ist in Abbildung 141 in den letzten beiden Zeilen wiedergegeben.

```
let $m := opr:tr()[fn:last()]
  return
    (opr:event-body-entry($m)
     ne fn:QName-in-context("nwst:Login", fn:false()))
    or
    (opr:event-body-entry($m)/nwst:Name eq "Gast")
       eq fn:empty(opr:event-body-entry($m)/nwst:Password)
```

Abbildung 141: SXQT-Ausdruck, der nur letzte SOAP-Nachricht prüft

Eine solche Optimierung ist nicht für jeden SXQT-Ausdruck möglich, auch wenn er ähnlich mit Allquantor konstruiert ist. Hier war es möglich, weil sich der Teilausdruck hinter dem Schlüsselwort `satisfies` nur auf die SOAP-Nachricht bezog, die durch den

Allquantor an die Variable `$m` gebunden wird. Die Optimierung ist auch möglich, wenn sich der Teilausdruck außer auf diese SOAP-Nachricht noch auf weitere bezieht, die vor ihr beobachtet worden sein müssen. Dann kann sich das Ergebnis des Teilausdrucks nach ihrer Beobachtung nicht mehr durch später beobachtete SOAP-Nachrichten verändern. Daher reicht es, den Teilausdruck zu prüfen, wenn sie beobachtet wurde.

Würde diese Optimierung aber auf den SXQT-Ausdruck in Abbildung 140 angewendet, wäre das nicht der Fall. Es wird mit der Funktion `opr:associated-response($m)` dem Request, auf den mit der Variable `$m` verwiesen wird, der Response zugeordnet. Der Response wird aber erst nach dem Request beobachtet. Daher kann sich das Ergebnis des Teilausdrucks nach dem Schlüsselwort `satisfies` auch noch nach der Beobachtung des Request verändern. Aus diesem Grund muss der Teilausdruck immer wieder geprüft werden, damit noch erkannt wird, wenn der Response nicht den spezifizierten Inhalt hat. Die aufgezeigte Optimierung ist für diesen SXQT-Ausdruck also nicht möglich.

9.7.4 Validation nicht für jede SOAP-Nachricht einzeln

Ein ganz anderer Weg zu verhindern, dass dieselben SOAP-Nachrichten immer wieder gegen die SXQT-Ausdrücke validiert werden, ist die in Kapitel 9.5.2.3 beschriebene Realisierung des manuell gestarteten Validators. Da bei ihr der Trace zunächst nur durch einen Beobachter aufgezeichnet und erst danach der vollständige Trace einmal validiert wird, wird ohnehin jede SOAP-Nachricht nur einmal gegen die SXQT-Ausdrücke validiert. Schon in Kapitel 9.2 wurde festgestellt, dass das eine erhebliche Optimierung darstellt. Es ist dann aber nicht mehr möglich, sofort für eine SOAP-Nachricht festzustellen, ob sie der Spezifikation entspricht. Erst wenn im Nachhinein die Validation manuell gestartet wird, kann das festgestellt werden. Für einige Anwendungen ist das jedoch akzeptabel.

Als Mittelweg zwischen dem Validieren jeder einzelnen SOAP-Nachricht und dem manuell gestarteten Validator, könnte der beobachtete Trace jeweils für mehrere beobachtete SOAP-Nachrichten gegen die SXQT-Ausdrücke validiert werden. Z. B. könnte das stets nach einer bestimmten Zahl von beobachteten SOAP-Nachrichten geschehen oder wenn der validierende Rechner wenig ausgelastet ist. Da der Trace dann seltener gegen die SXQT-Ausdrücke validiert wird als wenn das für jede SOAP-Nachricht geschieht, ist die Vorgehensweise effizienter.

Damit entfällt aber auch die Möglichkeit, dass sofort für jede beobachtete SOAP-Nachricht entschieden werden kann, ob sie der Spezifikation entspricht. Die Entscheidung erfolgt jedoch anders als beim manuell gestarteten Validator automatisch, wenn auch etwas verzögert. Diese Eigenschaft hat auch die Realisierung des Validators mit einer Warteschlange, die in Kapitel 9.5.3.2 beschrieben wurde. Bei ihr werden SOAP-Nachrichten in einer Warteschlange zwischengespeichert und nach und nach validiert. Daher kann auch bei ihr nicht sofort festgestellt werden, ob eine SOAP-Nachricht der Spezifikation entspricht, sondern erst etwas verzögert. Da sowohl die Realisierung mit Warteschlange als auch die beschriebene Optimierung diese Eigenschaft hat, liegt es nahe, dass Anwendungen, für die diese Eigenschaft akzeptabel ist, beides gemeinsam verwenden.

Somit gibt es unterschiedliche Möglichkeiten die Validation gegen die SXQT-Ausdrücke zu optimieren, so dass sich auch die Effizienz der Validation von Webkomponenten als Ganzes verbessern lässt. Das erhöht die ohnehin große Attraktivität der Validation als Werkzeug zur Verbesserung der Stabilität von Softwaresystemen aus Webkomponenten.

10 Spezifikation beliebiger XML-Dokumente

Vergleicht man das Verhältnis zwischen WSDL und Traces auf der einen Seite sowie XML-Schema und XML-Dokumenten (Instanzdokumenten) auf der anderen, stellt man fest, dass in beiden Fällen eine ähnliche Situation vorliegt. Ebenso wie WSDL-Dokumente, die bei Webkomponenten beobachtete Traces beschreiben, beschreiben auch XML-Schemadokumente deren Instanzdokumente. Die Beschreibungsfähigkeit ist in beiden Fällen begrenzt. Das legt nahe zu untersuchen, ob nicht das Spezifikationsverfahren SXQT auf die Spezifikation beliebiger Instanzdokumente übertragen werden kann. Diese Untersuchung soll in diesem Kapitel erfolgen.

10.1 Grenzen von XML-Schema

Dass XML-Schema begrenzt ist, wurde bereits in Kapitel 6 untersucht. Dort ging es um die Ausdrucksfähigkeit von WSDL. Da jedoch die Ausdrucksfähigkeit von WSDL auch von der Ausdrucksfähigkeit von XML-Schema abhängt, wurde diese dort mituntersucht. Das geschah insbesondere in den Kapiteln 6.1.1, 6.2.1 sowie 6.2.2.

In Kapitel 6.1.1 auf Seite 98 wurde festgestellt, dass es in XML-Schema nicht möglich ist, Beziehungen zwischen Werten im Instanzdokument festzulegen. Es wurde ein Beispiel gezeigt, in dem ein Element eine Zahl enthielt, die stets gleich der Anzahl von Kindelementen eines anderen Elementes sein muss. Diese Beziehung lässt sich in XML-Schema nicht ausdrücken. Das war auch nicht im Beispiel in Kapitel 6.2.1 auf Seite 104 möglich, bei dem ein optionales Element abhängig vom Wert eines anderen Elementes vorhanden sein musste oder nicht. Schließlich wurde in Kapitel 6.2.2 auf Seite 105 ein Beispiel untersucht, in dem ein Element entweder ein Attribut oder Kindelemente haben muss, aber nicht beides haben darf.

Für solche nicht in XML-Schema ausdrückbaren Anforderungen, wurde in Kapitel 7.7.3 gezeigt, wie sie als SXQT-Ausdrücke, also logische XQuery-Ausdrücke formuliert werden können, um Schnittstellen von Webkomponenten genauer zu beschreiben. Das sollte analog auch für Instanzdokumente möglich sein.

10.2 Spezifikation mit absoluten XQuery-Ausdrücken

Für Spezifikationen von Instanzdokumenten wird ein Trace nicht benötigt. Für die Beschreibung von Schnittstellen von Webkomponenten war er erforderlich, um mehrere SOAP-Nachrichten miteinander in Beziehung setzen zu können. Hier sollen jedoch nur einzelne Instanzdokumente spezifiziert werden und nicht mehrere.

Statt für einen Trace prüft daher ein logischer XQuery-Ausdruck für ein einzelnes Instanzdokument, ob es den Anforderungen der Spezifikation entspricht. Hierzu muss der logische XQuery-Ausdruck auf das Instanzdokument zugreifen können. In Kapitel 7.5.3 wurde festgelegt, der Trace dem SXQT-Ausdruck in der Input-Sequence übergeben

wird. Dazu wurde der Trace zu einem XML-Dokument vervollständigt und dessen Dokumentknoten als einziges Element der Input-Sequence an den SXQT-Ausdruck übergeben.

Für die Spezifikation beliebiger Instanzdokumente wird das Instanzdokument analog dazu übergeben. Da ein Instanzdokument bereits ein vollständiges XML-Dokument ist, muss es nicht vervollständigt werden. Sein Dokumentknoten wird als einziges Element der Input-Sequence übergeben. Er kann im logischen XQuery-Ausdruck mit der Standardfunktion `fn:input` ermittelt werden.

Anhand von zwei Beispielen soll veranschaulicht werden, wie logische XQuery-Ausdrücke für die Spezifikationen von Instanzdokumenten formuliert werden können. Das erste Beispiel ist fast identisch mit dem in Kapitel 7.7.3.2 auf Seite 181 für Traces vorgestellten. Es behandelte den Grenzfall der alternativen Strukturen aus Kapitel 6.2.2. Es wurde gefordert, dass innerhalb jeder SOAP-Nachricht jedes Element `my:a` entweder das Attribut `ref` oder ein Kindelement `my:str` hat, aber nicht beides gleichzeitig.

Offensichtlich kann diese Anforderung sofort auf beliebige Instanzdokumente übertragen werden und unabhängig von SOAP-Nachrichten gelten. Dann muss die Anforderung statt für jede SOAP-Nachricht im Trace für das Instanzdokument gelten. Auch der logische XQuery-Ausdruck, der die Anforderung formuliert, muss nur geringfügig geändert werden. Für Traces wurde er in Abbildung 104 auf Seite 181 gezeigt. Dort enthält er die Funktion `opr:tr`, um auf den Trace zuzugreifen, so dass die Anforderung für alle Message-Elemente im Trace geprüft wurde. Dadurch wurde er aber auch davon abhängig, dass ein Trace untersucht wird.

```
every $a in fn:input()//my:a
 satisfies fn:not(fn:exists($a/@ref) eq fn:exists($a/my:str))
```

Abbildung 142: Anforderung an alternative Struktur, verallgemeinert als XQuery-Ausdruck

Damit der logische XQuery-Ausdruck für beliebige Instanzdokumente verwendet werden kann, muss er verallgemeinert werden, wie in Abbildung 142 gezeigt. Statt der Funktion `opr:tr` wird nun die Standardfunktion `fn:input` verwendet. Dadurch wird statt für alle Elemente `my:a` im Trace für alle Elemente `my:a` im Instanzdokument geprüft, ob die Anforderung für sie gilt. Da auch der zum XML-Dokument vervollständigte Trace letztlich ein Instanzdokument ist, wäre der logische XQuery-Ausdruck in Abbildung 142 sogar auch weiterhin als SXQT-Ausdruck für Traces verwendbar.

Die zweite Anforderung, die nun als Beispiel dienen soll, ist sehr einfach. Sie legt fest, dass das Instanzdokument das Rootelement `env:Envelope` hat. Anforderungen dieser Art sind notwendig, weil es in XML-Schema nicht möglich ist, von Instanzdokumenten die Rootelemente festzulegen. In XML-Schema wird lediglich eine Art Vokabular festgelegt, das im Instanzdokument verwendet wird. So kann für Elemente mit bestimmten qualifizierten Namen festgelegt werden, welche Kindelemente und Attribute sie haben. Es kann aber nicht festgelegt werden, wo sie verwendet werden dürfen. Jedes global definierte Element kann daher das Rootelement des Instanzdokumentes sein.

Mit einem logischen XQuery-Ausdruck kann jedoch festgelegt werden, welches das Rootelement ist. Abbildung 143 zeigt einen solchen logischen XQuery-Ausdruck für Instanzdokumente, die SOAP-Nachrichten enthalten. In dieser Arbeit wurden bereits einige solcher Instanzdokumente gezeigt, z. B. in Abbildung 112 auf Seite 190. Wie in Kapitel 3.2 beschrieben, haben sie das Rootelement `env:Envelope`. Diese Tatsache wird mit dem logischen XQuery-Ausdruck in Abbildung 143 geprüft.

```
fn:exists(fn:input()/env:Envelope)
```

Abbildung 143: Anforderung an Rootelement als XQuery-Ausdruck

Dazu wird zunächst mit dem Pfadausdruck `fn:input()/env:Envelope` eine Sequenz ermittelt, die alle Kindelemente des Dokumentknoten enthält, die den Namen `env:Envelope` haben. Die Sequenz ist leer, wenn der Dokumentknoten ein anderes Kindelement als das Element `env:Envelope` hat. Ansonsten enthält sie genau das Element. Ob sie es enthält wird mit der Standardfunktion `fn:exists` getestet.

Neben den beiden gezeigten, einfachen Beispielen können auch Anforderungen an komplexere Beziehungen in Instanzdokumenten formuliert werden. Dann werden auch die logischen XQuery-Ausdrücke komplexer.

In den gezeigten Beispielen wurden Anforderungen stets so als logische XQuery-Ausdrücke formuliert, dass sie das gesamte Instanzdokument testen. Zum Testen muss der logische XQuery-Ausdruck einmal für das Instanzdokument ausgewertet werden. Er liefert den logischen Wert `true`, wenn das Instanzdokument der Anforderung entspricht, ansonsten den logischen Wert `false` oder einen Fehler. Das ist die Vorgehensweise, die auch für SXQT verwendet wurde. Dort wurde für den gesamten Trace geprüft, ob er einer Anforderung entspricht. Solche logischen XQuery-Ausdrücke sollen als absolute XQuery-Ausdrücke bezeichnet werden.

10.3 Spezifikation mit relativen XQuery-Ausdrücken

Als Alternative zu absoluten XQuery-Ausdrücken sind für die Spezifikation von Instanzdokumenten auch relative denkbar. Als relativer XQuery-Ausdruck soll ein solcher bezeichnet werden, der sich statt auf das gesamte Instanzdokument nur auf einzelne Elemente oder Attribute bezieht. Von einem solchen Knoten ausgehend wird er formuliert und testet nur für diesen, ob er die Anforderung erfüllt. Daher muss ein relativer XQuery-Ausdruck einmal für jeden Knoten im Instanzdokument ausgewertet werden, auf den er sich bezieht. Er liefert den logischen Wert `true`, wenn die Anforderung für den Knoten erfüllt ist, sonst den logischen Wert `false`.

Im relativen XQuery-Ausdruck wird eine Möglichkeit benötigt zu bestimmen, für welchen Knoten im Instanzdokument die Anforderung geprüft werden soll. Hierzu wurde in absoluten XQuery-Ausdrücken die Standardfunktion `fn:input` verwendet und in SXQT-Ausdrücken die Funktion `opr:tr`. Etwas ähnliches muss auch für relative XQuery-Ausdrücke bereitgestellt werden.

Hierzu könnte festgelegt werden, dass die Input-Sequence diesen Knoten enthält, so dass er mit der Standardfunktion `fn:input` ermittelt werden kann. Das soll jedoch hier nicht geschehen. Die Gründe sind die gleichen, aus denen in Kapitel 7.5.3 die Input-Sequence nicht direkt für den Trace verwendet wurde. Es kann nicht sichergestellt werden, dass es jeder XQuery-Prozessor unterstützt, in der Input-Sequence beliebige Knoten zu übergeben. Daher wurde in Kapitel 7.6.5 zum Zugriff auf den Trace die Funktion `opr:tr` definiert, die Details des Zugriffs auf ihn verbirgt.

Bei relativen XQuery-Ausdrücken soll ebenso vorgegangen werden. Es sei die Funktion `opr:context` definiert, die den Knoten liefert, für den der relative XQuery-Ausdruck die Anforderung prüfen soll. Sie verbirgt Details, wie der Knoten ermittelt wird. Ihre Funktionsdeklaration ist in Abbildung 144 gezeigt. Der Rumpf der Funktion wurde durch drei Punkte ersetzt.

```
define function opr:context()
  as node
{...}
```

Abbildung 144: Deklaration der Funktion `opr:context`

Mit Hilfe von zwei Beispielen soll wieder illustriert werden, wie Anforderungen als relative XQuery-Ausdrücke formuliert werden können. Das erste Beispiel greift die Anforderung aus Kapitel 6.2.2 wieder auf, für das ein absoluter XQuery-Ausdruck im vorherigen Kapitel 10.2 in Abbildung 142 auf Seite 248 gezeigt wurde. Sie legt fest, dass jedes Element `my:a` im Instanzdokument entweder das Attribut `ref` oder ein Kindelement `my:str` haben muss, aber nicht beides gleichzeitig haben darf. Um das für alle Elemente `my:a` zu testen, wurde im absoluten XQuery-Ausdruck ein Allquantor verwendet.

```
fn:not(fn:exists(opr:context()/@ref)
    eq fn:exists(opr:context()/my:str))
```

Abbildung 145: Anforderung an alternative Struktur, als relativer XQuery-Ausdruck

Im relativen XQuery-Ausdruck in Abbildung 145 ist der Allquantor nicht erforderlich. Er ist Elementen mit dem qualifizierten Namen `my:a` zugeordnet und wird daher ohnehin für jedes dieser Elemente geprüft. Daher wird nur noch der Teilausdruck benötigt, der im Allquantor hinter dem Schlüsselwort `satisfies` angegeben ist. In ihm wird statt der Variable des Allquantors nun die Funktion opr:`context` verwendet, um das zu prüfende Element zu ermitteln.

Das zweite Beispiel für einen relativen XQuery-Ausdruck betrachtet eine Anforderung, die ähnlich in Kapitel 6.1.1 vorgestellt wurde. Sie beschreibt eine Beziehung zwischen einem Attribut und den Kindelementen eines Elementes. Das Element `my:x` habe das Attribut `num` sowie Kindelemente `my:y`. Gefordert wird, dass die im Attribut `num` enthaltene Zahl stets gleich der Anzahl der Kindelemente `my:y` ist. Ein Beispiel für ein XML-Fragment, dass ein Element `my:x` enthält, ist in Abbildung 146 gezeigt. Es erfüllt die Anforderung.

```
<my:x num="3" xmlns:my="http://example.org/XYSample">
  <my:y>...</my:y>
  <my:y>...</my:y>
  <my:y>...</my:y>
</my:x>
```

Abbildung 146: Beispiel für XML-Fragment mit Gleichheit von Attribut und Anzahl Kindelemente

In XML-Schema lässt sich die Anforderung nicht ausdrücken. Als relativer XQuery-Ausdruck jedoch schon. Der XQuery-Ausdruck in Abbildung 147 leistet es, wenn er dem Element `my:x` zugeordnet wird. Die Funktion `opr:context` liefert dann das Element `my:x`. Mit Pfadausdrücken werden aus ihm der Wert des Attributes `num` und eine Sequenz seiner Kindelemente `my:y` ermittelt. Aus der Sequenz wird mit der Standardfunktion `fn:count` die Anzahl ihrer Elemente bestimmt und diese mit dem Wert des Attributes `num` verglichen.

```
opr:context()/@num eq fn:count(opr:context()/my:y)
```

Abbildung 147: Anforderung an Gleichheit von Attribut und Anzahl Kindelemente als XQuery-Ausdruck

10.4 Vergleich

Offenbar lassen sich Anforderungen an Instanzdokumente als absolute oder relative XQuery-Ausdrücke formulieren. Nachfolgend soll untersucht werden, wo jeweils die Vor- und Nachteile liegen.

Bei absoluten XQuery-Ausdrücken wird für das Instanzdokument als Ganzes geprüft, ob es der Anforderung entspricht. Das ist von Vorteil, wenn sich auch die Anforderung auf das Instanzdokument als Ganzes bezieht und nicht vor allem auf einzelne Knoten. Daher war ein absoluter XQuery-Ausdruck für die Anforderung aus Abbildung 143 zweckmäßig, mit der das Rootelement des Instanzdokumentes festgelegt wurde.

Relative XQuery-Ausdrücke sind dagegen von Vorteil, wenn sich die Anforderung auf einzelne Knoten bezieht und sie mit ihren Nachfahren oder auch ihren Vorfahren in Beziehung setzt. Ein Vergleich der logischen XQuery-Ausdrücke in Abbildung 142 und Abbildung 145 zeigt, dass dann die relativen XQuery-Ausdrücke kürzer sind, weil dann der Allquantor nicht benötigt wird.

Umgekehrt kann durch Zufügen des Allquantors aus jedem relativen XQuery-Ausdruck ein absoluter konstruiert werden. Der Allquantor muss den relativen XQuery-Ausdruck für jedes Element prüfen, dem der relative XQuery-Ausdruck zugeordnet ist. Dabei muss der Aufruf der Funktion `opr:context` stets durch die Variable des Allquantors ersetzt werden. Weitere Änderungen sind nicht erforderlich. Da außerdem absolute Ausdrücke wie in Abbildung 143 nur schwer als relative darstellbar sind, ist die Verwendung von absoluten XQuery-Ausdrücken mächtiger als die von relativen.

Vorteilhaft sind relative XQuery-Ausdrücke, wenn mit ihnen XML-Schema erweitert werden soll, so dass ein erweitertes XML-Schemadokument einen Namensraum präziser beschreibt als ein nicht erweitertes. Ein XML-Schemavalidator kann solche Erweiterungen mit prüfen. Absolute XQuery-Ausdrücke eignen sich hierzu nicht, weil sie sich auf ein Instanzdokument als ganzes beziehen. XML-Schema hat jedoch keine Elemente, die ein Instanzdokument als ganzes beschreiben. Statt dessen wird in XML-Schema ein Vokabular in Form von Typen, Elementen, Attributen und ähnlichem definiert. Daher sollten in XML-Schema nur logische XQuery-Ausdrücke eingebettet werden, die sich auf solche Teile beziehen, also nur relative.

Somit haben für die Spezifikation von Instanzdokumenten sowohl relative als auch absolute XQuery-Ausdrücke ihre Vorteile. Mit absoluten können auch Anforderungen formuliert werden, die sich mit relativen nicht oder zumindest nicht gut ausdrücken lassen. Die Einbettung in XML-Schema, für die relative XQuery-Ausdrücke benötigt werden, ist jedoch ein entscheidender Vorteil. XML-Schema ist das heute übliche Verfahren zur Deklaration von Strukturen von XML-Dokumenten und gleichzeitig sein Typsystem. Eine „nahtlose“ Integration in dieses ist erstrebenswert, damit Instanzdokumente weiterhin nur mit einem Spezifikationsverfahren spezifiziert werden müssen. Überall dort, wo heute XML-Schemadokumente verwendet werden, können dann erweiterte XML-Schemadokumente zum Einsatz kommen. Daher soll eine solche Einbettung nachfolgend im Detail untersucht werden.

10.5 Einbettung in XML-Schema

Für die Einbettung relativer XQuery-Ausdrücke in XML-Schema kann vieles von den Erkenntnissen aus Kapitel 8 übernommen werden. Dort wurde die Frage untersucht, wie sich Zusätze, insbesondere SXQT-Ausdrücke, in WSDL einbetten lassen. Das schloss eine Untersuchung der Einbettung in XML-Schema ein, weil XML-Schema in WSDL verwendet wird.

So wurde in Kapitel 8.1.3 festgestellt, dass XML-Schemadokumenten Zusätze in Form von Erweiterungselementen und -attributen zugefügt werden können. Diese sollen nach Kapitel 8.2 den Elementen von XML-Schema bzw. WSDL zugefügt werden, die das definieren, auf das sich der Zusatz bezieht. Das muss auch gelten, wenn die Zusätze relative XQuery-Ausdrücke sind, die XML-Schema zugefügt werden sollen.

In Kapitel 8.2 wurde untersucht, welche Elemente von WSDL und XML-Schema als Ablageorte für Zusätze in welchen Fällen in Frage kommen. Die Kapitel 8.2.1 bis 8.2.3 behandelten dazu die Elemente von XML-Schema. Diese Ergebnisse können auf dieses Kapitel übertragen werden, und sollen daher kurz zusammengefasst und auf relative XQuery-Ausdrücke bezogen werden.

Entsprechend der Kapitel 8.2.1 und 8.2.2 kann ein relativer XQuery-Ausdruck vor allem einer Deklaration eines Elementes oder Attributes oder einer Definition von einfachen oder komplexen Typen zugefügt werden. Der relative XQuery-Ausdruck bezieht sich dann vor allem auf die Elemente bzw. Attribute, die so deklariert werden. Bei einer Typdefinition bezieht sich der relative XQuery-Ausdruck auf alle Elemente bzw. Attribute, die den Typ haben.

Ähnlich verhält es sich auch mit den Elementen von XML-Schema, die in Kapitel 8.2.3 untersucht wurden, z. B. `xsd:sequence` und `xsd:restriction`. Sie werden Elementen beigefügt oder verwendet, um Typen zu definieren. Daher beziehen sie sich indirekt auf ein Element oder Attribut. Die relativen XQuery-Ausdrücke seien zu diesem relativ.

Neben der Frage wo Zusätze in XML-Schema oder WSDL eingebettet werden sollen, wurde speziell für SXQT-Ausdrücke in Kapitel 8.3 untersucht wie Zusätze zugefügt werden soll. Zusätze können in XML-Schema als Erweiterungselemente oder -attribute eingebettet werden. In Kapitel 8.3.1 wurde festgestellt, dass für SXQT-Ausdrücke Erweiterungselemente vorzuziehen sind, weil es sich bei ihnen um XQuery-Ausdrücke und damit mehrzeilige Zeichenketten handelt. Diese Entscheidung soll auch auf relative XQuery-Ausdrücke übertragen werden.

Das gilt auch für die Entscheidung aus Kapitel 8.3.2 ob die SXQT-Ausdrücke in der XQuery-Syntax oder in XQueryX gespeichert werden sollten, wobei XQueryX eine Darstellung eines XQuery-Ausdrucks in XML erlaubt. Ebenso wie bei SXQT-Spezifikationen von Webservices ist es auch für Spezifikationen beliebiger Instanzdokumente sehr wichtig, dass Menschen sie gut verstehen können. Das ist bei der XQuery-Syntax eher gegeben als in der für maschinelle Verarbeitung besser geeigneten Syntax XQueryX.

Die genaue Struktur von Erweiterungselementen kann aus Kapitel 8 nicht direkt übernommen werden. Der Grund ist, dass dort statt XML-Schema WSDL erweitert wurde. Erweiterungselemente von WSDL können beliebige Elemente sein, die beliebig Elementen von WSDL zugefügt werden können. Dagegen können Erweiterungselemente in XML-Schema nur innerhalb der beiden Kindelemente des Elementes `xsd:annotation` zugefügt werden. Daher können nur grundlegende Ideen übernommen werden, wie die Möglichkeit logische XQuery-Ausdrücke natürlichsprachlich zu kommentieren, wie es in Kapitel 8.3.3 vorgeschlagen wurde.

Das Element `xsd:annotation` wurde bereits in Kapitel 8.1.3 vorgestellt. Es kann den meisten Elementen von XML-Schema als erstes Kindelement zugefügt werden. Selbst hat es die beiden Kindelemente `xsd:documentation` und `xsd:appinfo`, die es beliebig oft und in beliebiger Reihenfolge enthalten kann. Dabei ist das Element `xsd:documentation` für Zusatzinformationen für menschliche Leser bestimmt, wie es die natürlichsprachlichen Kommentare zu den Anforderungen sind. Das Element

xsd:appinfo sollte dagegen Zusatzinformationen für eine maschinelle Verarbeitung enthalten. Die relativen XQuery-Ausdrücke können als solche angesehen werden, auch wenn sie zusätzlich von Menschen gelesen und interpretiert werden.

So wäre es naheliegend, relative XQuery-Ausdrücke direkt als Zeichenkette innerhalb von Elementen xsd:appinfo zu speichern. Es wird jedoch eine Möglichkeit benötigt zu erkennen, dass das Element xsd:appinfo einen relativen XQuery-Ausdruck des hier beschriebenen Spezifikationsverfahrens enthält. Daher sollte das Element xsd:appinfo ein Kindelement haben, an dessen qualifiziertem Namen das zu erkennen ist. Dieses enthält den relativen XQuery-Ausdruck. Es habe den lokalen Namen assert im Namensraum http://ti5.tu-harburg.de/venzke/20021015/xmlschema-extension, für den nachfolgend das Präfix xse verwendet werden soll. Dieser Namensraum lässt sich sehr einfach in XML-Schema definieren, wie das XML-Schemadokument in Abbildung 148 zeigt. Die sich ergebende Struktur für das Element xsd:annotation wird in Abbildung 149 veranschaulicht.

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://ti5.tu-harburg.de/venzke/20021015/xmlschema-extension" >

  <xsd:element name="assert" type="xsd:string" />
</xsd:schema>
```

Abbildung 148: XML-Schemadokument des Erweiterungselementes für XML-Schema

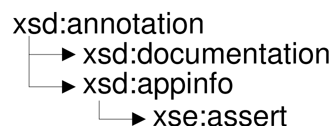

Abbildung 149: Struktur des Elementes xsd:annotation mit Erweiterungselement

Um mehrere relative XQuery-Ausdrücke einem Element von XML-Schema zuzuordnen, kann dessen Kindelement xsd:annotation mehrere Kindelemente xsd:appinfo haben. Jedes enthält in einem Kindelement xse:assert einen relativen XQuery-Ausdruck. Zu jedem Element xsd:appinfo kann dem Element xsd:annotation ein Kindelement xsd:documentation zugefügt werden, in dem der relative XQuery-Ausdruck in natürlicher Sprache beschrieben ist. Für andere Anwendungen können im Element xsd:annotation außerdem weitere Kindelemente xsd:documentation oder xsd:appinfo vorhanden sein, die weitere Kommentare bzw. andere Zusätze für die maschinelle Verarbeitung enthalten.

Die beschriebene Einbettung von relativen XQuery-Ausdrücken in XML-Schema soll an einem Beispiel illustriert werden. In Kapitel 10.3 wurde in Abbildung 147 auf Seite 250 ein relativer XQuery-Ausdruck gezeigt. Er legt für ein Element fest, dass dessen Attribut num stets die Anzahl seiner Kindelemente my:y enthalten muss. Der relative XQuery-Ausdruck ist in Abbildung 150 einem komplexen Typen zugefügt worden[64]. Daher gilt die Anforderung für jedes Element, das diesen Typ hat. Eine zweite Anforderung ist als weiteres Paar von Elementen xsd:documentation und xsd:appinfo angedeutet, wobei der Inhalt durch drei Punkte („...“) ersetzt wurde.

[64] Der Query-Prolog wurde in Abbildung 147 zur Verkürzung nicht mit gezeigt. In Abbildung 150 wurde er ergänzt.

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xse="http://ti5.tu-harburg.de/venzke/20021015/xmlschema-extension"
    xmlns:my="http://example.org/XYSample"
    targetNamespace="http://example.org/XYSample" >

  <xsd:element name="y" type="xsd:string" />

  <xsd:complexType name="xType">
    <xsd:annotation>
      <xsd:documentation>
        Das Attribut num muss stets die
        Anzahl der Kindelemente my:y enthalten.
      </xsd:documentation>
      <xsd:appinfo>
        <xse:assert>
          declare namespace
                  opr="http://ti5.tu-harburg.de/venzke/20021015/operations"
          opr:context()/@num eq fn:count(opr:context()/my:y)
        </xse:assert>
      </xsd:appinfo>
      <xsd:documentation> ... </xsd:documentation>
      <xsd:appinfo>
        <xse:assert> ... </xse:assert>
      </xsd:appinfo>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element ref="my:y"
                   minOccurs="0"
                   maxOccurs="unbounded" />
    </xsd:sequence>
    <xsd:attribute name="num" type="xsd:nonNegativeInteger" />
  </xsd:complexType>

  <xsd:element name="x" type="my:xType" />
</xsd:schema>
```

Abbildung 150: Beispiel für XML-Schemadokument mit Erweiterungselementen

10.6 Validation gegen erweitertes XML-Schema

In Kapitel 10.4 wurde bereits festgestellt, dass ein XML-Schemavalidator auch die relativen XQuery-Ausdrücke prüfen könnte, die in ein XML-Schemadokument eingebettet sind. Das war ein Vorteil von relativen gegenüber absoluten XQuery-Ausdrücken und soll hier kurz ausgeführt werden.

Unabhängig von logischen XQuery-Ausdrücken ist es heute üblich mit XML-Schemavalidatoren[65] zu prüfen, ob ein Instanzdokument einem bzw. auch mehreren XML-Schemadokumenten entspricht. Regeln für diese Validation, nach denen das zu prüfen ist, werden im Detail im Standard von XML-Schema [Thompson 01] angegeben. Ist das XML-Schemadokument wie im vorherigen Kapitel 10.5 um relative XQuery-Ausdrücke erweitert, werden diese von heute üblichen XML-Schemavalidatoren ignoriert. Um sie zu berücksichtigen, müssen die XML-Schemavalidatoren geeignet erweitert werden. Hierzu muss zur Auswertung der relativen XQuery-Ausdrücke ein XQuery-Prozessor zur Verfügung stehen.

[65] Es gibt eine Vielzahl von XML-Schemavalidatoren. [Sperberg-McQueen 02] verweist auf eine Reihe von ihnen.

Zusätzlich zu den Regeln aus dem Standard von XML-Schema muss ein erweiterter XML-Schemavalidator für jedes XML-Element oder Attribut alle relativen XQuery-Ausdrücke prüfen, die sich auf dieses beziehen. Wie in Kapitel 10.5 festgestellt, können die relativen XQuery-Ausdrücke z. B. in den Elementen zur Deklaration von XML-Elementen bzw. Attributen enthalten sein oder in den Elementen zur Definition von deren Typen.

Der XML-Schemavalidator muss XML-Elementen und Attributen in Instanzdokumenten ihren Deklarationen im XML-Schemadokument zuordnen, ebenso wie den Definitionen verwendeter Typen. Das ist erforderlich, um die Regeln des Standards von XML-Schema zu prüfen. Für die Erweiterung muss der XML-Schemavalidator in Elementen solcher Deklarationen nach Nachfahren `xsd:annotation/xsd:appinfo/xse:assert` suchen. Jeden dort gefundenen relativen XQuery-Ausdruck muss er relativ zum XML-Element bzw. Attribut auswerten. Wird dabei der logische Wert `true` geliefert, kann die Validation fortgesetzt werden. Ist das Ergebnis dagegen der logische Wert `false` oder wird ein Fehler geliefert, ist die Validation des Instanzdokumentes fehlgeschlagen.

Somit ist es nicht sehr aufwendig einen XML-Schemavalidator so zu erweitern, dass er XML-Schemadokumenten zugefügte relative XQuery-Ausdrücke mit berücksichtigt. Das macht es attraktiv, XML-Schema in der in Kapitel 10.5 gezeigten Art zu erweitern und so seine Beschreibungsfähigkeit deutlich zu verbessern. Der Standard von XML-Schema muss hierzu nicht geändert werden, weil in XML-Schema vorgesehene Erweiterungselemente verwendet werden. Trotzdem wäre diese Erweiterung direkt in den vielen Anwendungsfeldern einsetzbar, in denen heute XML-Schema verwendet wird.

11 Verwandte Arbeiten zur Spezifikation von Webservices

Das in Kapitel 7 entwickelte Spezifikationsverfahren SXQT für Schnittstellen von Webservices ist ein wichtiges Ergebnis dieser Arbeit. Schnittstellen können präziser beschrieben werden als nur mit WSDL, was in Kapitel 7.7 an verschiedenen Beispielen gezeigt wurde. Das lässt sich ebenfalls mit unterschiedlichen Spezifikationsverfahren aus der Literatur erreichen, die in diesem Kapitel vorgestellt und SXQT verglichen werden sollen. Verwandte Arbeiten zur automatischen Validation wurden bereits in Kapitel 9 vorgestellt.

Die Spezifikationsverfahren haben unterschiedliche Ziele und erlauben die Beschreibung von Webkomponenten und ihre Schnittstellen auf unterschiedliche Arten. Ziel von WSCL (Kapitel 11.1.1) und WSCI (Kapitel 11.1.2) ist, dynamisches Verhalten von Schnittstellen von Webservices zu beschreiben. BPML (Kapitel 11.2.1), BPEL4WS (Kapitel 11.2.2) und BPSS (Kapitel 11.2.3) haben dagegen das Ziel Geschäftsprozesse zu beschreiben. Indirekt können jedoch auch mit ihnen Schnittstellen von Webservices beschrieben werden. XL (Kapitel 11.3) ist eine Sprache zur Implementierung von Webkomponenten. Spezifikationen von Schnittstellen können XL-Programmen als Invarianten zugefügt werden, die den SXQT-Ausdrücken dieser Arbeit ähnlich sind. Mit DAML-S (Kapitel 11.4) werden dagegen Webservices mit Wissensrepräsentationstechniken spezifiziert, die es auch erlauben Seiteneffekte zu beschreiben, die durch Webservices in der realen Welt verursacht werden. Schließlich kann mit UDDI (Kapitel 11.5) angegeben werden, welche Spezifikationen ein Webservice erfüllt. Hierzu stellt es jedoch kein eigenes Spezifikationsverfahren zur Verfügung, sondern verweist auf Spezifikationen anderer Spezifikationsverfahren.

11.1 Spezifikation von dynamischem Verhalten in Schnittstellen

Zur Spezifikation dynamischen Verhaltens von abstrakten Schnittstellen von Webservices wurden vom W3C die beiden Sprachen WSCL und WSCI veröffentlicht. Beide erlauben, Anforderungen über Reihenfolgen von Operationsaufrufen zu beschreiben. Sie unterscheiden sich im Modell, nach dem alle Anforderungen an eine Schnittstelle zusammen formuliert werden. Im Vergleich zu SXQT haben beide jedoch eine eingeschränkte Ausdrucksfähigkeit.

11.1.1 Web Services Conversation Language (WSCL)

Die Web Services Conversation Language (WSCL) erlaubt, abstrakte Schnittstellen zu definieren, für die erlaubte Reihenfolgen von Operationsaufrufen festgelegt werden können. Hierzu wird ein Modell von Operationen und Transistionen verwendet. Anders als bei SXQT kann jedoch nichts weiteres spezifiziert werden.

WSCL wurde in der Version 1.0 von Hewlett-Packard [HP 03] entwickelt und vom W3C als W3C-Note [Banerji 02] veröffentlicht. Der W3C-Note wurden benötigte Informationen für diese Arbeit entnommen.

11.1.1.1 Spezifikation von Konversationen

Ziel von WSCL ist, für abstrakte Schnittstellen zu beschreiben, welche Nachrichten in welchen Reihenfolgen zwischen zwei Webkomponenten ausgetauscht werden dürfen. Hierzu werden sogenannte Konversationen deklariert, die einen logisch in sich abgeschlossenen Nachrichtenaustausch zwischen zwei Webkomponenten über eine abstrakte Schnittstelle repräsentieren. Konversationen werden in Form von Operationen und Transistionen definiert. Operationen beschreiben die XML-Nachrichten, die für einen Operationsaufruf ausgetauscht werden. Transistionen dienen zur Beschreibung erlaubter Reihenfolgen von Operationsaufrufen. WSCL hat keine Mechanismen zur Beschreibung protokollspezifischer Eigenschaften von Schnittstellen, also für gebundene Schnittstellen (siehe Kapitel 2.5). Hierzu müssen andere Sprachen verwendet werden, z. B. WSDL.

Deklarationen von Operationen, die im Kontext von WSCL als Interaktionen bezeichnet werden, haben Ähnlichkeit mit Operationen in abstrakten Schnittstellen von WSDL. Mit Hilfe von XML-Schema werden bis zu zwei XML-Nachrichten (Request und Response) beschrieben, die für einen Operationsaufruf ausgetauscht werden. Für den Response können mehrere Nachrichtentypen angegeben werden, die dann alternativ erlaubt sind.

In welchen Reihenfolgen Operationen innerhalb einer Konversation aufgerufen werden dürfen, wird mit Hilfe von Transistionen definiert. Für eine Transistion wird eine Quell- und eine Zieloperation angegeben. Sie legt fest, dass die Zieloperation nur aufgerufen werden darf, wenn vorher die Quelloperation aufgerufen wurde. Operationen und Transistionen können als Knoten und Kanten eines gerichteten Graphen verwendet werden, ähnlich wie Zustände und Zustandsübergänge von endlichen Automaten. Das ist in Abbildung 151 veranschaulicht. Zusammen mit einer ausgezeichneten Startoperation, mit der die Konversation beginnt und einer Endoperationen, mit der sie endet, können so die erlaubten Reihenfolgen von Operationsaufrufen spezifiziert werden.

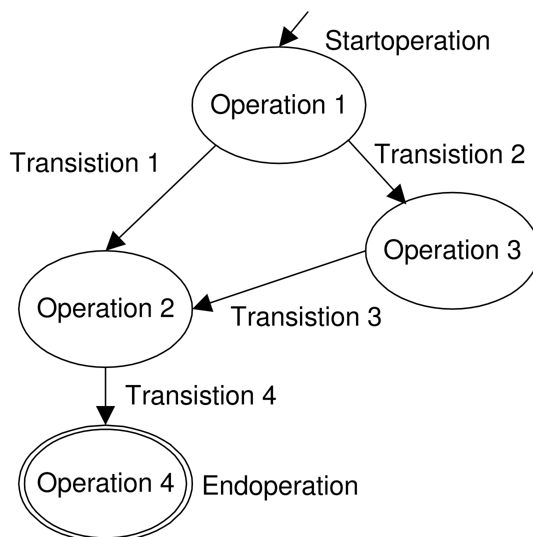

Abbildung 151: Beispiel für gerichteten Graph einer Konversation von WSCL

Eine solche Spezifikation ist nichtdeterministisch, da mehrere Transistionen die gleiche Quelloperation haben dürfen. Der Nichtdeterminismus kann eingeschränkt werden, wenn die Quelloperation unterschiedliche Nachrichtentypen als Response erlaubt. Dann kann für die Transistionen angegeben werden, dass nur ihre Zieloperation aufgerufen werden darf, wenn der Response der Quelloperation einen bestimmten Nachrichtentyp hatte.

Alle Operationen und Transistionen, die gemeinsam eine Konversation beschreiben, werden in einem XML-Dokument beschrieben. Sie sind Kindelemente des Rootelementes `wscl:Conversation`[66], in dessen Attributen die Start- und Endoperation angegeben sind.

Da eine Webkomponente gleichzeitig mit mehreren Webkomponenten in unterschiedlichen Konversationen Nachrichten austauschen kann, muss die sie erkennen können, zu welcher Konversation eine Nachricht gehört. Dazu wird in [Banerji 02] angenommen, dass in der gebundenen Schnittstelle den Nachrichten Bezeichner zugefügt werden, die die Konversation identifizieren. Im Falle von SOAP sind die Bezeichner in Headereinträgen enthalten.

Zur Definition von gebundenen Schnittstellen stellt WSCL keine eigenen Mechanismen bereit. In [Banerji 02] wird jedoch diskutiert, welche Möglichkeiten es gäbe, WSDL hierzu zu verwenden. Es wird festgestellt, dass Operationen von abstrakten Schnittstellen sowohl in WSDL als auch in WSCL definiert werden. Eine solche redundante Definition könnte akzeptiert werden. Alternativ könnten abstrakte Schnittstellen auch nur in WSCL oder WSDL definiert werden. Dann müsste entweder aus Definitionen von gebundenen Schnittstellen in WSDL auf in WSCL definierte Operationen verwiesen werden oder die Transistionen von WSCL müssten auf in WSDL definierte Operationen verweisen.

11.1.1.2 Vergleich mit SXQT

Viele in SXQT formulierbare Anforderungen, lassen sich WSCL nicht ausdrücken. In WSCL können nur Reihenfolgen von Operationsaufrufen innerhalb von Konversationen beschrieben werden und nur unabhängig von Inhalten von Nachrichten. Es können weder über WSDL hinausgehende Anforderungen an einzelne Request-/Response-Paare oder an einzelne SOAP-Nachricht beschrieben werden. Die Beschreibung von Headereinträgen und SOAP-Fehlernachrichten ist überhaupt nicht möglich. Außerdem müssen alle Anforderungen zusammen als ein Modell von Operationen und Transistionen formuliert werden.

Dabei verfolgen sowohl WSCL als auch SXQT das Ziel einer über WSDL hinausgehenden Beschreibung von Schnittstellen von Webservices. WSCL leistet das aber zunächst unabhängig von WSDL. In [Banerji 02] wird lediglich diskutiert, wie eine Kombination möglich wäre. Für SXQT ist dagegen festgelegt, dass nur solche Anforderungen als SXQT-Ausdrücke formuliert werden, die nicht schon in WSDL ausgedrückt sind.

Mit WSCL können ebenso wie mit SXQT die Reihenfolgen von Operationsaufrufen festgelegt werden. Bei WSCL ist das aber nicht abhängig vom Inhalt der Nachricht möglich. Auch können in WSCL keine Anforderungen an Beziehungen zwischen Operationsaufrufen unterschiedlicher Konversationen ausgedrückt werden, wie sie z. B. für Sperrmechanismen nötig sind und in Kapitel 6.1.3 vorgestellt wurden.

Unabhängig von Reihenfolgen von Operationsaufrufen fügt WSCL gegenüber WSDL keine Ausdrucksfähigkeit zu. So können weder Anforderungen über Request-/Response-Paare (siehe Kapitel 6.1.2) oder SOAP-Nachrichten (siehe Kapitel 6.1.1, 6.2.1 und 6.2.2) formuliert werden.

Ein wesentlicher Unterschied ist, dass mit WSCL abstrakte Schnittstellen spezifiziert werden, während das im SXQT für gebundene geschieht. Abstrakte Schnittstellen zu

[66] Das Präfix `wscl` stehe für den Namensraum `http://www.w3.org/2002/02/wscl10`.

spezifizieren, hat den Vorteil, dass Spezifikationen für viele Protokollbindungen wiederverwendet werden können. Es führt jedoch dazu, dass mit WSCL weder Anforderungen an Headereinträge (siehe Kapitel 6.2.5) noch an SOAP-Fehlernachrichten (siehe Kapitel 6.2.4) ausgedrückt werden können. Beides ist SOAP-spezifisch und existiert in den von WSCL spezifizierten abstrakten Schnittstellen nicht. Ebenso wie die zuvor genannten Anforderungen lassen sich diese in SXQT ausdrücken.

Ein grundsätzlicher Unterschied besteht auch darin, wie Anforderungen in Spezifikationen ausgedrückt werden. In SXQT werden sie einzeln als SXQT-Ausdrücke formuliert. In WSCL muss dagegen zunächst aus allen Anforderungen gemeinsam ein Modell aus Operationen und Transistionen gebildet werden. Das erschwert es, Anforderungen aus unterschiedlichen Quellen zusammenzufügen, also z. B. häufig benötigte Standardanforderungen direkt einer Spezifikation zuzufügen.

11.1.2 Web Services Choreography Interface (WSCI)

Wie WSCL erlaubt es auch die Sprache Web Services Choreography Interface (WSCI) für abstrakte Schnittstellen festzulegen, in welchen Reihenfolgen Operationen aufgerufen werden dürfen. Die Reihenfolgen werden mit einer Art abstrakter Implementierung definiert. Wieder gibt es in SXQT formulierbare Anforderungen, die nicht ausgedrückt werden können.

Ebenfalls wie WSCL wurde WSCI als W3C-Note [Arkin 02a] veröffentlicht und aus ihr Informationen für diese Arbeit entnommen. Entwickelt wurde WSCI von Mitarbeitern der Unternehmen Intalio [Intalio 03], BEA Systems [BEA 03], SAP [SAP 02a] und Sun Microsystems [Sun 02a].

11.1.2.1 Spezifikation von WSCI-Schnittstellen

Ziel von WSCI ist, in WSDL beschriebenen, abstrakten Schnittstellen Beschreibungen dynamischen Verhaltens zuzufügen. Die Beschreibung wird WSDL-Dokumenten in Form von WSCI-Schnittstellen zugefügt. Beschrieben werden mögliche WSCI-Prozesse, die ähnlich den Konversationen von WSCL einen logisch in sich abgeschlossenen Nachrichtenaustausch beschreiben. Definiert werden WSCI-Prozesse als sogenannte Aktivitäten, die aus anderen Aktivitäten mit Sprachelementen zusammengesetzt werden, die eine sequentielle, nebenläufige, alternative oder wiederholte Ausführung der Aktivitäten beschreiben. Operationsaufrufe werden als elementare Aktivitäten angesehen. Ergänzt um die Beschreibung von Ausnahmen und Transaktionen können mit WSCI Anforderungen formuliert werden, für die das mit WSCL nicht möglich ist.

Sprachelemente von WSCI werden wie SXQT-Ausdrücke WSDL-Dokumenten zugefügt. Es handelt sich um Elemente `wsci:interface`[67], mit denen WSCI-Schnittstellen beschrieben werden. Aus Sprachelementen von WSCI wird auf Definitionen von WSDL verwiesen, insbesondere auf Definitionen abstrakter Schnittstellen und Operationen. Durch Zufügen von WSCI-Schnittstellen wird WSDL um Beschreibungen von dynamischem Verhalten erweitert.

WSCI-Schnittstellen enthalten eine oder mehrere Beschreibungen von WSCI-Prozessen, die einen in sich logisch abgeschlossenen Nachrichtenaustausch darstellen. WSCI-Prozesse haben Ähnlichkeiten mit Konversationen von WSCL, werden aber anders beschrieben. Gestartet werden können sie wie Konversationen implizit durch den

[67] Das Präfix `wsci` stehe für den Namensraum `http://www.w3.org/2002/07/wsci10`.

Empfang einer Nachricht. Sie können jedoch auch explizit durch andere WSCI-Prozesse synchron oder asynchron gestartet werden, was die Wiederverwendung von Prozessbeschreibungen erlaubt.

Statt wie in WSCL mit Transistionen werden WSCI-Prozesse mit Hilfe sogenannter Aktivitäten beschrieben. WSCI-Prozesse sind selbst komplexe Aktivitäten. Komplexe Aktivitäten werden aus einfacheren Aktivitäten zusammengesetzt. Hierzu stellt WSCI Sprachelemente zur Verfügung. Das Sprachelement `Sequence` schreibt z. B. vor, dass die einfacheren Aktivitäten sequentiell hintereinander ausgeführt werden müssen. So müssen im Beispiel des WSCI-Prozesses in Abbildung 152 zwei Aktivitäten hintereinander ausgeführt werden. Neben `Sequence` gibt es Sprachelemente, mit denen ausgedrückt wird, dass die einfacheren Aktivitäten nebenläufig, nur eine der Aktivitäten abhängig von Bedingungen bzw. von Ereignissen oder die Aktivitäten mehrfach ausgeführt werden.

```
<wsci:process xmlns:wsci="http://www.w3.org/2002/07/wsci10">
              name="PlanAndBookTrip" instantiation="message">
  <wsci:sequence>
    <wsci:action name="ReceiveTripOrder"
                 role="tns:TravelAgent"
                 operation="tns:TAtoTraveler/OrderTrip">
    </wsci:action>
    <wsci:action name="ReceiveConfirmation"
                 role="tns:TravelAgent"
                 operation="tns:TAtoTraveler/bookTickets">
      <wsci:correlate correlation="tns:itineraryCorrelation" />
    </wsci:action>
  </wsci:sequence>
</wsci:process>
```

Abbildung 152: Beispiel für Beschreibung eines WSCI-Prozesses, vereinfacht aus [Arkin 02a][68]

WSCI stellt unterschiedliche elementare Aktivitäten zur Verfügung, von denen `Action` die wichtigste ist. Mit `Action` werden Aufrufe von Operationen modelliert, notiert als Element `wsci:action`, wie in Abbildung 152 gezeigt. Um welche Operation es sich handelt, wird im Attribut `wsci:action/@operation` angegeben. Es enthält den qualifizierten Namen einer in WSDL definierten abstrakten Schnittstelle, gefolgt vom Namen einer darin enthaltenen Operation, getrennt durch das Zeichen `/`.

Mit dem Kindelement `wsci:correlation` wird angegeben, wie erkannt wird, dass eine empfangene Nachricht zum beschriebenen WSCI-Prozess gehört. Das ist erforderlich, weil mehrere WSCI-Prozesse gleichzeitig die Kommunikation z. B. mit mehreren Webclients beschreiben können. Das Attribut `wsci:correlation/@correlation` verweist hierzu auf einen sogenannten Selector, mit dem der Teil der Nachricht angegeben wird, der den WSCI-Prozess identifiziert.

WSCI verwendet das Konzept von Kontexten, um die Umgebung zu beschreiben, in der Aktivitäten ausgeführt werden, was die Beschreibung von Ausnahmen der Verarbeitung und transaktionalem Verhalten einschließt. Für komplexe Aktivitäten können eigene Kontexte deklariert werden. Für die Beschreibung von Ausnahmen enthält der Kontext Ausnahme-Handler. Tritt in der Aktivität eine Ausnahmesituation auf, wird die Aktivität abgebrochen und die Ausnahme durch Ausnahme-Handler behandelt.

[68] Das Beispiel ist Ausschnitt eines Beispiels aus [Arkin 02a], Kapitel 1.7.5. Es wurde vereinfacht und das Präfix `wsci` zugefügt. Das Präfix `tns` steht für den durch das WSDL-Dokument definierten Namensraum, in dem der WSCI-Prozess enthalten ist.

In der Deklaration des Kontextes kann festgelegt werden, dass die komplexe Aktivität als Transaktion zu betrachten ist. Unterschieden werden atomare Transaktionen von Open-Nested-Transaktionen. Für atomare Transaktionen wird angenommen, dass unabhängig von WSCI ein Mechanismus existiert, der sie realisiert. Open-Nested-Transaktionen basieren dagegen auf Kompensation. Sie bestehen aus atomaren und andere Open-Nested-Transaktionen. Um eine Open-Nested-Transaktion abzubrechen (Rollback) oder zu kompensieren, müssen die enthaltenen Transaktionen abgebrochen oder nach deren Ende kompensiert werden.

11.1.2.2 Vergleich mit SXQT

Mit WSCI können wie mit WSCL Anforderungen an Reihenfolgen von Operationen ausgedrückt werden. Für Ausnahmen und Transaktionen werden jedoch zusätzliche Sprachelemente zur Verfügung gestellt. Anforderungen an SOAP-Nachrichten und Request-/Response-Paare, insbesondere Headereinträge und SOAP-Fehlernachrichten können dagegen wie in WSCL nicht genauer beschrieben werden als mit WSDL. Auch müssen alle Anforderungen zusammen als ein Modell formuliert werden. Dazu werden viele, spezielle Sprachelemente verwendet, mit denen jeweils nur bestimmte Klassen von Anforderungen beschrieben werden können.

Wie bei SXQT werden auch bei WSCI WSDL-Dokumenten Erweiterungen zugefügt. Beide definieren aus Sicht von WSDL Erweiterungselemente. WSCI fügt diese als Kindelemente von `wsdl:definitions` auf gleicher Ebene wie abstrakte und gebundene Schnittstellen zu, während SXQT sie als Kindelemente von `wsdl:binding` zufügt und damit die gebundenen Schnittstellen erweitert.

WSCI erweitert WSDL, so dass Anforderungen an erlaubte Reihenfolgen von Nachrichten beschrieben werden können. WSCI-Prozesse beschreiben dabei stets einen logisch in sich abgeschlossenen Nachrichtenaustausch. Wie bei WSCL ist es jedoch nicht möglich, Anforderungen an Beziehungen zwischen WSCI-Prozessen zu beschreiben. Damit kann z. B. der in Kapitel 6.1.3 vorgestellte Sperrmechanismus der Webkomponenten für eine Internetzeitung nicht spezifiziert werden.

Nicht in WSDL ausdrückbare Anforderungen an einzelne SOAP-Nachrichten und einzelne Request-/Response-Paare, insbesondere an SOAP-Fehlernachrichten oder Headereinträge können in WSCI ebenso wie in WSCL nicht formuliert werden. Hierzu sind keine Sprachelemente vorgesehen. WSCI-Schnittstellen werden wie Konversationen von WSCL im Sinne abstrakter Schnittstellen definiert. Da diese von SOAP unabhängig sind, können Details von SOAP wie SOAP-Fehlernachrichten und Headereinträge nicht beschrieben werden. Daher können die in den Kapiteln 6.1.1, 6.1.2, 6.2.1, 6.2.2, 6.2.4 und 6.2.5 vorgestellten Anforderungen mit WSCI nicht beschrieben werden. Mit SXQT ist das möglich.

Wie bei WSCL müssen auch bei WSCI zunächst alle Anforderungen zusammen als ein Modell formuliert werden. Dazu wird eine Art abstrakter Implementierung erstellt, die die erlaubten Reihenfolgen von Operationsaufrufen beschreibt. Unabhängig voneinander können Anforderungen nicht formuliert werden, was in SXQT möglich ist. Das erschwert das Zufügen von Anforderungen aus unterschiedlichen Quellen oder Standardanforderungen. Möglich ist lediglich, mit Hilfe mehrerer WSCI-Schnittstellen unterschiedliche Nutzungsszenarien einer abstrakten Schnittstelle zu beschreiben.

WSCI stellt eine Vielzahl von Sprachelementen zur Verfügung, mit denen jeweils nur eine bestimmte Art von Anforderungen formuliert werden kann. So gibt es mehrere Sprachelemente zum Ausdrücken von Nebenläufigkeit, weitere für eine alternative oder

sequentielle Ausführung von Aktivitäten. Spezielle Sprachelemente wurden auch für Ausnahmen und Transaktionen definiert.

Mit SXQT werden dagegen alle Anforderungen mit den gleichen, einheitlichen Sprachelementen von XQuery beschrieben. Mit ihnen lassen sich nicht nur Reihenfolgen von Operationen ausdrücken, sondern andere Klassen von Anforderungen wie z. B. an Inhalte von einzelnen SOAP-Nachrichten. Für Klassen von Anforderungen, die bei der Entwicklung von SXQT nicht berücksichtigt wurden, müssen keine neue Sprachelemente definiert werden.

Spezielle Sprachelemente für unterschiedliche Klassen von Anforderungen erschweren auch die Implementierung von Werkzeugen wie dem im Kapitel 9 vorgestellten Validator. Mit WSCI müsste der Validator für jedes Sprachelement Programmcode enthalten, der prüft, ob eine Beobachtung der Spezifikation entspricht. Bei SXQT reicht es dagegen, die SXQT-Ausdrücke für den beobachteten Trace auszuwerten. Damit erscheint die Verwendung einheitlicher Sprachelemente von SXQT vorteilhafter als die spezielleren von WSCI.

11.2 Spezifikation von Geschäftsprozessen

Drei Sprachen wurden definiert, um Geschäftsprozesse zu spezifizieren, die durch Kommunikation zwischen Webkomponenten abgewickelt werden. Mit ihnen können indirekt auch Schnittstellen zwischen Webkomponenten beschrieben werden.

Der Begriff Geschäftsprozess stammt aus der Betriebswirtschaft. In [Schwarze 97] wird er wie folgt definiert:

> *„Ein Geschäftsprozeß (Unternehmensprozeß, Business Process)*
> - *besteht aus logisch zusammengehörigen Vorgängen, die man als Vorgangskette bezeichnet,*
> - *ist ziel- und ergebnisorientiert,*
> - *ist verbunden mit dem Austausch von Informationen bzw. Leistungen zwischen Objekten in der Organisation,*
> - *wird durch ein auslösendes Startereignis aktiviert (Informationsinput)*
> - *wird durch ein Endereignis beendet (Informationsoutput).“*[69]

Für diese Arbeit relevant sind Geschäftsprozesse, die durch Kommunikation zwischen Webkomponenten abgewickelt werden. Die „Objekte der Organisation“ werden dann durch Webkomponenten realisiert. Sie sind es, die Vorgänge des Geschäftsprozesses ausführen. Der Austausch von Informationen wird durch Austausch von Nachrichten zwischen den Webkomponenten realisiert. Auch das Startereignis kann der Empfang einer Nachricht sein.

Die nachfolgend vorgestellten Sprachen unterscheiden sich darin, wie sie Geschäftsprozesse modellieren. In BPML (Kapitel 11.2.1) und BPEL4WS (Kapitel 11.2.2) wird jeder Vorgang durch einen Operationsaufruf gestartet. Als Geschäftsprozess wird das Verhalten einer Webkomponente beschrieben, die für jeden Vorgang die zugehörige Operation aufruft. In BPSS (Kapitel 11.2.3) wird dagegen der Schwerpunkt auf die ausgetauschten Nachrichten gelegt. Es wird beschrieben, welche Geschäftsdokumente in einem Geschäftsprozess in welcher Reihenfolge zwischen den beteiligten Webkomponenten ausgetauscht werden.

---

[69] Zitat aus [Schwarze 97], Seite 168.

In beiden Fällen können über die Geschäftsprozesse auch Schnittstellen von Webkomponenten beschrieben werden. Offensichtlich ist das bei BPSS. Eine Festlegung welche Geschäftsdokumente (Nachrichten) in welcher Reihenfolge zwischen zwei Webkomponenten ausgetauscht werden, beschreibt die Schnittstelle zwischen ihnen. In BPML und BPEL4WS erfolgt die Beschreibung dagegen indirekt. Beschrieben wird das Verhalten einer abstrakten Webkomponente. Ihr Verhalten, also welche Nachrichten sie in welcher Reihenfolge austauscht, ist für die beschriebene Schnittstelle verbindlich.

11.2.1 Business Process Modelling Language (BPML)

Ziel der Business Process Modelling Language (BPML) ist die Beschreibung von Geschäftsprozessen. Indirekt können auch Schnittstellen von Webservices beschrieben werden. BPML ist im Wesentlichen eine Obermenge von WSCI. Zusätzliche Sprachelemente z. B. für Variablen, Wertzuweisung, Ausdrücke und Inhalte von Nachrichten erhöhen ihre Ausdrucksfähigkeit für die Spezifikation von Schnittstellen. Trotzdem gibt es Anforderungen, die in BPML nicht ausgedrückt werden können, obwohl das in SXQT möglich ist. Zum Verständnis der in der Spezifikation ausgedrückten Anforderungen müssen diese erst wieder aus der abstrakten Implementierung abgeleitet werden, mit der der Geschäftsprozess beschrieben ist.

Der Standard von BPML [Arkin 02b] wurde von der Organisation BPMI.org [BPMI 03] veröffentlicht. Er dient als Grundlage für die nachfolgende Untersuchung. Mitgewirkt haben an ihr Mitarbeiter der Unternehmen Intalio [Intalio 03], SAP [SAP 02a] und Sun Microsystems [Sun 02a], Versata [Versata 03], CSC [CSC 03] sowie SeeBeyond [SeeBeyond 03]. Interessant ist, dass fünf der Autoren auch Autoren der W3C-Note von WSCI sind.

11.2.1.1 BPML als Obermenge von WSCI

Durch die Übereinstimmung in fünf Autoren, ist BPML im Wesentlichen eine Obermenge von WSCI. Zusätzliche Sprachelemente erlauben, Prozesse so vollständig zu beschreiben, dass Spezifikationen durch Interpreter ausgeführt werden können. Zugefügt werden vor allem Variablen, sowie ein Sprachelement für die Wertzuweisung. XPath wird als Sprache für Ausdrücke unterstützt. Es wird eine Bibliothek von BPML-spezifischen Funktionen zur Verfügung gestellt. Außerdem gibt es Sprachelemente für die Synchronisation nebenläufiger Aktivitäten, zum periodischen Aufrufen von Prozessen sowie zusätzliche Sprachelemente für die Ausnahmebehandlung. Ein Sprachelement analog zu WSCI-Schnittstellen fehlt.

Der offensichtlichen Ähnlichkeit zwischen BPML und WSCI stehen ihre unterschiedlichen Ziele gegenüber. Im FAQ [SAP 02b] zu WSCI des Unternehmens SAP wird festgestellt, dass WSCI zu BPML komplementär ist. WSCI beschreibt Interaktionen zwischen Webkomponenten, während BPML die dahinterliegenden Geschäftsprozesse beschreibt[70]. Für beide Ziele werden aber die gleichen Sprachelemente verwendet.

Nach [Arkin 02b] werden mit BPML ausführbare Geschäftsprozesse sowie unterstützende Entitäten spezifiziert. Es wird festgestellt, dass BPML für unterschiedliche Ziele verwendet werden kann. Die Spezifikation komplexer Webservices ist nur eine von ihnen. Dass Geschäftsprozesse „ausführbar" sind, bedeutet, dass Interpreter eine Spezifikation als Programm ausführen können. Damit ist BPML auch eine Programmiersprache für Webservices.

---

[70] Die Aussage wurde auf das wesentliche Vereinfacht. Außerdem wurde der Begriff „Service" hier durch „Webkomponente" ersetzt, um die Begriffsbildung an diese Arbeit anzupassen.

Geschäftsprozesse werden in BPML mit Prozessbeschreibungen ähnlich denen aus Kapitel 11.1.2 spezifiziert. Daher können auch Vorgänge in Webkomponenten mit solchen Prozessbeschreibungen spezifiziert werden. Ein Prozess kann Operationsaufrufe über mehrere Schnittstellen einer Webkomponente enthalten. Z. B. kann eine Prozessbeschreibung festlegen, dass ein Operationsaufruf bei der Webkomponente dazu führt, dass diese selbst eine Reihe von Operationen bei anderen Webkomponenten aufruft.

Die wichtigsten Konzepte, die BPML gegenüber WSCI zu einer ausführbaren Sprache vervollständigen sind Variablen, die Wertzuweisung und Ausdrücke. Variablen, die in [Arkin 02b] als Properties bezeichnet werden, sind mit dem Typsystem von XML-Schema typisiert. Mit der Aktivität `Assign` können ihnen Werte zugewiesen werden, die mit Hilfe von Ausdrücken berechnet werden können. Als Sprache für Ausdrücke wird XPath in der Version 1.0 unterstützt. Als Erweiterung sind auch andere Sprachen wie z. B. XQuery zugelassen. Für die Verwendung in Ausdrücken wird eine Bibliothek mit BPML-spezifische Funktionen zur Verfügung gestellt.

Die Aktivität `Action`, mit der Operationsaufrufe repräsentiert werden, wird von BPML um die Möglichkeit erweitert, Werte aus empfangenen Nachrichten in Variablen zu kopieren oder von Variablen in zu sendende Nachrichten. Wieder können XPath-Ausdrücke verwendet werden, um Werte aus einer Nachricht zu extrahieren oder Werte für eine zu sendende Nachricht zu berechnen.

Auch die zusätzlichen Sprachelemente für Ausnahmebehandlung, Synchronisation und das periodische Starten von Prozessen sind vor allem im Sinne einer Programmiersprache wichtig. Neben dem in WSCI vorhandenen Konzept für die Ausnahmebehandlung, wird ein zweites ergänzt, mit dem durch Abbruch von Aktivitäten ausgelöste Ausnahmen behandelt werden können. Bereits in WSCI spezifizierbare nebenläufige Prozesse können in BPML mit Signalen synchronisiert werden. Ein Prozess kann auf ein Signal warten, bis es ein anderer auslöst. Schedules ermöglichen, periodisch oder zu einem bestimmten Zeitpunkt ein Ereignis auszulösen, dass einen Prozess startet oder eine Ausnahme auslöst.

Anders als Prozessbeschreibungen von WSCI sind solche von BPML nicht in WSDL-Dokumenten enthalten, sondern in eigenen BPML-Dokumenten. WSDL-Dokumente können importiert werden, um auf Definitionen in ihnen zu verweisen. Neben Prozessen können im BPML-Dokument auch globale Variablen und Schedules definiert werden.

11.2.1.2 Vergleich mit SXQT

Durch die zugefügten Sprachelemente ist Ausdrucksfähigkeit von BPML größer als die von WSCI. Trotzdem lässt sich aber nicht alles ausdrücken, was mit SXQT ausgedrückt werden kann. Außerdem können Anforderungen nur durch Interpretation der abstrakten Implementierungen von BPML ermittelt werden, wobei eine Vielzahl von Sprachelementen verstanden werden muss. Da auch mit BPML nur abstrakte Schnittstellen beschrieben werden, lassen sich wie bei WSCL und WSCI SOAP-Fehlernachrichten und Headereinträge nicht genauer beschreiben als mit WSDL.

BPML ist vor allem eine Sprache zur Spezifikation von Geschäftsprozessen, nur indirekt können Schnittstellen von Webservices beschrieben werden, wie es das Ziel dieser Arbeit ist. Hierzu muss aus allen Anforderungen an die Schnittstelle gemeinsam ein Modell in Form einer abstrakten Implementierung der Webkomponente erstellt werden. Aus dem Verhalten der abstrakten Implementierung kann dann umgekehrt, indirekt wieder auf die Anforderungen an die Schnittstelle geschlossen werden. Wie bei WSCL und WSCI ist es so nicht möglich, Anforderungen voneinander unabhängig zu formulieren, was vor allem für Forderungen aus unterschiedlichen Quellen und Standardanfor-

derungen vorteilhaft wäre. Außerdem besteht die Gefahr, dass beim Erstellen der abstrakten Implementierung ungewollt zusätzliche Anforderungen mitspezifiziert werden. Die Gefahr besteht bei SXQT weniger, weil jede Anforderung einzeln als SXQT-Ausdruck formuliert wird.

Eine Spezifikation in Form einer abstrakten Implementierung ist auch für Werkzeuge wie den Validator schwierig zu verwenden. Der Validator müsste die Prozesse ausführen und mit den beobachten SOAP-Nachrichten vergleichen. Wie bei WSCI muss der Validator dabei alle Sprachelemente von BPML unterstützen. Probleme bereitet vor allem in der Spezifikation ausgedrückter Nichtdeterminismus. Der Validator kann nicht in jedem Fall erkennen, welche nichtdeterministische Wahl getroffen wurde. Er muss dann alle nicht ausschließbaren Möglichkeiten in Betracht ziehen. Dieses Problem wird in [Cook 99] behandelt. Bei SXQT stellt es sich nicht. Für einen beobachteten Trace müssen lediglich die SXQT-Ausdrücke ausgewertet werden.

Durch die Möglichkeit Inhalte von Nachrichten in Spezifikationen zu berücksichtigen, erhöht sich die Ausdrucksfähigkeit von BPML gegenüber WSCI. So kann spezifiziert werden, dass erlaubte Reihenfolgen von Operationsaufrufen auch von Inhalten von Nachrichten abhängig sind. Mit globalen Variablen können Reihenfolgen von Operationsaufrufen unterschiedlicher Prozesse beschrieben werden. Durch die Beschreibung, wie sich Werte in Responses ergeben, können unterschiedliche Anforderungen an einzelne SOAP-Nachrichten und einzelne Request-/Response-Paare beschrieben werden. Anforderungen an Requests lassen sich aber nicht ausdrücken. Außerdem erschwert es das Verständnis der Spezifikation, wenn Anforderungen erst wieder aus einer abstrakten Implementierung erschlossen werden müssen. In SXQT-Spezifikationen können Anforderungen dagegen direkt formuliert werden. Sie sind so besser verständlich.

Wie WSCI und WSCL beschreibt auch BPML Geschäftsprozesse nur im Sinne von abstrakten Schnittstellen. Gebundene Schnittstellen werden mit WSDL beschrieben. Das verhindert, Anforderungen an SOAP-Fehlernachrichten und Headereinträgen genauer als mit WSDL zu beschreiben, was mit SXQT möglich ist.

11.2.2 Business Process Execution Language for Web Services (BPEL4WS)

Ziel der Business Process Execution Language for Web Services (BPEL4WS) ist bei BPML die Spezifikation von Geschäftsprozessen. Mit beiden Sprachen werden Geschäftsprozesse nach den gleichen Grundprinzipien spezifiziert. Dazu werden im Wesentlichen auch aufeinander abbildbare Sprachelemente verwendet. Daher ist auch ihre Ausdrucksfähigkeit ähnlich.

BPEL4WS wurde von den Unternehmen BEA [BEA 03], IBM [IBM 03] und Microsoft [Microsoft 02a] entwickelt. Die Spezifikation [Thatte 02] wurde u.a. von IBM im Web veröffentlicht. Aus ihr wurden Informationen für diese Arbeit entnommen.

BPEL4WS ist selbst Nachfolger der Sprachen XLANG [Thatte 01] von Microsoft sowie WSFL [Leymann 01] von IBM. Die Autoren der Spezifikationen von XLANG (Satish Thatte) und von WSFL (Frank Leymann) sind auch Autoren der Spezifikation von BPEL4WS [Thatte 02]. Ebenso auch drei weitere Personen, die an der Entwicklung von WSFL mitgewirkt haben. In [Thatte 02] wird festgestellt, dass BPEL4WS XLANG und WSFL ablöst, weil es eine Annäherung dieser beiden Sprachen ist. Daher wird in dieser Arbeit nicht weiter auf sie eingegangen.

11.2.2.1 Unterschiede zwischen BPEL4WS und BPML

BPEL4WS und BPML erlauben mit unterschiedlichen, aber weitgehend vergleichbaren Sprachelementen Geschäftsprozesse zu beschreiben. Auch das Modell, was ein Geschäftsprozess ist, ist in beiden Fällen das gleiche. Es können ausführbare und abstrakte Geschäftsprozesse beschrieben werden, von denen erstere mit einem Interpreter ausgeführt werden können und letztere für die Spezifikation von Schnittstellen von Webkomponenten zweckmäßig sind. Gegenüber BPML zusätzlich erlaubt BPEL4WS die Zuweisung sogenannter nichtdeterministischer Werte, um zu spezifizieren, dass ein Wert an eine Variable zugewiesen wird, ohne diesen jedoch anzugeben. In BPEL4WS fehlt dagegen eine Möglichkeit Variablen zu definieren, die für mehrere Prozesse global sind. Als Typen für Variablen werden solche von in WSDL deklarierten Nachrichten verwendet. Außerdem können mehrere abstrakte Schnittstellen zu Service-Links zusammengefasst werden.

In Details unterscheidet sich BPEL4WS von BPML. Es gibt z. B. zum Beschreiben von Operationsaufrufen in BPEL4WS drei Sprachelemente, statt nur einem im BPML. Das Sprachelement `invoke` beschreibt einen Operationsaufruf bei einem anderen Webservice. Aufrufe von Operationen des spezifizierten Webservices werden dagegen durch die beiden Sprachelemente `receive` und `reply` ausgedrückt.

In BPEL4WS kann explizit angegeben werden, dass ein Geschäftsprozess abstrakt ist. Hierzu wird seiner Beschreibung das Attribut `abstractProcess` mit dem Wert `yes` zugefügt. Dann kann die Prozessbeschreibung nicht ausgeführt werden. Es wird aber ermöglicht, nichtdeterministisches Verhalten mit Hilfe sogenannter nichtdeterministischer Werte auszudrücken. Mit ihnen kann in der Spezifikation beschrieben werden, dass eine Wertzuweisung stattfindet und ohne den zugewiesenen Wert festzulegen. Dieses Detail kann beim der Implementierung festgelegt werden.

Variablen, die in [Thatte 02] als Container bezeichnet werden, können neben Typen von XML-Schema auch solche von in WSDL deklarierten Nachrichten haben. Das vereinfacht, empfangene oder zu sendende Nachrichten zu repräsentieren. Nicht verwechselt werden dürfen Container mit Properties von BPEL4WS, mit denen festgelegt wird, wo sich in unterschiedlichen Nachrichtentypen Werte mit gleicher Bedeutung befinden, wie z. B. eine Sozialversicherungsnummer.

In BPEL4WS gibt es anders als in BPML keine Variablen, die für mehrere Prozesse global sind. Das verhindert, eine Kommunikation zwischen unterschiedlichen Prozessen auszudrücken. Prozesse werden unabhängig voneinander definiert[71]. Variablen sind lokal für einen Prozess.

Um die Rollen von zwei miteinander kommunizierenden Webkomponenten zu definieren, die gegenseitig beieinander Operationen aufrufen, erweitert BPEL4WS WSDL um Service-Links. Ein Service-Link definiert zwei Rollen, denen jeweils eine Menge von abstrakten Schnittstellen zugeordnet wird. Zur Kommunikation über einen Service-Link, nehmen zwei Webkomponenten jeweils eine der Rollen an. Sie müssen dann die Schnittstellen bereitstellen, die zur Rolle gehören und können Operationen in Schnittstellen der anderen Rolle aufrufen. In der Spezifikationen eines Geschäftsprozesses wird angegeben, über welche Service-Links dieser kommuniziert und welche Rolle er dabei

---

[71] Hiervon ausgenommen sind geschachtelte Prozesse, deren Beschreibung sich in der eines übergeordneten Prozesses befindet. Die Aussage im Text bezieht sich auf Prozesse, die keinen übergeordneten Prozess haben.

hat. Service-Links können aber auch unabhängig von in BPEL4WS spezifizierten Geschäftsprozessen WSDL-Dokumenten zugefügt werden, um zu beschreiben über welche abstrakten Schnittstellen zwei Webkomponenten miteinander kommunizieren.

11.2.2.2 Vergleich mit SXQT

Die große Ähnlichkeit von BPEL4WS zu BPML führt zu einer ähnlichen Ausdrucksfähigkeit. Was ausgedrückt werden kann, wird nachfolgend kurz wiederholt. Danach werden wesentliche Unterschiede betrachtet, auch zu SXQT. Ein Unterschied zu BPML ist das Fehlen von Variablen, die für mehrere Prozesse global sind, wodurch sich die Ausdrucksfähigkeit reduziert. Dagegen fügen nichtdeterministische Werte Ausdrucksfähigkeit zu, die in SXQT bereits vorhanden ist. Service-Links manifestieren vor allem Vorstellungen über die Kommunikation zwischen Webkomponenten, die von denen dieser Arbeit abweichen.

Vieles, was im Vergleich zwischen BPML und SXQT in Kapitel 11.2.1.2 festgestellt wurde, gilt auch für den Vergleich zwischen BEPL4WS und SXQT. BPEL4WS kann als Erweiterung von WSDL angesehen werden. Ziel ist wieder Geschäftsprozesse zu beschreiben. Um wie mit SXQT Schnittstellen von Webkomponenten zu beschreiben, müssen zunächst alle Anforderungen gemeinsam als abstrakte Implementierung formuliert werden. Um die Anforderungen wieder aus der Spezifikation zu ermitteln, muss das Verhalten der abstrakten Implementierung untersucht werden. Die automatische Validation für BPEL4WS-Spezifikationen zu realisieren, ist aufwendiger als für SXQT-Spezifikationen.

Mit BPEL4WS kann nicht alles ausgedrückt werden, was mit SXQT ausgedrückt werden kann. Es ist möglich, Reihenfolgen von Operationsaufrufen zu beschreiben, auch abhängig von Nachrichteninhalten. Auch können z. T. Anforderungen an einzelne SOAP-Nachrichten und einzelne Request-/Response-Paare ausgedrückt werden. Wegen der Beschreibung von abstrakten Schnittstellen können aber keine SOAP-Fehlernachrichten oder Headereinträge genauer als mit WSDL beschrieben werden.

Typen von in WSDL deklarieren Nachrichten für Variablen zu verwenden, fügt BPEL4WS gegenüber BPML keine zusätzliche Ausdrucksfähigkeit zu. Es vereinfacht lediglich, Nachrichten in Geschäftsprozessen zu beschreiben. SXQT verwendet das Typsystem von XQuery, das auf XML-Schema beruht. Es ist nicht möglich, Typen von in WSDL deklarierten Nachrichten zu verwenden.

Dass in BPEL4WS anderes als bei BPML Variablen fehlen, die für mehrere Prozesse global sind, verhindert die Beschreibung von Abhängigkeiten zwischen Prozessen. Es ist daher nicht möglich, Anforderungen an Reihenfolgen von Operationsaufrufen unterschiedlicher Prozesse zu formulieren. Solche sind aber z. B. erforderlich, um den Sperrmechanismus der Webkomponente für eine Internetzeitung zu spezifizieren. Mit SXQT lassen sich solche Anforderungen beschreiben.

Mit den in BPML nicht vorhandenen nichtdeterministischen Werten kann bewusst vermieden werden zu beschreiben, welcher Wert in einer Wertzuweisung zugewiesen wird. Solchen Nichtdeterminismus zu beschreiben, ist mit SXQT trivial möglich. SXQT-Spezifikationen lassen ohnehin zunächst nichtdeterministisches Verhalten zu, das erst durch Anforderungen eingeschränkt werden muss, die als SXQT-Ausdrücke formuliert werden.

Die Einführung von Service-Links manifestiert für BPEL4WS Vorstellungen, wie die Kommunikation zwischen Webkomponenten stattfinden sollte, die von denen in dieser Arbeit abweichen. Mit Service-Links findet die Kommunikation zwischen Webkompo-

nenten durch wechselseitige Operationsaufrufe statt. Die Beziehung zwischen ihren ist also symmetrisch. Dagegen wird in SXQT davon ausgegangen, dass eine Webkomponente (der Webservice) Dienste bereitstellt, die von anderen Webkomponenten (den Webclients) genutzt werden. Schnittstellen, über die auf die Dienste zugegriffen wird, sollten so entwickelt werden, dass die Interoperabilität maximiert wird und Webclients und Webservices leicht zu implementieren sind. Das wird erschwert, wenn beide Kommunikationspartner Operationen bereitstellen müssen. Z. B. würden Firewalls, die einen Verbindungsaufbau nur in einer Richtung zulassen, dann eine Kommunikation verhindern. Daher ist es zweckmäßig Schnittstellen so zu konstruieren, dass nur ein Kommunikationspartner Operationen zur Verfügung stellt. Nur solche Schnittstellen werden von WSDL und SXQT unterstützt.

SXQT ließe sich jedoch leicht erweitern, so dass Service-Links spezifiziert werden können. Dazu könnten die Beschreibungen von Service-Links von BPEL4WS verwendet werden, die unabhängig von Geschäftsprozessbeschreibungen im WSDL-Dokument abgelegt werden. Im Modell von SXQT müsste der Beobachter alle Nachrichten in den Trace aufzeichnen, die bei einer Webkomponente über den Service-Link ausgetauscht werden. Dann könnten Anforderungen an den Nachrichtenaustausch über den Service-Link mit SXQT-Ausdrücken formuliert werden.

11.2.3 ebXML Business Process Specification Schema (BPSS)

Die Sprache ebXML Business Process Specification Schema (BPSS) dient zur Spezifikation von Geschäftsprozessen im Kontext von ebXML. Ziel der ebXML-Initiative ist es, als Nachfolger von EDI den elektronischen Austausch von in XML kodierten Geschäftsdaten zwischen Unternehmen zu ermöglichen. Dazu werden Geschäftsprozesse mit BPSS spezifiziert, mit denen festgelegt wird, in welchen Reihenfolgen zwischen den Unternehmen welche Typen von Geschäftsdokumenten ausgetauscht werden. BPSS hat damit andere Ziele als SXQT, kann aber trotzdem auch zur Beschreibung von Schnittstellen von Webkomponenten verwendet werden. Mit beiden Spezifikationsverfahren können Anforderungen formuliert werden, die sich mit dem anderen nicht ausdrücken lassen.

11.2.3.1 Electronic Business using eXtensible Markup Language (ebXML)

BPSS stammt von der ebXML-Initiative [ebXML 03], die 1999 von der UN/CEFACT [UN/CEFACT 03] und OASIS [OASIS 03] gegründet wurde. Daneben sind viele Unternehmen und mehrere Standardisierungsgremien beteiligt. Mit ebXML beschreiben sich Unternehmen in Form eines sogenannten Collaboration Protocol Profile (CPP), wobei auf unterstützte Geschäftsprozesse verwiesen wird. Die CPPs werden in einer Registrierung abgelegt. Andere Unternehmen können in der Registrierung Geschäftspartner finden und mit ihnen Verträge in Form sogenannter Collaboration Protocol Agreements (CPAs) abschließen.

In [ebXML 99] wird als Ziel der ebXML-Initiative beschrieben, ein offenes, technisches Framework zu erstellen, mit dem XML für den Austausch elektronischer Geschäftsdaten konsistent und einheitlich eingesetzt werden kann. Nach [Mertz 01] soll ebXML ein Nachfolger des Electronic Data Interchange (EDI) werden, das heute in großen Unternehmen zu diesem Zweck eingesetzt wird. [Eisenberg 01] stellt jedoch fest, dass schon mit EDI Unternehmen unterschiedlichster Größen miteinander „ad hoc“ ohne vorherige Vereinbarungen elektronischen Handel treiben können sollten. Diese Vision wurde aber nicht erreicht, weil nur für große Unternehmen die Implementierung von EDI bezahlbar ist. Mit ebXML sollen nun einerseits bisherige Investitionen in EDI bewahrt und gleichzeitig die neuen technischen Möglichkeiten von XML genutzt werden.

Als Alternative zu ebXML wird häufig das Konsortium RosettaNet [RosettaNet 03] genannt, das sich ebenfalls zum Ziel gesetzt hat Geschäftsprozesse zu beschreiben. Ein Überblick dazu wird in [RosettaNet 01] gegeben. RosettaNet ist ein Konsortium einer großen Zahl von Unternehmen. Wie bei ebXML sollen standardisierte Geschäftsprozesse erlauben, dass Unternehmen durch Austausch von XML-Dokumenten elektronischen Handel treiben können. Wie Geschäftsprozesse spezifiziert werden, wird in [RosettaNet 00] beschrieben. Es wird ein natürlichsprachliches Dokument erstellt, dem DTD-Dokumente zur Beschreibung ausgetauschter XML-Nachrichten zugefügt werden. Da solche natürlichsprachlichen Spezifikationen nicht automatisch verarbeitbar sind, spielen sie für diese Arbeit keine Rolle. Daher wird RosettaNet nicht weiter betrachtet.

In ebXML soll eine Registrierung unterstützen, dass Unternehmen Geschäftspartner finden und mit diesen elektronischen Handel treiben können. In [Eisenberg 01] wird zur Veranschaulichung ein häufig zitiertes Szenario vorgestellt, dass die Schritte beschreibt, wie zwei Unternehmen mit ebXML eine Handelsbeziehung aufbauen. Es ist in Abbildung 153 veranschaulicht. Im Szenario erstellt Unternehmen A eine ebXML-Anwendung, wozu es vorher in der Registrierung prüft, welche vorhandenen Geschäftsprozesse oder Teile von ihnen verwendet werden sollen (1). Nach der Erstellung der Anwendung registriert Unternehmen A eine Beschreibung von sich einschließlich seiner Anwendung sowie gegebenenfalls neue Geschäftsprozesse in der Registrierung (2). Dort sucht Unternehmen B nach Geschäftspartnern und findet Unternehmen A als geeignetes Unternehmen (3). Es schließt direkt mit Unternehmen A einen Vertrag ab, was automatisch geschehen kann (4). Im Vertrag werden Geschäftsprozesse und die Rollen der Unternehmen in diesen festgelegt. Danach tauschen beide Unternehmen elektronische Geschäftsdokumente gemäß der Geschäftsprozesse aus, um elektronischen Handel zu treiben (5).

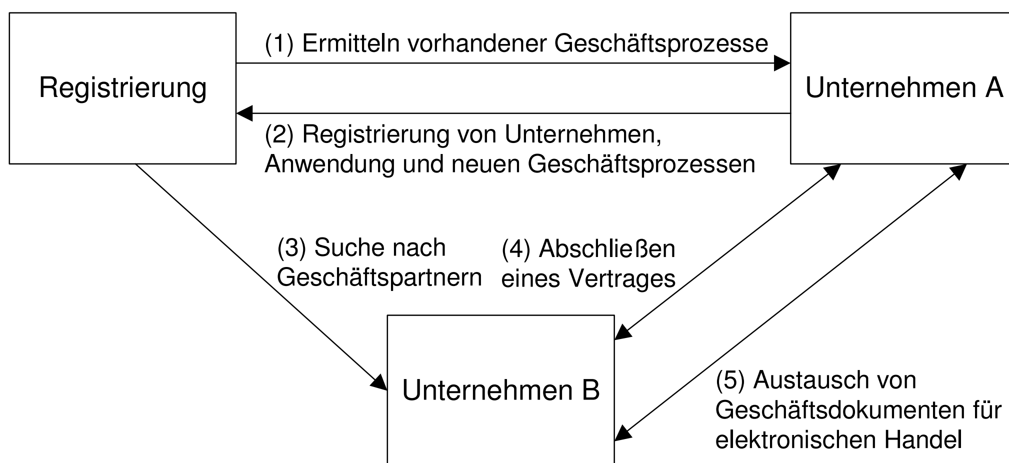

Abbildung 153: Szenario für ebXML[72]

Zur Beschreibung von Unternehmen und Verträgen zwischen diesen werden das Collaboration Protocol Profile (CPP) bzw. das Collaboration Protocol Agreement (CPA) verwendet. Das CPP beschreibt ein Unternehmen. Es legt dazu fest, an welchen Geschäftsprozessen das Unternehmen in welchen Rollen teilnehmen kann und welche Schnittstellen hierzu zur Verfügung gestellt werden. Das CPA ist ein Vertrag zwischen zwei oder mehr Geschäftspartnern, wobei festgelegt wird, welche Rolle welcher Geschäftspartner im Geschäftsprozess hat. Das CPA lässt sich automatisch aus den CPPs der Geschäftspartner ableiten.

---

[72] Die Abbildung ist ähnlich in [Eisenberg 01], Seite 8 enthalten. Sie wurde für diese Arbeit vereinfacht.

11.2.3.2 Spezifikation von Geschäftsprozessen mit BPSS

Zur Spezifikation der Geschäftsprozesse, auf die von CPPs und CPAs verwiesen wird, gibt es die Sprache Business Process Specification Schema (BPSS). Sie werden mit Hilfe von Collaborations beschrieben, die für Geschäftspartner festlegen, welche Transaktionen in welcher Reihenfolge ausgeführt werden dürfen. Transaktionen beschreiben dabei den Austausch von bis zu zwei Geschäftsdokumenten sowie zusätzlicher Geschäftssignale, mit denen der Zustand der des Geschäftsprozesses kommuniziert wird. Spezifiziert wird BPSS in [Clark 01], aus dem die nachfolgenden Informationen entnommen wurden.

Ein Geschäftsprozess wird in BPSS im Sinne von zwischen Geschäftspartnern ausgetauschten Nachrichten beschrieben. Es wird also nicht wie bei BPML und BPEL4WS eine Entität beschrieben, die den Geschäftsprozess steuert. An einem Geschäftsprozess können mehr als zwei Geschäftspartner beteiligt sein. Festgelegt wird, welche Nachrichten in welcher Reihenfolge zwischen den beteiligten Geschäftspartnern ausgetauscht werden.

Ausgetauschte Nachrichten enthalten Geschäftsdokumente in Form von XML-Dokumenten, denen Dokumente anderer Formate als Anhänge zugefügt werden können. Geschäftsdokumente werden nicht in BPSS spezifiziert. Statt dessen werden hierzu DTDs oder XML-Schemadokumente erstellt. Nachrichten zugefügte Anhänge dürfen beliebige Formate haben. Ihr Aufbau muss nicht spezifiziert sein. [Mertz 01] stellt fest, dass empfohlen wird, Nachrichten in Form von SOAP-Nachrichten zu kodieren. Zwingend ist das aber nicht.

Ebenso wie in WSDL eine Operation den Austausch von bis zu zwei SOAP-Nachrichten repräsentiert, beschreibt eine Geschäftstransaktion in BPSS den Austausch von bis zu zwei Geschäftsdokumenten. Es wird stets zunächst ein Geschäftsdokument im Request geschickt, danach kann optional mit einem zweiten im Response geantwortet werden. In der Beschreibung der Geschäftstransaktion können für den Response mehrere Typen von Geschäftsdokumenten angegeben werden, die alternativ als Response erlaubt sind. Dabei kann deklariert werden, ob eine Nachricht eines Typs einen Fehler anzeigt.

Anders als bei Operationen von WSDL können in einer Geschäftstransaktion neben den Geschäftsdokumenten noch zusätzlich sogenannte Geschäftssignale ausgetauscht werden. Das sind anwendungsunabhängig definierte Nachrichten, mit dem sich Kommunikationspartner über den Zustand der Geschäftstransaktion informieren. Z. B. wird mit dem Signal `ReceiptAcknowledgment` angezeigt, dass ein Geschäftsdokument empfangen wurde. Welche Geschäftssignale in einer Geschäftstransaktion verwendet werden, wird in deren Beschreibung deklariert.

Mit Geschäftstransaktionen sind semantische Eigenschaften verbunden, die über die Beschreibung ausgetauschter Nachrichten hinausgehen. Eine Reihe von ihnen kann in der Beschreibung der Geschäftstransaktion deklariert werden. Festgelegt ist, dass jede Geschäftstransaktion als ganzes atomar ist. Kann sie nicht erfolgreich abgeschlossen werden, muss sie von beiden Kommunikationspartnern als „null und nichtig" betrachtet werden, auch wenn bereits Nachrichten ausgetauscht wurden. Für jede Geschäftstransaktion muss eine maximale Dauer deklariert werden, nach der sie abgebrochen wird. Es kann deklariert werden, dass eine Geschäftstransaktion einen rechtsgültigen Vertrag abschließt. Außerdem kann vorgeschrieben werden, dass Mechanismen z. B. für die Nachweisbarkeit (non-repudiation), Autorisierung, Authentisierung, Vertraulichkeit oder Zuverlässigkeit der Nachrichtenübertragung verwendet werden müssen. Welche Mechanismen dazu eingesetzt werden müssen, wird aber nicht festgelegt.

Welche Geschäftstransaktionen für die Interaktion zwischen Kommunikationspartnern verwendet werden, wird mit sogenannten Collaborations beschrieben. Unterschieden werden binäre Collaborations zwischen zwei Kommunikationspartnern von Multiparty-Collaborations, an denen mehr als zwei beteiligt sind.

Binäre Collaborations haben Ähnlichkeiten mit abstrakten Schnittstellen in WSDL. Sie enthalten die Menge der Geschäftstransaktionen, über die zwei Kommunikationspartner miteinander kommunizieren. Zusätzlich können sie auch untergeordnete binäre Collaborations enthalten, die dann Teil von ihr werden und so wiederverwendet werden können. Anders als bei WSDL, ähnlich wie bei Service-Links von BPEL4WS, können jedoch Requests von beiden Kommunikationspartnern gesendet werden. In der binären Collaboration werden dazu für beide Kommunikationspartner Rollen definiert und festgelegt, welche Rolle den Request einer Geschäftstransaktion sendet.

Um Multiparty-Collaborations zwischen mehr als zwei Kommunikationspartnern zu spezifizieren, werden mehrere binäre zusammengefasst. Jede binäre Collaboration beschreibt die Kommunikation zwischen einem Paar von Kommunikationspartnern. Für die Kommunikationspartner werden Rollen definiert. Diese werden auf die Rollen der enthaltenen binären Collaborations abgebildet.

Anders als bei WSDL erlaubt es BPSS zu spezifizieren, in welchen Reihenfolgen Geschäftstransaktionen in Collaborations ausgeführt werden dürfen. Das ist Teil der Spezifikation der binären oder Multiparty-Collaboration. Ähnlich wie bei WSCL wird mit Hilfe von Transistionen definiert, dass eine Geschäftstransaktion oder untergeordnete binäre Collaboration nur ausgeführt werden darf, wenn vorher eine andere ausgeführt wurde. Es ergeben sich Graphen ähnlich dem in Abbildung 151 auf Seite 258. Für Transistionen können Bedingungen deklariert werden, die ihre Ausführung verbieten können. Außerdem kann Nebenläufigkeit und Synchronisation beschrieben werden.

Binäre und Multiparty-Collaborations sind es, auf die von CPPs und CPAs verwiesen wird, um sich auf Geschäftsprozesse zu beziehen. Im CPP wird auf Rollen in Collaborations verwiesen, um anzugeben, welche Geschäftsprozesse ein Unternehmen in welchen Rollen unterstützt. Ein CPA weist jedem am Vertrag beteiligten Unternehmen eine Rolle in einer Collaboration zu.

11.2.3.3 Vergleich mit SXQT

Anders als bei SXQT ist es das Ziel von BPSS, Geschäftsprozesse im Sinne der Kommunikation zwischen Unternehmen zu beschreiben. Die Verwendung von SOAP wird nur empfohlen, WSDL überhaupt nicht betrachtet. Aus allen Anforderungen an eine Schnittstelle muss zusammen ein gemeinsames Modell erstellt werden. Sowohl mit BPSS als auch mit SXQT lassen sich Anforderungen formulieren, die mit dem anderen nicht ausdrückbar sind.

Obwohl das Ziel von BPSS die Beschreibung von Geschäftsprozessen in Sinne der Kommunikation zwischen Unternehmen ist, lassen sich mit binären Collaborations auch Schnittstellen von Webkomponenten beschreiben. Eine binäre Collaboration beschreibt, welche Nachrichten in welcher Richtung zwischen zwei Webkomponenten ausgetauscht werden dürfen, und damit die Schnittstelle zwischen ihnen. Dabei können auch erlaubte Reihenfolgen von Nachrichten wie in SXQT festgelegt werden.

Die Vergleichbarkeit zwischen BPSS und SXQT wird dadurch erschwert, dass Geschäftstransaktionen anders modelliert sind als Operationen in SOAP. Zunächst haben Geschäftstransaktionen typischerweise eine zeitliche Ausdehnung von Tagen, sind also viel länger als Operationsaufrufe von SOAP, bei denen Request und Response über

dieselbe HTTP-Verbindung übermittelt werden. Außerdem kann wegen der Geschäftssignale eine Geschäftstransaktion mehr als zwei Nachrichten umfassen. Beides erfordert eine Abbildung auf SOAP, bei der eine Geschäftstransaktionen auf mehrere Operationsaufrufe abgebildet wird.

WSDL wird für ebXML nicht verwendet. BPSS beschreibt Geschäftstransaktionen davon unabhängig, wobei zur Beschreibung von Geschäftsdokumenten DTDs oder XML-Schemadokumente verwendet werden. Sollen die Schnittstellen von Webkomponenten mit BPSS beschrieben werden und wird für sie auch eine WSDL-Beschreibung z. B. für Werkzeuge benötigt, müssen die Schnittstellen redundant doppelt beschrieben werden. Das kann zu Inkonsistenzen zwischen beiden Beschreibungen führen und erhöht den Aufwand. SXQT vermeidet das, indem es WSDL erweitert und nur nicht bereits Beschriebenes spezifiziert.

Mit der von BPSS gewählten Vorgehensweise lassen sich viele Anforderungen nicht beschreiben, die sich mit SXQT beschreiben lassen. So können z. B. einzelne SOAP-Nachrichten nicht genauer beschrieben werden als mit DTDs bzw. XML-Schema. Anforderungen an Request-/Response-Paare und Reihenfolgen ausgetauschter Nachrichten können nicht von Inhalten der Nachrichten abhängig gemacht werden. Auch fehlen Sprachelemente, mit denen Abhängigkeiten zwischen mehreren Geschäftsprozessen beschrieben werden können, wie es z. B. für den Sperrmechanismus der Webkomponente für eine Internetzeitung erforderlich ist. Da SOAP nur empfohlen aber nicht von BPSS unterstützt wird, kann SOAP-spezifisches, wie SOAP-Fehlernachrichten und Headereinträge überhaupt nicht beschrieben werden. Nicht einmal das mit WSDL beschreibbare kann festgelegt werden, weil WSDL nicht verwendet wird.

Umgekehrt führt vor allem die Möglichkeit, semantische Eigenschaften von Geschäftstransaktionen zu deklarieren, zu einer in SXQT nicht vorhandenen Ausdrucksfähigkeit. In SXQT kann z. B. nicht ausgedrückt werden, dass durch eine ausgetauschte Nachricht ein rechtsgültiger Vertrag geschlossen wird, da das nicht durch die Inhalte der ausgetauschten Nachrichten erkennbar ist. Auch das Vorhandensein eines Mechanismusses zur Autorisierung kann nur spezifiziert werden, wenn ein bestimmter Mechanismus angenommen wird und dieser an den ausgetauschten Nachrichten erkennbar ist.

Werkzeugen, die BPSS-Spezifikationen verarbeiten, muss für jede Deklaration bekannt sein, wie mit ihr umzugehen ist. Ein Validator müsste z. B. für jede Deklaration Programmcode enthalten, der prüft, ob sie für beobachtete Nachrichten erfüllt ist. Bei SXQT reicht es dagegen, die SXQT-Ausdrücke für einen beobachteten Trace auszuwerten. Das vereinfacht die Implementierung des Validators.

Wie bei den anderen bisher in diesem Kapitel vorgestellten Spezifikationstechniken, muss auch bei BPSS zunächst aus allen Anforderungen an eine Schnittstelle ein Modell erstellt werden. Sie können nicht wie in SXQT einzeln formuliert werden. Das erschwert, Anforderungen aus unterschiedlichen Quellen oder Standardanforderungen der Spezifikation zuzufügen. Auch müssen die Anforderungen bei Bedarf wieder erst aus dem Modell erschlossen werden.

11.3 Spezifikation mit Invarianten in Implementierungssprache - XL

Die XML Programming Language (XL) ist eine Sprache zur Implementierung von Webkomponenten. Die Implementierung wird durch Verwendung des Typsystems von XML-Schema, sowie spezielle Sprachelemente für XML und Webkomponenten erleichtert. Dabei deklarierbare Invarianten enthalten logische XQuery-Ausdrücke ähnlich den SXQT-Ausdrücken, mit denen Anforderungen formuliert werden können. Da Invarianten in XL-Programmen enthalten sind, können sie aber nur zur Spezifikation von

Anforderungen an in dieser Sprache implementierten Webkomponenten verwendet werden. Dabei müssen Invarianten in der Webservicesicht gültig sein. Anders als SXQT handelt es sich nicht um eine Erweiterung von WSDL.

XL ist eine Entwicklung der Technischen Universität München [TU-München 03] zusammen mit dem Unternehmen XQRL. Die Website zu XL [Rost 03] beschreibt XL mit unterschiedlichen Veröffentlichungen. Für diese Arbeit benötigte Informationen wurden aus [Florescu 03] und [Florescu 02] entnommen.

11.3.1.1 Implementierung von Webkomponenten mit XL

Die Sprache XL befindet sich noch in der Entwicklung. XML dient stets zur Strukturierung von Daten. Als Typsystem wird XML-Schema verwendet. Um mit solchen Daten umzugehen, wird für Ausdrücke XQuery verwendet. Neben in vielen Programmiersprachen üblichen Sprachelementen (z. B. `If-Then-Else` und Wertzuweisung) gibt es solche für den Umgang mit XML-Dokumenten und zum Ausdrücken von Nichtdeterminismus, Parallelität und durch Datenabhängigkeiten gewählte Ausführungsreihenfolgen. Die Verwaltung von Konversationen wie in WSCL werden automatisch unterstützt. Weitere Sprachelemente erlauben, das Verhalten von Webkomponenten und einzelnen Operationen deklarativ zu steuern.

In [Florescu 03] wird XL als Plattform bezeichnet. Die Plattform enthält einen Übersetzer für die Sprache XL (XL Compiler) und eine Laufzeitumgebung (XL Virtual Machine). Der Übersetzer wandelt ein XL-Programm in einen sogenannten Statement-Graph, der von der Laufzeitumgebung interpretiert wird.

Das Paper [Florescu 02] stellt fest, dass die Sprache XL eine Erweiterung von XQuery ist. Dazu wird XQuery für alle Ausdrücke verwendet. Das ermöglicht die Verarbeitung von XML-Dokumenten. Da XQuery aber nur erlaubt, seiteneffektfreie Ausdrücke zu formulieren, fügt XL weitere Sprachelemente für das imperative Programmieren zu.

Daneben gibt es Sprachelemente, mit denen eine Webkomponente als ganzes beschrieben wird. Für eine Webkomponente wird ein Modul definiert, in dem globale Variablen deklariert, Funktionen definiert und weitere globale Deklarationen enthalten sind. Außerdem wird jede Operation, die die Webkomponente bereitstellt, ähnlich einer Prozedur einer höheren Programmiersprache beschrieben.

Als Typsystem wird das gleiche wie in XQuery verwendet, also XML-Schema. Jeder Wert hat einen solchen Typ. Bei der Deklaration von Variablen kann ein Typ optional angegeben werden. Strukturierte Daten werden mit komplexen Typen von XML-Schema beschrieben.

XL unterstützt lokale und globale Variablen. Anders als bei XQuery können Werte an Variablen zugewiesen oder verändert werden, wozu es unterschiedliche Sprachelemente gibt. Globale Variablen müssen deklariert werden, für lokale ist das nicht erforderlich. Die Variablen `$input` und `$output` sind implizit definiert. Sie verweisen auf den Body des Requests bzw. Responses des Operationsaufrufs, der gerade verarbeitet wird. Der zurückzusendende Response wird erstellt, indem auf die Variable `$output` ein Wert zugewiesen wird.

Neben lokalen und globalen Variablen gibt es solche, die für Konversationen global sind. Konversationen haben die gleiche Bedeutung wie bei WSCL (siehe Kapitel 11.1.1.1). Sie fassen Operationsaufrufe zusammen, die logisch zusammengehören. Jede Konversation wird mit einer URI identifiziert, die implizit in jeder SOAP-Nachricht in einem Headereintrag mitgesendet wird. So kann XL Konversationen automatisch ver-

walten. Es können Variablen deklariert werden, für die es für jede Konversation eine eigene Instanz gibt und mit denen der Zustand der Konversation verwaltet werden kann.

Für das imperative Programmieren stellt XL Sprachelemente zur Verfügung, von denen einige aus viele Programmiersprachen bekannt sind. So gibt es Sprachelemente für die Wertzuweisung, `If-Then-Else`, unterschiedliche Schleifen, die sequentielle Ausführung und die Ausnahmebehandlung. Weniger gängig sind Sprachelemente zum Beschreiben einer nebenläufigen oder nichtdeterministischen Ausführung. Anweisungen können auch so kombiniert werden, dass ihre Ausführungsreihenfolge automatisch gemäß ihrer Datenabhängigkeiten gewählt wird.

XL enthält ein Sprachelement, mit dem Operationen anderer Webservices synchron oder asynchron aufgerufen werden können. Dazu wird die URI des Webservices, der die Operation enthält, angegeben sowie ein Ausdruck, der den Body des Requests berechnet. Für synchrone Aufrufe wird zusätzlich eine Variable angegeben, in der nach dem Aufruf der Body des Responses geliefert wird. Bei asynchronen Aufrufen kann eine Operation angegeben werden, die beim Eintreffen des Responses aufgerufen wird. In der aktuellen Version von XL gibt es keine Möglichkeit Headereinträge zu beeinflussen. Das soll in späteren Versionen ergänzt werden.

Neben Sprachelementen für das imperative Programmieren gibt es auch solche, mit denen das Verhalten der Webkomponente als ganzes oder für einzelne Operationen deklarativ beeinflusst werden kann. So kann z. B. für eine Webkomponente deklariert werden, dass bestimmte Operationen aufgerufen werden sollen, wenn sich der Wert einer Variable ändert oder ein Request für eine unbekannte Operation empfangen wird. Es kann auch deklariert werden, dass alle Operationsaufrufe in der sogenannten History aufgezeichnet werden sollen. Außerdem können für die Webkomponente Invarianten angegeben werden, also logische Ausdrücke, die stets zutreffen müssen. Ähnlich dazu können für eine Operation auch Pre- und Postconditions angegeben werden, die vor bzw. nach einem Operationsaufruf erfüllt sein müssen.

11.3.1.2 Invarianten in XL

Histories haben große Ähnlichkeiten mit den Traces von SXQT, Invarianten enthalten logische XQuery-Ausdrücke ähnlich den SXQT-Ausdrücken. Auch die Operationen für SXQT-Ausdrücke aus Kapitel 7.6 können in XL implementiert werden.

Die History, in die alle Operationsaufrufe in der beschriebenen Webkomponente aufgezeichnet werden können, ist eine XML-Datenstruktur, die über die implizit deklarierte Variable `$history` erreichbar ist. [Florescu 02] stellt fest, dass für jeden Operationsaufruf unter anderem sein Request und Response aufgezeichnet wird, ebenso wie der Name der Operation und die URI des Webclients. Offensichtlich enthält damit die History ähnliche Informationen wie die in Kapitel 7.2.2 vorgestellten Traces von SXQT.

Invarianten enthalten logische XQuery-Ausdrücke, die zu jedem Zeitpunkt zutreffen müssen. Von ihnen kann eine beliebige Menge für eine Webkomponente definiert werden. Führt eine Anweisung dazu, dass der logische XQuery-Ausdruck einer Invariante den logischen Wert `false` liefert, wird die Wirkung der Anweisung rückgängig gemacht und die Verarbeitung mit einer Ausnahme abgebrochen. Die Ausnahme kann von der Webkomponente behandelt werden.

Invarianten werden typischerweise verwendet, um erlaubte Werte von globalen Variablen einzuschränken. [Florescu 02] zeigt das Beispiel für eine Invariante in Abbildung 154. Die Variable `$customer` verweist auf ein Element, dessen Kindelement age stets einen Wert enthalten muss, der größer als 18 ist. Ist das nicht der Fall wird eine Ausnahme ausgelöst.

```
invariant   $customer/age > 18
 throw <error> You are too young!
       </error>;
```

Abbildung 154: Beispiel für Invariante in XL[73]

Da die History über eine globale Variable erreichbar ist, können auch ihre erlaubten Werte mit Invarianten eingeschränkt werden und so bisherige Operationsaufrufe miteinander und mit anderen globalen Variablen in Beziehung gesetzt werden. Damit können logische XQuery-Ausdrücke, ähnlich den SXQT-Ausdrücken, in Invarianten verwendet werden.

Auch die Operationen für SXQT-Ausdrücke, die in Kapitel 7.6 vorgestellt wurden, lassen sich in XL als Funktionen implementierten. Sie können dann wie in SXQT in logischen XQuery-Ausdrücken von Invarianten verwendet werden.

11.3.1.3 Vergleich mit SXQT

Invarianten können also ähnlich den SXQT-Ausdrücken zur Spezifikation von Schnittstellen von Webservices verwendet werden. Solche logischen XQuery-Ausdrücke können aber nur in Webservicesicht (WSS) Webkomponenten zugefügt werden, die in XL implementiert sind. In WSDL spezifiziertes bleibt dabei unberücksichtigt. Alternativ zu Spezifikation mit Invarianten kann ein XL-Programm auch selbst als abstrakte Implementierung zur Spezifikation verwendet werden, wie bei BPML und BPEL4WS. Dann ergeben sich auch zu diesen ähnliche Eigenschaften.

Diese Arbeit kann als Zuarbeit für die Entwicklung der Sprache XL betrachtet werden. Die große Ähnlichkeit von Invarianten in XL zu SXQT-Ausdrücken dieser Arbeit legt nahe, letztere XL-Programmen in Invarianten zuzufügen. Das ermöglicht, dass Anforderungen an die Schnittstelle einer Webkomponente direkt in deren Implementierung enthalten sind und bei der Ausführung geprüft werden. Auch die Operationen zur Vereinfachung der SXQT-Ausdrücke aus Kapitel 7.6 können in XL implementiert und von XL-Programmen importiert werden. Mit ihnen können auch weitgehend Unterschiede zwischen der Struktur von Histories und Traces von SXQT verborgen werden. Für das Zufügen von Spezifikationen als Invarianten können die Erkenntnisse dieser Arbeit über die Spezifikation mit SXQT-Ausdrücken im Kontext von XL angewendet werden.

Spezifikationen können aber nur in XL implementierten Webkomponenten zugefügt werden. Anders als bei der in Kapitel 8 vorgeschlagenen Ablage von SXQT-Ausdrücken in WSDL, wird hier die Spezifikation nicht unabhängig von der Implementierung gespeichert. Damit sind Spezifikationen nicht für beliebige Webkomponenten allgemeingültig, sondern nur für die in XL implementierte.

In WSDL spezifiziertes bleibt anders als bei SXQT unberücksichtigt. Zwar können auch mit XL implementierte Webkomponenten mit WSDL beschrieben werden, wie in [Florescu 03] festgestellt. In WSDL spezifiziertes wird aber nicht geprüft. Vielmehr werden Operationen in XL unabhängig von WSDL beschrieben, wobei aber nicht alles ausgedrückt werden kann, was in WSDL möglich ist. Z. B. lässt sich der Aufbau von Body-, Header- oder Detaileinträgen in Fehlernachrichten nicht ausdrücken. Das muss unabhängig von XL in WSDL spezifiziert werden, wird aber dann nicht von XL geprüft. Geprüft werden nur die als Invarianten angegebenen logischen XQuery-Ausdrücke.

---

[73] Beispiel aus [Florescu 02], Seite 5.

Logische XQuery-Ausdrücke müssen in der in Kapitel 7.2.4 vorgestellten Webservicesicht (WSS) gültig sein. In der History werden alle Operationsaufrufe von allen Webclients bei der implementierten Webkomponente aufgezeichnet, aber nicht Operationsaufrufe der Webkomponente bei anderen Webservices. Diese Art zu beobachten, ist die in Kapitel 7.2.4 beschriebene Webservicesicht. Die Webclientsicht, bei der ein einzelner Webclient beobachtet wird, ist dagegen nicht möglich. Logische XQuery-Ausdrücke müssen also in der Webservicesicht gültig sein. Sind sie nur in der Webclientsicht gültig, können also nicht als Invarianten verwendet werden.

Fraglich erscheint, ob es in XL gelingt, bei einer großen History die potentiell vielen Invarianten einer Spezifikation effizient zu prüfen. Gegenüber dem Validator aus Kapitel 9 kommt hier erschwerend hinzu, dass zumindest konzeptionell für jede Anweisung des XL-Programms alle Invarianten geprüft werden. Der Validator aus Kapitel 9 prüft die Spezifikation dagegen „nur" einmal für jede SOAP-Nachricht. Außerdem wurden in Kapitel 9.7 Optimierungen vorgeschlagen. Zwar beschreibt auch [Florescu 03] unterschiedliche Optimierungen. Diese beziehen sich aber allgemein auf die Ausführung von XL-Programmen und nicht speziell auf Invarianten.

Unabhängig von Invarianten können Schnittstellen von Webkomponenten mit XL-Programmen als abstrakte Implementierung spezifiziert werden. Das entspricht der Vorgehensweise von BPML (siehe Kapitel 11.2.1) und BPEL4WS (siehe Kapitel 11.2.2). Es wird eine abstrakte Implementierung für eine Webkomponente angegeben. Aus der Implementierung lässt sich ableiten, wie sich die Webkomponente über die zu beschreibende Schnittstelle verhält. Jede Webkomponente, die die Schnittstelle nach der Spezifikation implementiert, muss sich ebenso verhalten.

Durch die gleiche Vorgehensweise ergeben sich auch Eigenschaften, die denen von BPML und BPEL4WS ähnlich sind. So muss aus allen Anforderungen an eine Schnittstelle zunächst eine abstrakte Implementierung erstellt werden. Das erschwert Anforderungen aus unterschiedlichen Quellen zusammen- oder Standardanforderungen zuzufügen. Um aus der abstrakten Implementierung wieder die Anforderungen zu ermitteln, muss sie analysiert werden. Auch die automatische Validation ist aufwendiger als bei SXQT oder der Spezifikation mit Invarianten in XL.

Abstrakte XL-Implementierungen als Spezifikationen haben auch von BPML und BPEL4WS abweichende Eigenschaften. So handelt es sich bei XL um keine Erweiterung von WSDL, so dass in WSDL ausgedrücktes redundant erneut formuliert werden muss. Dadurch, dass in XL die Verwendung von SOAP angenommen wird, können Anforderungen an SOAP-Fehlernachrichten spezifiziert werden. Fehlende Sprachelemente erlauben es jedoch trotzdem nicht, Anforderungen an Headereinträge zu spezifizieren. Schließlich erlaubt es XL durch globale Variablen, ebenso wie BPML, aber im Gegensatz zu BPEL4WS, Abhängigkeiten zwischen unterschiedlichen Konversationen zu beschreiben.

11.4 Spezifikation von Seiteneffekten in der realen Welt - DAML-S

DAML-S verwendet zur Beschreibung von Webservices Wissensrepräsentationstechniken. Es soll die Automatisierung von Aufgaben ermöglichen, die heute Menschen durchführen müssen. Im Gegensatz zu den bisher vorgestellten Spezifikationsverfahren, einschließlich SXQT, können Seiteneffekte beschrieben werden, die Webservices in der realen Welt bewirken. Wissensrepräsentationstechniken werden aber auch verwendet, um Schnittstellen zu spezifizieren. Um solche Spezifikationen zu verarbeiten, sind komplexe Algorithmen aus dem Bereich der künstlichen Intelligenz (KI) erforderlich. Die Art Spezifikationen zu formulieren sowie die gewählte Vorgehensweise für die Integration mit WSDL birgt die Gefahr von Interoperabilitätsproblemen.

11.4.1 Semantisches Web

DAML-S gehört in den Bereich des semantischen Webs (semantic web). Als dessen Vision wird in [Berners-Lee 01] beschrieben, dass das World Wide Web (WWW) zu einem verteilten Wissensrepräsentationssystem weiterentwickelt wird. Zusätzlich zu den für Menschen verständlichen Webseiten soll Wissen für die automatische Verarbeitung zugefügt werden. Das erfordert eine formale Beschreibung der Bedeutung abgelegter Informationen.

Das geschieht mit Hilfe des Ressource Description Framework (RDF) [Lassila 99]. Mit ihm wird Wissen in Form von Aussagen (statements) formuliert, die aus Subjekt, Verb und Objekt bestehen. Eine Aussage fügt einem Subjekt das Verb als Eigenschaft (property) zu. Die Eigenschaft hat das Objekt als Wert, wie in [Champin 01] beschrieben. RDF erlaubt eine Menge von Aussagen als XML-Dokument zu speichern und zu verbreiten.

Die in Aussagen verwendete Terminologie, also die Subjekte, Verben und Objekte, werden in sogenannten Ontologien definiert. Nach [Berners-Lee 01] bestehen Ontologien für das semantische Web aus einer Taxonomie und einer Menge von Inferenzregeln. Die Taxonomie definiert Klassen und Beziehungen zwischen ihnen. Eine wichtige Beziehung ist die zwischen Ober- und Unterklasse, die es erlaubt Eigenschaften an Unterklassen zu vererben. Inferenzregeln erlauben es, aus Aussagen weitere Aussagen abzuleiten.

Für die Definition von Ontologien wurden unterschiedliche Sprachen entwickelt, die selbst in RDF ausgedrückt werden. Einige werden in [SemanticWeb 02] genannt. Beim W3C befindet sich hierzu RDF-Schema [Brickley 03] in der Standardisierung. Es wird durch die Sprache Ontology Interchange Layer (OIL) [Horrocks 00] erweitert, von dem wiederum die Sprache DAML+OIL [Harmelen 01] eine Erweiterung ist. Daraus wurde die wieder beim W3C in der Standardisierung befindliche Web Ontology Language (OWL) [Harmelen 03] abgeleitet.

11.4.2 DAML-S

DAML-S (DARPA Agent Markup Language – Services) ist eine Ontologie für Webservices. Sie wird in [Martin 02] in der Version 0.7 definiert und in [McIlraith 03] kurz beschrieben. DAML-S basiert heute auf DAML+OIL. [Martin 02] stellt aber fest, dass spätere Versionen voraussichtlich auf OWL basieren werden, wenn die Standardisierung von OWL weit genug fortgeschritten ist.

Ziel von DAML-S ist es nach [McIlraith 03] Aufgaben zu automatisieren, die heute von Menschen durchgeführt werden müssen. So sollen automatisch Webkomponenten zu einem System komponiert werden können, auch wenn diese nicht für eine gemeinsame Verwendung entwickelt wurden. Außerdem soll z. B. eine automatische Überwachung der Ausführung von Systemen aus Webkomponenten (Monitoring), die Simulation von Webservices, die Verifikation von Webservice-Eigenschaften und die Unterstützung beim Finden von Fehlern (Debugging) ermöglicht werden.

Zu beachten ist jedoch, dass der Begriff Webservice im Kontext von DAML-S anders verstanden wird als in Kapitel 2 für diese Arbeit eingeführt. [Martin 02] führt ihn für DAML-S wie folgt ein: “*By ‘service’ we mean Web sites that do not merely provide static information but allow one to effect some action or change in the world, such as the sale of a product or the control of a physical device.*”[74] Der Begriff Webservice

[74] Zitat aus [Martin 02], Seite 1.

wird im Kontext von DAML-S also unabhängig von der Technologie zum Zugriff definiert. Auch solche sind eingeschlossen, auf die mit einem Webbrowser zugegriffen wird. Informationsdienste zur reinen Abfrage oder auch zur Speicherung von Informa tionen ohne Effekte in der realen Welt sind jedoch nicht mit eingeschlossen.

Die Ontologie DAML-S beschreibt Webservices mit Hilfe von drei Teilontologien: Service-Profile, Process-Model und Grounding. Das Service Profile hat die Aufgabe, die Suche nach geeigneten Webservices zu unterstützen. Hierzu wird festgelegt, wie eine Taxonomie von Webservices so gebildet wird, dass die Eigenschaften von Webservices beschrieben werden können.

Das Process-Model erlaubt es Kontroll- und Datenfluss, Vor- und Nachbedingungen, sowie Seiteneffekte in der realen Welt zu beschreiben. Sowohl das Verständnis, was ein Prozess ist, als auch die Vorgehensweise wie ein solcher definiert wird, sind denen von BPML und BPEL4WS ähnlich. Es wird der Kontroll- und Datenfluss des Programms beschrieben, das einen Webservice realisiert. Zur Beschreibung werden in DAML-S jedoch stets Wissensrepräsentationstechniken verwendet. Das gilt auch für die Beschreibung von Seiteneffekten in der realen Welt, für die eine KI-inspirierte Aktionssprache zur Verfügung gestellt wird.

Unterschieden werden drei Arten von Prozessen: atomare, zusammengesetzte und einfache. Atomare Prozesse beschreiben einen Aufruf einer Operation beim Webservice. Zusammengesetzte Prozesse sind dagegen Kompositionen von atomaren oder zusammengesetzten Prozessen. Zur Komposition stehen Kontrollkonstrukte ähnlich denen von BPML und BPEL4WS zur Verfügung, wie z. B. `If-Then-Else` oder `Sequence`. Einfache Prozesse sind Abstraktionen von atomaren oder zusammengesetzten Prozessen, die aber selbst nicht aufrufbar sind.

Das Grounding beschreibt Details, wie ein Webclient auf den beschriebenen Webservice zugreifen kann. Anders als beim Service-Profile und dem Process-Model, die den Webservice abstrakt beschreiben, werden hier konkrete Datenformate, Protokolle und Netzwerkadressen festgelegt. Für das Grounding wird WSDL verwendet. Atomare Prozesse werden in WSDL auf Operationen abgebildet, deren Ein- und Ausgaben auf Nachrichtendeklarationen. Als Typsystem wird statt XML-Schema das von DAML+OIL verwendet. Für die Abbildung auf XML-Fragmente in SOAP-Nachrichten wird eine eigene Kodierung definiert.

11.4.3 Vergleich mit SXQT

Im Vergleich zu SXQT fällt auf, dass DAML-S mit Techniken der Wissensrepräsentation sehr ambitionierte Ziele verfolgt. Solche Spezifikationen zu verwenden, erfordert jedoch schon für einfache Anwendungen Algorithmen aus dem Bereich der KI, die wesentlich komplexer sind als für SXQT benötigte. Die Verwendung von DAML+OIL als Typesystem in WSDL führt zu Interoperabilitätsproblemen.

Zu den ambitionierten Zielen gehört, dass Seiteneffekte in der realen Welt beschrieben werden können. Z. B. müsste für einen Webservice eines Lieferservices für Pizzas („Pizzaservice“) die Bedeutung spezifiziert werden, was eine Pizza ist und was es bedeutet, dass diese geliefert wird. Solche Effekte haben keinen Einfluss auf die mit dem Webservices ausgetauschten Nachrichten und können daher nicht mit SXQT beschrieben werden. Solche Bedeutungen maschinell verarbeitbar zu spezifizieren, erfordert Techniken der Wissensrepräsentation, die durchgängig in DAML-S verwendet werden.

Mit solchen Spezifikationen umzugehen, erfordert Algorithmen aus den Bereich der KI, die viel komplexer sind als die für SXQT benötigten. Der Grund ist, dass in Spezifikati-

on Terminologien verwendet werden, die erst in Ontologien definiert sind. Diese müssen von Programmen interpretiert werden. Außerdem müssen Inferenzregeln verwendet werden, um aus ihnen Schlüsse zu ziehen. Das ist aber nur mit Algorithmen aus dem Bereich der KI möglich. SXQT-Ausdrücke lassen sich dagegen mit einfacheren Algorithmen auswerten. In Kapitel 9 wurde gezeigt, dass solche SXQT-Spezifikationen besonders für die automatische Validation auf einfache Art verwendet werden können.

Nicht nur die Algorithmen für die Verarbeitung von Spezifikation von DAML-S sind komplexer. Sie sind auch für Entwickler schwerer verständlich, was zu Fehlinterpretationen führen kann und damit die Interoperabilität zwischen Webkomponenten gefährdet. Auch Entwickler müssen Ontologien einschließlich der Inferenzregeln interpretierten, um aus ihnen Schlüsse zu ziehen. Dieser Aufgabe ist fehleranfälliger als die Interpretation der SXQT-Ausdrücke dieser Arbeit.

Zu Interoperabilitätsproblemen führt auch die Verwendung von DAML+OIL als Typsystem für WSDL. Wie in Kapitel 4.1 dieser Arbeit festgestellt, ist XML-Schema das einzige Typsystem, dessen Verwendung im WSDL-Standard beschrieben ist. Für jedes andere Typsystem ist nicht sichergestellt, dass es jedem Leser eines WSDL-Dokumentes bekannt ist. Daher wurde in Kapitel 5.3.2 für diese Arbeit festgelegt, dass nur XML-Schema als Typsystem verwendet werden sollte.

11.5 Verweise auf Spezifikationen mit tModels - UDDI

Universal Description, Discovery and Integration (UDDI) ist eine Spezifikation für einen Verzeichnisdienst für Webservices. Für jeden Webservice kann angegeben werden, welche Spezifikationen er erfüllt. Hierzu kann mit tModels auf unterschiedlichste Spezifikationen verwiesen werden, die auch natürlichsprachlich formuliert sein können. Der Verweis auf ein tModel legt fest, dass ein Webservice die bezeichnete Spezifikation erfüllt. Verweise können so zur Bezeichnung beliebiger Festlegungen über einen Webservice verwendet werden, einschließlich über Seiteneffekte des Webservices in der realen Welt. Eine automatische Verarbeitung von Spezifikationen, wie die in Kapitel 9 vorgestellte automatische Validation, unterstützt diese Vorgehensweise aber nicht.

11.5.1 Verzeichnis von Webservices

Ziel von UDDI ist es, Verzeichnisdienste zu ermöglichen, in denen Unternehmen ihre Webservices veröffentlichen können, so dass Entwickler, die deren Dienste benötigen, sie finden können. Hierzu legt UDDI eine XML-Datenstruktur fest, mit der Unternehmen sich und ihre als Webservices bereitgestellten Dienste beschreiben können. Die Beschreibungen werden in UDDI-Servern abgelegt. Ziel ist es, die Suche nach Webservices zu ermöglichen, die in Softwaresysteme integriert werden sollen. UDDI legt Schnittstellen fest, wie auf UDDI-Server zugegriffen wird, um Beschreibungen zu veröffentlichen oder zu suchen. Die Schnittstellen machen UDDI-Server zu Webservices, die z. B. von Suchmaschinen verwendet werden können.

Definiert wurde UDDI durch die Organization for the Advancement of Structured Information Standards (OASIS) [OASIS 03], die auch an der Standardisierung von ebXML beteiligt ist (siehe Kapitel 11.2.2.1). Die Informationen über UDDI in dieser Arbeit sind hauptsächlich aus [UDDI 00] sowie dem XML-Schemadokument zu UDDI Version 2.0 [UDDI 01] entnommen.

In [UDDI 00] wird festgestellt, dass Informationen über Unternehmen konzeptionell in drei Teile gegliedert sind: White-Pages, Yellow-Pages und Green-Pages. White-Pages enthalten allgemeine Informationen über das Unternehmen, wie seine Adresse und Kontaktinformationen. Dagegen ordnen die Yellow-Pages das Unternehmen in Standard-

Taxonomien ein, die Ähnlichkeiten mit der Strukturierung von Branchenbüchern haben. Die Green-Pages enthalten schließlich technische Informationen, wie auf einen Webservice des Unternehmens zugegriffen werden kann. Hierzu sind Verweise auf tModels enthalten, die z. B. auf das WSDL-Dokument des Webservices verweisen.

Diese Informationen über das Unternehmen werden zusammen in einer XML-Datenstruktur abgelegt, die das Rootelement `uddi:businessEntity`[75] hat. Es enthält in Kindelementen und Attributen die Informationen der White- und Yellow-Pages. Nachfahren `uddi:businessEntity/uddi:businessServices/uddi:businessService` beschreiben außerdem die Dienste, die das Unternehmen anbietet. Jeder Dienst kann über mehrere Webservices zugänglich sein, die jeweils mit einem Bindingtemplate in einem Element `uddi:businessService/uddi:bindingTemplates/uddi:bindingTemplate` beschrieben werden. Bindingtemplates enthalten die Verweise auf tModels.

11.5.2 Beschreibung von Webservices in UDDI

Ein tModel ist eine XML-Datenstruktur, die auf eine Spezifikation verweist und über sie Metadaten enthält, wie in Abbildung 155 veranschaulicht. Auf ein tModel zu verwiesen bedeutet, dass der Webservice die Spezifikation erfüllt. Das ist einheitlich möglich, unabhängig davon, was die Spezifikation aussagt und wie sie formuliert ist. Auf die Spezifikation muss lediglich mit einer URI verwiesen werden können. Die URI ist im tModel enthalten.

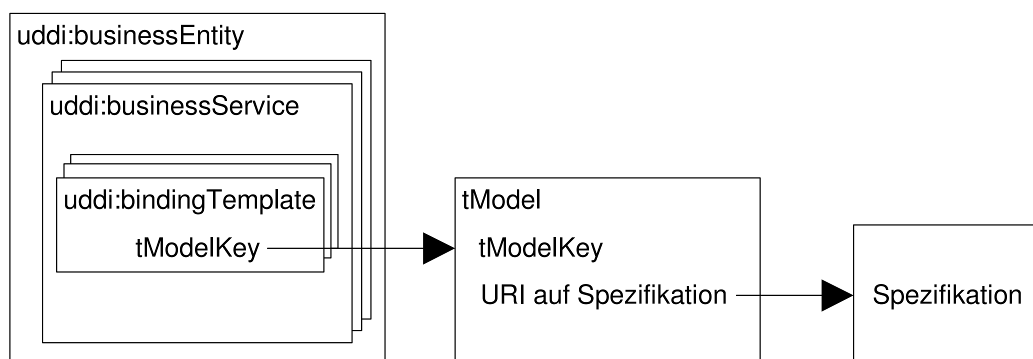

Abbildung 155: Datenstrukturen und Verweise zur Beschreibung von Webservices in UDDI

Der Verweis auf ein tModel erfolgt mit Hilfe von tModelKeys. Ein tModelKey ist eine Zeichenkette, die das tModel eindeutig identifiziert und in der Datenstruktur des tModels als Attribut enthalten ist. Das Bindingtemplate eines Webservices kann mehrere tModelKeys enthalten. Dann muss der Webservice alle Spezifikationen erfüllen, auf deren tModels mit den tModelKeys verwiesen wird. In diesem Sinne sind tModelKeys Bezeichner für Bedeutungen, die in Spezifikationen beschrieben sind.

Welche Bedeutung durch einen tModelKey bezeichnet wird und wie seine Spezifikation formuliert ist, spielt für die Beschreibung mit tModels keine Rolle. So wird z. B. für jedes WSDL-Dokument ein tModel definiert. Ein Webservice gibt mit dessen tModelKey an, dass er durch das WSDL-Dokument beschrieben wird. Das WSDL-Dokument kann dabei auch um SXQT-Ausdrücke erweitert sein, wie in Kapitel 8 beschrieben. Dann muss der Webservice auch diesem Teil der Spezifikation entsprechen. Ebenso kann mit tModelKeys auch auf Spezifikationen anderer Spezifikationsverfahren verwiesen werden, die in diesem Kapitel 11 vorgestellt wurden.

---

[75] Das Präfix `uddi` stehe für den Namensraum `urn:uddi-org:api_v2`.

Spezifikationen können in natürlicher Sprache formuliert sein. Trotzdem kann ein tModel auf sie verweisen und durch Angabe des tModelKeys eindeutig angegeben werden, dass der Webservice diese Spezifikation erfüllt. Das erlaubt z. B. auf Geschäftsprozesse zu verweisen, die von der Organisation RosettaNet [RosettaNet 03] natürlichsprachlich spezifiziert wurden.

Da Spezifikationen auch Seiteneffekte in der realen Welt beschreiben können, können solche auch mit tModelKeys bezeichnet werden. Insbesondere können solche Seiteneffekte in natürlicher Sprache beschrieben werden, aber auch formal mit Ontologien wie im Kapitel 11.4 für DAML-S beschrieben. Eine solche Spezifikation könnte z. B. festlegen, dass als Seiteneffekt des Webservices eine Pizza geliefert wird. Durch Angabe des tModelKeys kann diese Tatsache für einen Webservices eines Lieferservices für Pizza angegeben werden.

11.5.3 Vergleich mit SXQT

tModelKeys und tModels ermöglichen also den Verweis auf unterschiedlichste Bedeutungen, mit denen in UDDI Webservices einschließlich ihrer Seiteneffekte in der realen Welt beschrieben werden. Anders als bei SXQT wird eine automatische Verarbeitung wie die in Kapitel 9 beschriebene automatische Validation jedoch nicht unterstützt. UDDI verweist auf Spezifikationen anderer Spezifikationsverfahren. Auch Verweise auf SXQT-Spezifikationen sind möglich. Ein eigenes Spezifikationsverfahren wird jedoch nicht zur Verfügung gestellt. UDDI und SXQT haben also unterschiedliche, zueinander orthogonale Ziele.

12 Zusammenfassung und Ausblick

12.1 Zusammenfassung

Ziel dieser Arbeit war Wege aufzuzeigen, Interoperabilitätsprobleme zwischen Webkomponenten auszuschließen, so dass aus ihnen stabile Softwaresysteme zusammengesetzt werden können. In Kapitel 1.2 wurde vorgestellt, was dazu untersucht werden soll.

12.1.1 Interoperabilitätsprobleme in Standards

Zu prüfen war, in wieweit Standards für Webkomponenten selbst die Interoperabilität gefährden. Das wurde in Kapitel 5 behandelt. Als Problem wurden die vielen Optionen ausgemacht, die der SOAP- und WSDL-Standard zulassen. Implementieren Werkzeuge unterschiedliche Mengen von Optionen kann es zu Interoperabilitätsproblemen zwischen ihnen kommen. Optionen sind in den Standards jedoch für ihre langfristige Stabilität und die Konsensbildung notwendig.

Daher wurde in Kapitel 5 eine zweistufige Standardisierung aus breiten Standards und Präzisierungsstandards vorgeschlagen. Die Standards von SOAP und WSDL sind in diesem Sinne breite Standards. Über sie besteht Konsens. Auch bei technischem Fortschritt können sie stabil gehalten werden. Für die Interoperabilität werden sie durch Präzisierungsstandards ergänzt. Diese fügen ihnen zusätzliche Annahmen hinzu und schränken sie so weit ein, dass die Interoperabilität sichergestellt werden kann. Für Werkzeuge kann so explizit angeben werden, welchen Präzisierungsstandard sie erfüllen. Welche Annahmen ein Präzisierungsstandard für SOAP und WSDL enthalten sollte, wurde in den Kapiteln 5.2 bzw. 5.3 diskutiert.

12.1.2 Spezifikationsverfahren für Schnittstellen von Webkomponenten

Zu untersuchen war weiterhin, ob Schnittstellen mit WSDL so präzise spezifiziert werden können, dass Webkomponenten stets zueinander interoperabel sind, wenn sie die Spezifikation erfüllen. In Kapitel 6 wurden jedoch Annahmen an Schnittstellen vorgestellt, die sich in WSDL nicht ausdrücken lassen. Sie nicht zu beachten, kann zur Inkompatibilität zwischen Webclient und Webservice führen.

Daher wurde in Kapitel 7 das Spezifikationsverfahren SXQT (Specifications using XQuery expressions on Traces) entwickelt, mit dem sich im wesentlichen alle in Kapitel 6 vorgestellten Anforderungen ausdrücken lassen. Ausgegangen wurde von einem Spezifikationsverfahren für beliebige Entitäten von C.A.R. Hoare, das auf Prädikatenlogik beruht. Für die Spezifikation der Schnittstellen von Webkomponenten wurden Änderungen am zugrundeliegenden Modell vorgenommen und für die prädikatenlogischen Ausdrücke die Sprache XQuery gewählt.

Das SXQT zugrundeliegende Modell beruht auf Traces, in denen Ereignisse aufgezeichnet werden. Ereignisse sind Beobachtungen von SOAP-Nachrichten, die von einer Webkomponente gesendet oder empfangen werden. Für jedes Ereignis wird in der Reihenfolge des Auftretens von einem Beobachter die SOAP-Nachricht sowie zusätzliche, beobachtete Informationen in den sogenannten Trace aufgezeichnet. Der Trace kann als XML-Dokument angesehen werden, in dem die beobachteten SOAP-Nachrichten als XML-Fragmente enthalten sind. Absolute Zeiten werden nicht betrachtet.

Zur Spezifikation wird jede Anforderung an eine Schnittstelle als prädikatenlogischer Ausdruck (SXQT-Ausdruck) formuliert, der entscheidet, ob ein beobachteter Trace der Anforderung entspricht. Um mit XML-Fragmenten im Trace umgehen zu können, werden SXQT-Ausdrücke in der Sprache XQuery formuliert, die hierzu geeignete Operationen bereitstellt und sich beim W3C in der Standardisierung befindet.

In Kapitel 7.6 wurden weitere Operationen diskutiert, um SXQT-Ausdrücke adäquat formulieren zu können. Sie stammen aus Hoares Spezifikationsverfahren oder wurden neu vorgeschlagen. Die meisten konnten in XQuery als benutzerdefinierte Funktionen realisiert werden.

Eine SXQT-Spezifikation enthält außer einer Menge von SXQT-Ausdrücken ein WSDL-Dokument. Nur im WSDL-Dokument nicht beschriebene Anforderungen werden einzeln als SXQT-Ausdrücke formuliert. Das WSDL-Dokument wird als Teil der Spezifikation angesehen, weil es wesentlich zur Beschreibung der Schnittstelle beiträgt. Im Spezifikationsverfahren von Hoare entspricht es dem zusätzlich zur Spezifikation, implizit vorhandenen Alphabet.

Die SXQT-Ausdrücke werden im WSDL-Dokument abgelegt, um SXQT-Spezifikationen als in sich abgeschlossene Einheiten verbreiten zu können. Eine hierzu geeignete Erweiterung von WSDL wurde in Kapitel 8 vorgeschlagen. Zunächst fand die Untersuchung unabhängig von SXQT statt, so dass Ergebnisse auf andere Spezifikationsverfahren übertragbar sind. Konkret wurde vorgeschlagen, als Zeichenketten kodierte SXQT-Ausdrücke WSDL-Dokumenten in gebundenen Schnittstellen als Erweiterungselemente zuzufügen. Solche Erweiterungen sind nach dem WSDL-Standard zulässig.

In Kapitel 7.7 wurde gezeigt, dass sich mit SXQT wesentliche Anforderungen beschreiben lassen, die in WSDL nicht ausgedrückt werden können. Hierzu wurden für unterschiedliche Anforderungen SXQT-Ausdrücke angegeben, die sich nach Kapitel 6 nicht in WSDL ausdrücken lassen. Da ein WSDL-Dokument Teil jeder SXQT-Spezifikation ist, lässt sich außerdem mit SXQT alles ausdrücken, was mit WSDL ausgedrückt werden kann.

12.1.3 Automatische Validation

Zu untersuchen war auch, wie Abweichungen zwischen Webkomponenten und ihren Spezifikationen entdeckt werden können. Außerdem sollte ein Weg gefunden werden, Änderungen von Schnittstellen zu erkennen, die Anpassungen von Webclients erfordern. Beides wurde gemeinsam in Kapitel 9 untersucht. Als Lösung wurde die Idee der Programmchecker für seiteneffektfreie Funktionen zur automatischen Validation von Webkomponenten fortentwickelt.

Bei der automatischen Validation werden SOAP-Nachrichten von einem Validator beobachtet. Er realisiert den Beobachter aus dem Modell von SXQT. Zusätzlich prüft er, ob der aufgezeichnete Trace der SXQT-Spezifikation entspricht. Wurde nur ein Trace erkannt, für die das nicht der Fall ist, kann geschlossen werden, dass die Implementierung des Webclients bzw. Webservices nicht die Spezifikation erfüllt. Die automatische Validation kann beim Testen einer Webkomponente verwendet werden oder in ihrem produktiven Betrieb, um alle auftretenden Verstöße gegen die Spezifikation zu erkennen. Es kann aber nie geschlossen werden, dass eine Webkomponente in jedem Fall die Spezifikation erfüllt.

Um mit der automatischen Validation relevante Änderungen von Schnittstellen eines Webservices zu erkennen, muss die Validation auf der Seite des Webclients stattfinden.

Es wird stets gegen die Version der Spezifikation validiert, die zum Zeitpunkt der Implementierung des Webclients zur Verfügung stand. Diese Version enthält alle Annahmen über Schnittstellen des Webservices, die bei der Implementierung des Webclients bekannt waren. Wird der Webservice geändert, so dass von ihm gesendete SOAP-Nachrichten gegen diese Version der Spezifikation verstoßen, ist das eine für den Webclient relevante Änderung. Andere Änderungen sind für ihn nicht relevant.

Zwei weitere Anwendungen, der automatischen Validation ergeben sich, wenn der Validator SOAP-Nachrichten „ausfiltern“ kann, die nicht der Spezifikation entsprechen. Dann lassen sich mit ihr Webkomponenten vor böswilligen Zugriffen schützen. Außerdem können Verstöße einer Webkomponente gegen die Spezifikation vor ihren Kommunikationspartnern verborgen werden, so dass diese stets eine Webkomponente beobachten, die der Spezifikation entspricht.

Neben solchen grundsätzlichen Überlegungen wurde in Kapitel 9 konkret vorgestellt, wie ein Validator in die Architektur der Webkomponenten integriert wird und was für die automatische Validation im Detail zu prüfen ist. Zu prüfen ist für eine beobachtete SOAP-Nachricht, ob sie dem SOAP-Standard und dem Präzisierungsstandard für SOAP entspricht. Daneben muss sie dem WSDL-Dokument entsprechen, was zum Teil durch Validation gegen das darin enthaltene XML-Schemadokument geprüft werden kann. Schließlich muss der Validator den Trace aufzeichnen und für jede beobachtete SOAP-Nachricht die SXQT-Ausdrücke der SXQT-Spezifikation für den gesamten Trace auswerten. Hierzu wurden Optimierungen vorgeschlagen.

12.1.4 Verallgemeinerung auf Spezifikation von XML-Dokumenten

In Kapitel 10 wurden Ideen von SXQT auf die Spezifikation beliebiger XML-Dokumente verallgemeinert. Analog zu in WSDL beschriebenen Webservices, sind XML-Dokumente in XML-Schema beschrieben. Auch die Ausdrucksfähigkeit von XML-Schema ist begrenzt. Daher wurde ebenfalls für XML-Schema vorgeschlagen, wie es um logische XQuery-Ausdrücke erweitert werden kann, so dass dessen Ausdrucksfähigkeit deutlich verbessert wird.

12.2 Ausblick

Der nachfolgende Ausblick zeigt auf, in welchen Bereichen die Forschung an SXQT und der automatischen Validation fortgesetzt werden sollte. Interessant wären Untersuchungen, wie beides für Beispiele von Webkomponenten, Problemklassen oder komplexe Kommunikationsszenarien eingesetzt wird. Auch weitere Anwendungen für SXQT-Spezifikationen, wie die Verifikation, wären möglich.

12.2.1 Anwendung auf unterschiedliche Webkomponenten

Für Beispiele von Webkomponenten sollte gezeigt werden, wie sie mit SXQT spezifiziert werden. SXQT-Spezifikationen können entweder möglichst vollständig sein oder auch bewusst unvollständig, so dass nur einzelne Anforderungen zusätzlich zur WSDL-Beschreibung an eine Webkomponente gestellt werden. Untersuchte Webkomponenten können viel einfacher sein als die Webkomponente für eine Internetzeitung, die in dieser Arbeit als Beispiel verwendet wurde. Besonders wenn versucht wird, möglichst vollständig zu spezifizieren, ist das sinnvoll. Bei komplexen Webservices mit vielen Operationen bietet es sich dagegen an, nur einzelne Anforderungen zu stellen. Trotzdem ist dann mehr spezifiziert als nur durch das WSDL-Dokument, weil es Teil der SXQT-Spezifikation ist.

12.2.2 Anwendung auf Problemklassen

Interessant wären auch Arbeiten, die grundsätzlich untersuchen, wie sich Anforderungen bestimmter Problemklassen in SXQT ausdrücken lassen. Z. B. könnte die Beschreibung von ACID-Transaktionen im Kontext von Webkomponenten untersucht werden. Anforderungen könnten verlangen, dass ein einzelner Operationsaufruf innerhalb einer ACID-Transaktion ausgeführt wird oder auch mehrere gemeinsam. Alternativ könnte gefordert werden, dass Operationsaufrufe mehrerer Webkomponenten zu einer verteilten Transaktion gehören. Schließlich könnten Anforderungen festlegen, wie ein bereits abgeschlossener Operationsaufruf mit weiteren kompensiert wird.

Zu untersuchen wäre, wie solche Anforderungen in SXQT auszudrücken sind. Dazu muss der Schwerpunkt auf die beobachtbaren SOAP-Nachrichten gelegt werden. Findet z. B. die Koordination der Transaktionen über SOAP-Nachrichten statt, können auch diese im Trace aufgezeichnet und so im SXQT-Ausdruck berücksichtigt werden. Es sollte untersucht werden, ob neue Operationen hilfreich sind, die eine prägnantere Formulierung der SXQT-Ausdrücke erlauben. Möglicherweise muss SXQT auch für solche Problemkassen erweitert werden.

12.2.3 Erweiterung für komplexere Kommunikationsszenarien

Naheliegend wäre auch zu untersuchen, wie SXQT zum Beschreiben von Kommunikationsszenarien zwischen vielen Webkomponenten erweitert werden kann. In der Arbeit wurde stets ein Webservice betrachtet, der mit mehreren Webclients kommuniziert. Bei Beobachtung in Webservicesicht konnten so die SOAP-Nachrichten der unterschiedlichen Webclients miteinander in Beziehung gesetzt werden. Nicht möglich war es aber zu beschreiben, wie Webclients mit mehreren Webservices kommunizieren, wie in Abbildung 156 symbolisiert. Noch weniger konnten Kommunikationsszenarien wie in Abbildung 157 spezifiziert werden, bei denen Webservices auch Webclients sind, die wieder bei anderen Webservices Operationen aufrufen.

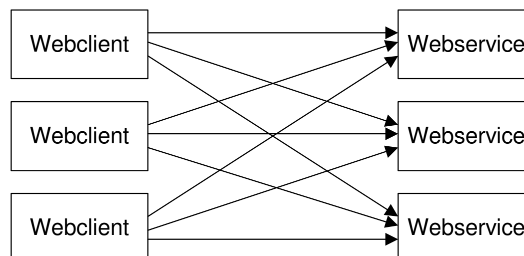

Abbildung 156: Webclients, die Operationen mehrerer Webservices aufrufen

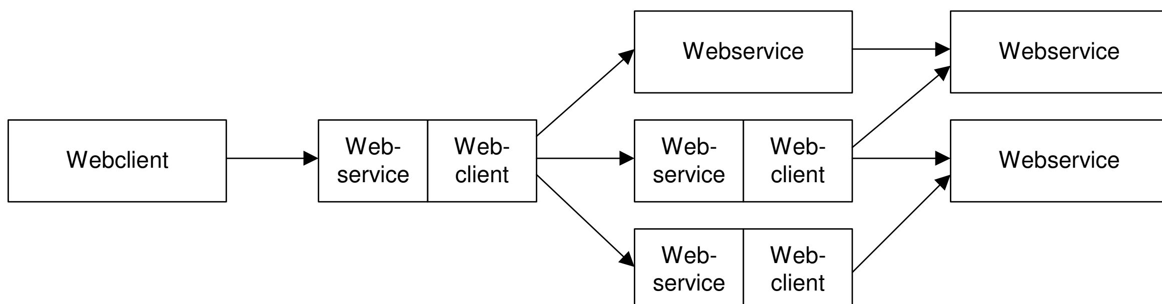

Abbildung 157: Komplexes Kommunikationsszenario

Grundsätzlich wären solche komplexeren Kommunikationsszenarien mit den Ideen von SXQT beschreibbar. Lediglich die vereinfachende Annahme, dass ein Webclient nur auf einen Webservice zugreift und vor allem die Tatsache, dass ein Beobachter stets nur eine Webkomponente beobachtet, verhindern das. Benötigt würde ein Beobachter, der die gesendeten und empfangenen SOAP-Nachrichten mehrerer Webkomponenten gleichzeitig beobachten und im Trace aufzeichnen kann.

Es müsste untersucht werden, ob dann die Struktur des Traces erweitert werden muss, so dass z. B. für jede SOAP-Nachricht ihr SOAP-Sender und SOAP-Empfänger mit notiert wird. Solche Informationen müssten in SXQT-Ausdrücken in geeigneter Form mitberücksichtigt werden, was die Definition zusätzlicher Operationen zweckmäßig machen kann.

12.2.4 Anwendung für die Verifikation

Neben solchen Erweiterungen könnten für SXQT-Spezifikationen weitere Anwendungen untersucht werden. In der Arbeit wurde ausführlich auf die automatische Validation eingegangen. Außerdem wurden SXQT-Spezifikationen als Dokumentation für Entwickler eingeführt. Interessant wäre zu untersuchen, wie solche Spezifikationen außerdem für die Verifikation verwendet werden können. Bei der Verifikation werden Eigenschaften der spezifizierten Webkomponenten mathematisch bewiesen, die zwar nicht direkt in der Spezifikation enthalten sind, sich aber aus ihr ableiten lassen.

Logische Ausdrücke einer Spezifikation sind eine gute Basis für die Verifikation. Wie gut SXQT-Ausdrücke hierzu geeignet sind, bleibt zu untersuchen. Möglicherweise müssen für die Verifikation die Anforderungen in einer anderen Logik ausgedrückt werden. Dann sollte diese generell für SXQT verwendet werden, damit die Anforderungen nicht in zwei Logiken formuliert werden müssen.

Auch wenn SXQT-Ausdrücke für die Verifikation geeignet sind, bleibt das Problem, dass ein Teil der SXQT-Spezifikation als WSDL-Dokument formuliert ist, wobei XML-Schema zur Beschreibung von XML-Fragmenten verwendet wird. Außerdem enthalten auch der SOAP-Standard und sein Präzisierungsstandard Aussagen über Schnittstellen, aus denen mit der Verifikation Schlüsse gezogen werden können. So müsste ein Beweisverfahren für die Verifikation die unterschiedlichen, verwendeten Beschreibungsverfahren unterstützen, was Beweisverfahren typischerweise nicht erlauben.

Daher erscheint es zweckmäßig, alle Anforderungen zunächst als SXQT-Ausdrücke zu formulieren. Dann muss das Beweisverfahren nur diese unterstützen. Daher sollten Regeln entwickelt werden, die WSDL-Dokumente sowie enthaltene XML-Schemadokumente auf SXQT-Ausdrücke abbilden. Außerdem sollte der SOAP-Standard sowie sein Präzisierungsstandard in Form von SXQT-Ausdrücken formuliert werden. Dann würden alle für die Verifikation relevanten Anforderungen als SXQT-Ausdrücke zur Verfügung stehen.

Auf diese Art erscheint SXQT auch für die Verifikation eine geeignete Grundlage zu bieten. Als dritte Anwendung, neben der automatischen Validation und der Verwendung als Dokumentation für Entwickler, ist das ein weiteres Argument für die Zweckmäßigkeit des in dieser Arbeit entwickelten Spezifikationsverfahrens für Schnittstellen von Webkomponenten.

A Formulierungen mit XPath-Ausdrücken

In dieser Arbeit werden zur prägnanten Beschreibung von XML-Dokumenten im Text relative XPath-Ausdrücke verwendet. Das ist zweckmäßig, weil an vielen Stellen die Struktur von XML-Dokumenten beschrieben werden muss, z. B. dass Elemente bestimmte Kindelemente oder Attribute haben. Um nicht häufig Formulierungen wie „das Attribut `type` des Elementes `wsdl:port`“ schreiben zu müssen, bieten sich relative XPath-Ausdrücke für kürzere und präzise Formulierungen an.

Alle verwendeten relativen XPath-Ausdrücke können auch als relative Pfadausdrücke von XQuery (siehe Kapitel 7.4.6) angesehen werden. Das ist möglich, weil XPath und XQuery eine sehr ähnliche Syntax und Semantik haben und in der Arbeit nur einfache Pfadausdrücke verwendet werden. Sie wurden als relative XPath-Ausdrücke von XPath 1.0 [Clark 99a] eingeführt, weil dem Leser deren Kenntnis unterstellt wurde.

In den meisten Fällen sind verwendete XPath-Ausdrücke relativ zu einem Element, dessen qualifizierter Name dem Leser aus dem Text bekannt ist. Um anzugeben, dass es sich um dieses Element handelt, beginnt der relative XPath-Ausdruck mit dessen qualifiziertem Namen. Von diesem ausgehend wird der relative XPath-Ausdruck zu den bezeichneten Kindelementen oder Attributen formuliert.

Ist z. B. das Element `xsd:schema` aus dem Text bekannt und wird vom „Attribut `xsd:schema/@targetNamespace`“ gesprochen, dann handelt es sich um das Attribut mit Namen `targetNamespace` des Elementes `xsd:schema`. Analog steht dann „die Kindelemente `xsd:schema/xsd:complexType`“ für die Kindelemente mit Namen `xsd:complexType` des Elementes `xsd:schema`.

Mit komplexeren XPath-Ausdrücken können komplexere Sachverhalte beschrieben werden. Z. B. steht bei einem bekannten Element `env:Body` der Text „das Element `env:Body/env:Fault/env:Detail`“ für das Kindelement `env:Detail` des Kindelementes `env:Fault` des Elementes `env:Body`. Ist im Kontext eine Sequenz von Elementen `wsdl:fault` bekannt, bezeichnet „das Attribut `wsdl:fault[2]/@message`“ das Attribut mit Namen `message` des zweiten Elementes `wsdl:fault` der Sequenz.

In einigen Fällen werden XPath-Ausdrücke auch relativ zu Elementen verwendet, die dem Leser nicht bekannt sind. Solche Ausdrücke beschreiben, welches Elternelement ein Element oder Attribut haben muss. Ist z. B. das Element `xsd:schema` nicht bekannt, dann steht „das Attribut `xsd:schema/@targetNamespace`“ für ein Attribut mit Namen `targetNamespace`, das zu einem Element `xsd:schema` gehört. Analog bezeichnet dann „die Elemente `xsd:schema/xsd:complexType`“ alle Elemente `xsd:complexType`, die ein Elternelement `xsd:schema` haben. Diese Art relative XPath-Ausdrücke zu verwenden, wird auch in der Sprache XSLT [Clark 99b] verwendet, um für sogenannte Templates festzulegen, für welche Art von Knoten sie verwendet werden können.

B Verwendete Präfixe

Präfix	URI des Namensraumes	Gehört zu
env	http://www.w3.org/2002/06/soap-envelope	SOAP
enc	http://www.w3.org/2002/06/soap-encoding	SOAP
flt	http://www.w3.org/2002/06/soap-faults	SOAP
fn	http://www.w3.org/2002/11/xquery-functions	XQuery
hfp	http://www.w3.org/2001/XMLSchema-hasFacetAndProperty	XML-Schema
op	http://www.w3.org/2002/11/xquery-operators	XQuery
opr	http://ti5.tu-harburg.de/venzke/20021015/operations	SXQT
my	Siehe unten.	
nwfrw	http://example.org/2002/Security	Webkomponente für eine Internetzeitung
nwse	http://example.org/NewsService020917/Errors	Webkomponente für eine Internetzeitung
nwsext	http://example.org/NewsService020917/Extension	Webkomponente für eine Internetzeitung
nwsi	http://example.org/NewsService020917/Interface	Webkomponente für eine Internetzeitung
nwst	http://example.org/NewsService020917/Types	Webkomponente für eine Internetzeitung
nwstr	http://example.org/2002/TIP-Transactions	Webkomponente für eine Internetzeitung
q	http://www.w3.org/2001/06/xqueryx	XQuery
rpc	http://www.w3.org/2002/06/soap-rpc	SOAP
soap12	http://www.w3.org/2002/07/wsdl/soap12	WSDL
tra	http://ti5.tu-harburg.de/venzke/20021015/traces	SXQT
uddi	urn:uddi-org:api_v2	UDDI
wex	http://ti5.tu-harburg.de/venzke/20021015/wsdl-extension	SXQT
wscl	http://www.w3.org/2002/02/wscl10	WSCL
wsci	http://www.w3.org/2002/07/wsci10	WSCI
wsdl	http://www.w3.org/2002/07/wsdl	WSDL
xsd	http://www.w3.org/2001/XMLSchema	XML-Schema
xse	http://ti5.tu-harburg.de/venzke/20021015/xmlschema-extension	SXQT
xsi	http://www.w3.org/2001/XMLSchema-instance	XML-Schema

Tabelle 7: In der Arbeit verwendete Präfixe für Namensräume in XML

Der Präfix `my` wird in dieser Arbeit für kleinere Beispiele verwendet. Er wird hierzu an die URIs unterschiedlicher Namensräume gebunden, die für das Verständnis der Beispiele jedoch keine Rolle spielen.

C Tabellenverzeichnis

Tabelle 1: Im SOAP-Standard festgelegte Fehlercodes 34
Tabelle 2: Anforderungen für einen Präzisierungsstandard für SOAP 95
Tabelle 3: Anforderungen für einen Präzisierungsstandard für WSDL 96
Tabelle 4: Operationen für Traces von Hoare 169
Tabelle 5: Klassen von Anforderungen an Schnittstellen von Webkomponenten 178
Tabelle 6: Mögliche Fehlersituationen mit Programmchecker 220
Tabelle 7: In der Arbeit verwendete Präfixe für Namensräume in XML 291

D Abbildungsverzeichnis

Abbildung 1: Komponente mit Klassen und Schnittstellen ... 8
Abbildung 2: Zwei Webclients, die auf einen Webservice zugreifen ... 10
Abbildung 3: Über eine Brücke verbundene Komponenten ... 12
Abbildung 4: Abbildung eines PIMs in PSMs und in Implementierungen ... 15
Abbildung 5: Aufrufe einer Operation mit Request-/Response-Protokoll ... 18
Abbildung 6: Möglicher Aufbau einer Webseite der Internetzeitung ... 24
Abbildung 7: Direkter Nachrichtenaustausch zwischen SOAP-Sender und SOAP-Empfänger ... 30
Abbildung 8: SOAP-Message-Path mit zwei SOAP-Intermediaries ... 30
Abbildung 9: Serialisierung einer SOAP-Nachricht mit zwei Headereinträgen ... 31
Abbildung 10: Beispiel für eine SOAP-Fehlernachricht ... 34
Abbildung 11: Einfacher Wert in SOAP-Datenmodell ... 38
Abbildung 12: Einfacher Wert , kodiert als XML-Fragment ... 38
Abbildung 13: Struktur in SOAP-Datenmodell ... 38
Abbildung 14: Struktur, kodiert als XML-Fragment ... 38
Abbildung 15: Einfacher, typisierter Wert, kodiert als XML-Fragment ... 39
Abbildung 16: SOAP-Datenmodell mit mehrfach referenziertem Knoten und Zyklus ... 39
Abbildung 17: Komplexer Graph, kodiert als XML-Fragment ... 40
Abbildung 18: SOAP-Nachricht mit durch SOAP-Encoding gebildetem Bodyeintrag ... 40
Abbildung 19: Request-/Response-Kommunikationsmuster ... 41
Abbildung 20: SOAP-Response-Kommunikationsmuster ... 42
Abbildung 21: SOAP-Response, gebildet nach der RPC-Repräsentation ... 44
Abbildung 22: HTTP-Request mit SOAP-Nachricht für Webmethode `POST` ... 47
Abbildung 23: HTTP-Request für Webmethode `GET` ... 47
Abbildung 24: HTTP-Response mit SOAP-Nachricht ... 48
Abbildung 25: Verwendbarkeit einfacher bzw. komplexer Typen für Attribute bzw. Elemente ... 52
Abbildung 26: Beispiel für Verweis von Instanzdokument auf Schemadokument ... 52
Abbildung 27: Ausschnitt der Hierarchie vordefinierter Typen von XML-Schema ... 54
Abbildung 28: Beispiel für Restriction zur Festlegung der Zeichenkettenlänge in XML-Schema ... 56
Abbildung 29: Beispiel für Restriction zur Definition eines Aufzählungstyps in XML-Schema ... 56
Abbildung 30: Beispiel für Restriction mit einem regulären Ausdruck in XML-Schema ... 57
Abbildung 31: Beispiel für `xsd:sequence` in XML-Schema ... 58
Abbildung 32: Beispiel für Instanzdokument zum Schemadokument in Abbildung 31 ... 58
Abbildung 33: Beispiel für `xsd:choice` in XML-Schema ... 58
Abbildung 34: Beispiel für Instanzdokument zum Schemadokument in Abbildung 33 ... 59
Abbildung 35: Beispiel für Referenz auf globale Deklaration eines Elementes in XML-Schema ... 59
Abbildung 36: Beispiel für Attribut `minOccurs` und `maxOccurs` in XML-Schema ... 60
Abbildung 37: Beispiel für Instanzdokument zum Schemadokument in Abbildung 36 ... 60
Abbildung 38: Beispiel für komplexen Typ mit Attributen in XML-Schema ... 61
Abbildung 39: Beispiel für Instanzdokument zum Schemadokument in Abbildung 38 ... 61
Abbildung 40: Beispiel für komplexen, aus einem einfachen abgeleiteten Typ in XML-Schema ... 62
Abbildung 41: Beispiel für Instanzdokument zum Schemadokument in Abbildung 40 ... 62
Abbildung 42: Struktur von WSDL-Beschreibungen ... 64
Abbildung 43: Grundsätzlicher Aufbau von WSDL-Dokumenten ... 65
Abbildung 44: Beispiel für in WSDL-Dokument eingebettetes XML-Schemadokument ... 66
Abbildung 45: Zwei Beispiele für Deklarationen von Nachrichten in WSDL ... 67

Abbildung 46: Ausschnitt von Definition einer abstrakten Schnittstelle in WSDL 69
Abbildung 47: Ausschnitt von Definition einer gebundenen Schnittstelle in WSDL 71
Abbildung 48: Beispiel für SOAP-Nachricht beschrieben mit `style="document"` 73
Abbildung 49: Beispiel für SOAP-Nachricht beschrieben mit `style="rpc"` 73
Abbildung 50: Beispiel für Deklaration eines Dienstes und Definition eines Diensttyps in WSDL 76
Abbildung 51: Beispiel für Response der Operation `Categories` 98
Abbildung 52: Beispiel für Response der Operation `List` mit Headereintrag 99
Abbildung 53: Beispiel für Request der Operation `Get` 100
Abbildung 54: Beispiel für Response der Operation `Get` 100
Abbildung 55: Beispiel für Response der Operation `Login` 102
Abbildung 56: Beispiel für Request der Operation `Get`, der Sperre setzt 103
Abbildung 57: Beispiel für Request der Operation `Login` mit Passwort 104
Abbildung 58: Deklaration des Bodyeintrages der Operation `Login` 104
Abbildung 59: XML-Fragment eines mehrfach referenzierten Knotens 105
Abbildung 60: Deklaration des Elementes für mehrfach referenzierten Knoten 106
Abbildung 61: Nach SOAP-Encoding ungültiges XML-Fragment 106
Abbildung 62: Beispiel für Request der Operation `ChangeCategory` 107
Abbildung 63: Deklaration des Elementes `nwst:Category` mit `xsd:enumeration` 107
Abbildung 64: Beispiel für einen Response der Operation `Categories` 108
Abbildung 65: Beispiel für eine SOAP-Fehlernachricht 109
Abbildung 66: Definition der Operation `Get` in abstrakter Schnittstelle 110
Abbildung 67: Definition der Operation `Get` in gebundener Schnittstelle mit Headereintrag 113
Abbildung 68: Beispiel für Request der Operation `Get` mit Headereintrag `nwst:TipURL` 114
Abbildung 69: Deklaration des Headereintrages `nwst:TipURL` 114
Abbildung 70: Beispiel für XML-Fragment, das die Beobachtung eines Ereignisses beschreibt 126
Abbildung 71: Beispiel für Trace, der vier Beobachtungen von Ereignissen enthält 128
Abbildung 72: XML-Schemadokument des Namensraumes für Traces 129
Abbildung 73: Beobachter mit verschiedenen Sichten 130
Abbildung 74: XML-Dokument, dargestellt mit Sequenzen 140
Abbildung 75: Beispiel für bedingten Ausdruck in XQuery 146
Abbildung 76: Beispiel für FLWOR-Ausdruck mit `let` in XQuery 146
Abbildung 77: Beispiel für FLWOR-Ausdruck mit `for` in XQuery 147
Abbildung 78: Beispiel für FLWOR-Ausdruck mit `for` und `where` in XQuery 147
Abbildung 79: Beispiel für Ausdruck mit Quantor `every` in XQuery 148
Abbildung 80: Beispiel für Ausdruck mit Quantor `some` in XQuery 148
Abbildung 81: Beispiel für Ausdruck mit Query-Prolog in XQuery 148
Abbildung 82: Beispiel für Ausdruck mit benutzerdefinierter Funktion in XQuery 149
Abbildung 83: XML-Dokument zur Veranschaulichung von Sequenzen 152
Abbildung 84: Anforderung an einzelne SOAP-Nachrichten als SXQT-Ausdruck 153
Abbildung 85: Anforderung an einzelne SOAP-Nachrichten als SXQT-Ausdruck, verkürzt notiert 154
Abbildung 86: Anforderung an einzelne Request-/Response-Paare als SXQT-Ausdruck 155
Abbildung 87: Anforderung an einzelne Request-/Response-Paare als SXQT-Ausdruck, anders formuliert 157
Abbildung 88: Operationen zum Extrahieren von Teilen aus Message-Element 159
Abbildung 89: Realisierungen der Funktionen `opr:event-body-entry` und `opr:event-name` 160
Abbildung 90: Realisierung der Funktion `opr:same-event-class` 161
Abbildung 91: Realisierungen der Funktionen `opr:head`, `opr:tail` und `opr:reverse` 163
Abbildung 92: Realisierungen der Funktionen `opr:prefix`, `opr:subsequence` und `opr:interleaves`. 164
Abbildung 93: Realisierungen der Funktionen `opr:restrict` und `opr:count-restricted` 166
Abbildung 94: Muster zur Konstruktion von Funktionen f* 168
Abbildung 95: Realisierung der Funktion `opr:tr` 171
Abbildung 96: Realisierungen von `opr:associated-request` und `opr:associated-response` 172
Abbildung 97: Realisierungen der Funktionen `opr:requests` und `opr:responses` 172

Abbildung 98: Realisierung der Funktion `opr:tr-save` 173
Abbildung 99: Beispiel für Response der Operation `UnlockAll` 174
Abbildung 100: Beispiel für spezifikationsspezifische Filteroperationen....174
Abbildung 101: Anforderung an einzelne SOAP-Nachrichten, notiert mit Operationen....175
Abbildung 102: Anforderung an einzelne Request-/Response-Paare, notiert mit Operationen 176
Abbildung 103: Anforderung an optionalen Parameter als SXQT-Ausdruck....181
Abbildung 104: Anforderung an alternative Struktur als SXQT-Ausdruck....181
Abbildung 105: Anforderung an Zuordnung von Subfehlercode zu Fehlercode als SXQT-Ausdruck....182
Abbildung 106: Anforderung an Bekanntheit von Headereinträgen als SXQT-Ausdruck 183
Abbildung 107: Anforderung an Beziehung zwischen Header- und Bodyeintrag als SXQT-Ausdruck....184
Abbildung 108: Beispiel für SOAP-Fehlernachricht mit Subfehlercode `nwse:AccessDenied`....186
Abbildung 109: Anforderung an Subfehlercode als SXQT-Ausdruck 186
Abbildung 110: Anforderung an gelieferte SessionID als SXQT-Ausdruck 188
Abbildung 111: Anforderung an Verarbeitungsstrategie als SXQT-Ausdruck....189
Abbildung 112: Beispiel für Request der Operation `Unlock` 190
Abbildung 113: Anforderung an notwendige Sperre als SXQT-Ausdruck....191
Abbildung 114: Benutzerdefinierte Hilfsfunktion `my:sessionID`....191
Abbildung 115: Benutzerdefinierte Hilfsfunktion `my:lock-possibly-set` 192
Abbildung 116: Benutzerdefinierte Hilfsfunktion `my:unlock-possibly-not-available` 193
Abbildung 117: Benutzerdefinierte Hilfsfunktion `my:response-possibly-not-error` 194
Abbildung 118: Benutzerdefinierte Hilfsfunktion `my:response-not-error` 194
Abbildung 119: Anforderung der Klasse CD behandelt wie in Klasse CC als SXQT-Ausdruck 196
Abbildung 120: Anforderung der Klasse CD als SXQT-Ausdruck mit Funktion `fn:document` 197
Abbildung 121: Definition des Typs `xsd:boolean` mit Element `xsd:appinfo` aus [Brion 01] 202
Abbildung 122: Hierarchie der Elemente in WSDL-Dokumenten 204
Abbildung 123: Beispiel für SXQT-Ausdruck in XQuery-Syntax 211
Abbildung 124: Beispiel für SXQT-Ausdruck in XQueryX-Syntax 211
Abbildung 125: Struktur der Ereiterungselemente für WSDL, eingebettet in gebundene Schnittstelle....213
Abbildung 126: XML-Schemadokument der Erweiterungselemente für WSDL 214
Abbildung 127: Ausschnitt aus WSDL-Dokument mit Erweiterungselementen....215
Abbildung 128: Funktionsweise eines Programmcheckers 217
Abbildung 129: Validatoren für Webkomponenten....223
Abbildung 130: Validator mit Filter....227
Abbildung 131: Zuordnung des Validators zu Webclient bzw. Webservice 229
Abbildung 132: HTTP-Proxy zwischen HTTP-Client und HTTP-Server 230
Abbildung 133: Validator als HTTP-Proxy, dem Webservice zugeordnet....230
Abbildung 134: Validator in SOAP-Implementierung, dem Webservice zugeordnet 231
Abbildung 135: Manuell gestarteter Validator mit Beobachter in Implementierung des Webservices 233
Abbildung 136: Validator mit Warteschlange und Beobachter in Implementierung des Webservices....233
Abbildung 137: Definition der Operation `Get` in abstrakter Schnittstelle....239
Abbildung 138: Definition der Operation `Get` in gebundener Schnittstelle mit Headereintrag 241
Abbildung 139: Zwei durch Konjunktion zusammengefasste SXQT-Ausdrücke 244
Abbildung 140: Zwei durch Konjunktion zusammengefasste SXQT-Ausdrücke, optimiert....245
Abbildung 141: SXQT-Ausdruck, der nur letzte SOAP-Nachricht prüft 245
Abbildung 142: Anforderung an alternative Struktur, verallgemeinert als XQuery-Ausdruck....248
Abbildung 143: Anforderung an Rootelement als XQuery-Ausdruck....249
Abbildung 144: Deklaration der Funktion `opr:context`....250
Abbildung 145: Anforderung an alternative Struktur, als relativer XQuery-Ausdruck 250
Abbildung 146: Beispiel für XML-Fragment mit Gleichheit von Attribut und Anzahl Kindelemente....250
Abbildung 147: Anforderung an Gleichheit von Attribut und Anzahl Kindelemente als XQuery-Ausdruck....250
Abbildung 148: XML-Schemadokument des Erweiterungselementes für XML-Schema 253
Abbildung 149: Struktur des Elementes `xsd:annotation` mit Erweiterungselement 253

Abbildung 150: Beispiel für XML-Schemadokument mit Erweiterungselementen 254
Abbildung 151: Beispiel für gerichteten Graph einer Konversation von WSCL 258
Abbildung 152: Beispiel für Beschreibung eines WSCI-Prozesses, vereinfacht aus [Arkin 02a] 261
Abbildung 153: Szenario für ebXML 270
Abbildung 154: Beispiel für Invariante in XL 276
Abbildung 155: Datenstrukturen und Verweise zur Beschreibung von Webservices in UDDI 281
Abbildung 156: Webclients, die Operationen mehrerer Webservices aufrufen 286
Abbildung 157: Komplexes Kommunikationsszenario 286

E Literaturverzeichnis

[AltaVista 03] ALTAVISTA: *AltaVista's Babel Fish Translation Service*. Website. AltaVista Company, Palo Alto, 2003. http://babelfish.altavista.com/ (6.5.03)

[Apache 01] APACHE: *The Apache XML Projekt*. Website. The Apache Software Foundation, Forest Hill, 2001. http://xml.apache.org/ (25.10.02)

[Apparao 02] APPARAO, Vidur; u. a.: *XML Protocol (XMLP) Requirements*. W3C-Working-Draft. World Wide Web Consortium (W3C), Juni 2002. http://www.w3.org/TR/2002/WD-xmlp-reqs-20020626 (12.7.02)

[Arkin 02a] ARKIN, Assaf; u. a.: *Web Service Choreography Interface (WSCI) 1.0*. W3C-Note. World Wide Web Consortium (W3C), August 2002. http://www.w3.org/TR/2002/NOTE-wsci-20020808/ (17.1.03)

[Arkin 02b] ARKIN, Assaf: *Business Process Modelling Language*. Working-Draft. The Business Process Management Initiative (BPMI.org), November 2002. http://www.bpmi.org/bpml-spec.esp (13.2.03)

[Austin 02] AUSTIN, Daniel; u. a.: *Web Services Architecture Requirements*. W3C-Working-Draft. World Wide Web Consortium (W3C), April 2002. http://www.w3.org/TR/2002/WD-wsa-reqs-20020429 (11.7.02)

[Baker 02] BAKER, Mark; NOTTINGHAM, Mark: *The "application/soap+xml" media type*. Working-Draft. World Wide Web Consortium (W3C), Juni 2002. http://www.w3.org/2000/xp/Group/2/06/18/draft-baker-soap-media-reg-01.txt (6.5.03)

[Banerji 02] BANERJI, Arindam; u. a.: *Web Services Conversation Language (WSCL) 1.0*. W3C-Note. World Wide Web Consortium (W3C), März 2002. http://www.w3.org/TR/2002/NOTE-wscl10-20020314/ (18.1.03)

[Bartsch 01] BARTSCH, Roy; GOERIGK, Wolfgang: "*Mechanical a-posteriori Verification of Results: A Case Study for a Safety Critical AI System*". In: KHATIB, Lina; PECHEUR, Charles: *Model-Based Validation of Intelligence - Papers from 2001 AAAI Spring Symposium*. AAAI Press, Menlo Park, März 2001. http://www.informatik.uni-kiel.de/~wg/Berichte/AAAI-2001.ps.gz (16.12.03)

[BEA 03] BEA: *Bea Systems*. Website. Bea Systems, San Jose, 2003. http://www.bea.com/ (6.5.03)

[Berglund 02] BERGLUND, Anders; u. a.: *XML Path Language (XPath) 2.0*. W3C-Working-Draft. World Wide Web Consortium (W3C), August 2002. http://www.w3.org/TR/2002/WD-xpath20-20020816 (25.10.02)

[Berners-Lee 98] BERNERS-LEE, Tim; u. a.: *Uniform Resource Identifiers (URI): Generic Syntax*. RFC 2396. Internet Engineering Task Force (IETF), August 1998. http://www.ietf.org/rfc/rfc2396.txt (5.8.02)

[Berners-Lee 01] BERNERS-LEE, Tim; u. a.: "*The Semantic Web*". In: Scientific American, Vol. 284, Nr. 5, S. 34–43, Mai 2001. http://www.sciam.com/2001/0501issue/0501berners-lee.html (3.3.03)

[Blum 89] BLUM, Manuel and KANNAN, Sampath: "*Designing programs that check their work*". In: ACM: *Proceedings of the twenty first annual ACM symposium on theory of computing*. ACM Press, New York, S. 86-97, Mai 1989.

[Blum 93] BLUM, Manuel; u. a.: "*Self-Testing/Correcting with Applications to Numerical Problems*". In: Journal of Computer and System Sciences, Vol. 47, Nr. 3, S. 549-595, Dezember 1993. http://www.cs.cornell.edu/Info/People/ronitt/PAP/blr.ps (13.12.02)

[Blum 95] BLUM, Manuel and KANNAN, Sampath: "*Designing programs that check their work*". In: Journal of the ACM, Vol. 42, Nr. 1, S. 269-291, Januar 1995. ftp://ftp.cis.upenn.edu/pub/kannan/jacm.ps.gz (12.12.02)

[Boag 02] BOAG, Scott; u. a.: *XQuery 1.0: An XML Query Language*. W3C-Working-Draft. World Wide Web Consortium (W3C), November 2002. http://www.w3.org/TR/2002/WD-xquery-20021115/ (6.5.03)

[Box 00a] BOX, Don; u. a.: *Essential XML - Beyond Markup*. Addison Wesley, Boston, 2000.

[Box 02] BOX, Don: *Understanding GXA*. Microsoft, Redmond, Juli 2002. http://msdn.microsoft.com/library/en-us/dngxa/html/understandgxa.asp (030310)

[BPMI 03] BPMI.ORG: *BPMI.org, The Business Process Management Initiative*, Website. The Business Process Management Initiative (BPMI.org), Aurora, 2003. http://www.bpmi.org/ (6.5.03)

[Bray 99] BRAY, Tim; u. a.: *Namespaces in XML*. W3C-Recommendation. World Wide Web Consortium (W3C), Januar 1999. http://www.w3.org/TR/1999/REC-xml-names-19990114/ (2.4.03)

[Bray 00] BRAY, Tim; u. a.: *Extensible Markup Language (XML) 1.0 (Second Edition)*. W3C-Recommendation. World Wide Web Consortium (W3C), Oktober 2000. http://www.w3.org/TR/2000/REC-xml-20001006 (18.9.02)

[Brickley 03] BRICKLEY, Dan; GUHA, R.V.: *RDF Vocabulary Description Language 1.0: RDF Schema*. W3C-Working-Draft. World Wide Web Consortium (W3C), Januar 2003. http://www.w3.org/TR/2003/WD-rdf-schema-20030123/ (7.3.03)

[Brion 01] BRION, Paul V.; MALHOTRA, Ashok: *XML Schema Part 2: Datatypes*. W3C-Recommendation. Mai 2001. http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/ (18.9.02)

[Busby 01] BUSBY, Steve; JEZIERKSI, Edward: *Microsoft .NET/COM Migration and Interoperability*. Microsoft, Redmond, August 2001. http://msdn.microsoft.com/library/en-us/dnbda/html/cominterop.asp (24.7.02)

[Champin 01] CHAMPIN, Piere-Antoine: *RDF-Tutorial*. Laboratoire d'Ingénierie des Systèmes d'Information, Université Claude Bernard Lyon 1, April 2001. http://www710.univ-lyon1.fr/~champin/rdf-tutorial/rdf-tutorial.pdf (26.2.03)

[Champion 02] CHAMPION, Michael; HOLLANDER, Dave: *Web Services Architecture Working Group Charter*. World Wide Web Consortium (W3C), Juli 2002. http://www.w3.org/2002/01/ws-arch-charter (2.8.02)

[Chinnici 02] CHINNICI, Roberto, u. a.: *Web Services Description Language (WSDL) Version 1.2*. Working-Draft. World Wide Web Consortium (W3C), Juli 2002. http://www.w3.org/TR/2002/WD-wsdl12-20020709/ (2.9.02)

[Christensen 01] CHRISTENSEN, Erik, u. a.: *Web Services Description Language (WSDL) 1.1*. W3C-Note. World Wide Web Consortium (W3C), März 2001. http://www.w3.org/TR/2001/NOTE-wsdl-20010315 (18.9.02)

[Clark 99a] CLARK, James; DEROSE, Steve: *XML Path Language (XPath) - Version 1.0*. W3C-Recommendation. World Wide Web Consortium (W3C), November 1999. http://www.w3.org/TR/1999/REC-xpath-19991116 (6.5.03)

[Clark 99b] CLARK, James: *XSL Transformations (XSLT) - Version 1.0*. W3C-Recommendation. World Wide Web Consortium (W3C), November 1999. http://www.w3.org/TR/1999/REC-xslt-19991116 (18.9.02)

[Clark 01] CLARK, Jim; u. a.: *ebXML Business Process Specification Schema Version 1.01*. Technical Specification. UN/CEFACT und OASIS, 2001. http://www.ebxml.org/specs/ebBPSS.pdf (17.2.03)

[Cook 99] COOK, Jonathan E.; WOLF, Alexander L.: "*Software Process Validation: Quantitatively Measuring the Correspondance of a Process to a Model*". In: ACM Transactions on Software Engineering and Methodology, Vol. 8, Nr. 2, S.147-176, April 1999. http://www.cs.nmsu.edu/~jcook/papers/vjournal.ps.gz (20.12.02)

[Cowan 01] COWAN, John; TOBIN, Richard: *XML Information Set*. W3C-Recommendation. World Wide Web Consortium (W3C), Oktober 2001. http://www.w3.org/TR/2001/REC-xml-infoset-20011024/ (18.9.02)

[CSC 03] CSC: *CSC: Consulting, Systems Integration and Outsourcing*. Website. Computer Sciences Corporation (CSC), El Segund, 2003. http://www.csc.com/ (17.3.03)

[Developmentor 01] DEVELOPMENTOR: *Guerrilla XML – Student Manua*". Seminarunterlagen. Developmentor, Torrance, Mai 2001.

[Draper 02] DRAPER, Denise; u. a.: *XQuery 1.0 and XPath 2.0 Formal Semantics*. W3C-Working-Draft. World Wide Web Consortium (W3C), November 2002. http://www.w3.org/TR/2002/WD-query-semantics-20021115/ (22.4.03)

[ebXML 99] EBXML: *Electronic Business XML Initiative - Terms of Reference*. UN/CEFACT und OASIS, September 1999. http://www.ebxml.org/documents/199909/terms_of_reference.pdf (21.3.03)

[ebXML 03] EBXML: *ebXML - Enabling A Global Electronic Market*. Website. UN/CEFACT und OASIS, 2003. http://www.ebxml.org/ (6.5.03)

[Eddon 99] EDDON, Guy; EDDON, Henry: *Inside COM+ Base Services*. Microsoft Press, Redmond, 1999.

[Eisenberg 01] EISENBERG, Brian; u. a.: *ebXML Technical Architecture Specification v1.0.4.*, Final Draft. UN/CEFACT und OASIS, Februar 2001. http://www.ebxml.org/specs/ebTA.pdf (17.2.03)

[Engesser 88] ENGESSER, Hermann; u. a.: *Duden Informatik - Ein Sachlexikon für Studium und Praxis*. Bibliographisches Institut & F.A. Brockhaus, Mannheim, 1988.

[Ewald 01] EWALD, Tim: "*Interoperability*". In: DEVELOPMENTOR: *Conference .NET*. Konferenzunterlagen. Developmentor, Torrance, August 2001.

[Fallside 01] FALLSIDE, David C.: *XML Schema Part 0: Primer*. W3C-Recommendation. World Wide Web Consortium (W3C), Mai 2001. http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/ (18.9.02)

[Fallside 02] FALLSIDE, David; LAFON, Yves: *XML Protocol Working Group Charter*. World Wide Web Consortium (W3C), Mai 2002. http://www.w3.org/2000/09/XML-Protocol-Charter (2.8.02)

[Fernandez 02] FERNANDEZ, Mary; u. a.*: XQuery 1.0 and XPath 2.0 Data Model*. W3C-Working-Draft. World Wide Web Consortium (W3C), November 2002. http://www.w3.org/TR/2002/WD-query-datamodel-20021115/ (22.4.03)

[Fielding 99] FIELDING. R.; u.a.: *Hypertext Transfer Protocol -- HTTP/1.1*. RFC 2616, Internet Engineering Task Force (IETF), Juni 1999. http://www.ietf.org/rfc/rfc2616.txt (18.9.02)

[Florescu 02] FLORESCU, Daniela; u. a.: "*XL: An XML Programming Language for Web Service Specification and Composition"*. In: LASSNER, David, u. a.: *Proceedings of the eleventh international conference on World Wide Web*. ACM Press, New York, Mai 2002. http://www2002.org/CDROM/refereed/481/ (6.5.03)

[Florescu 03] FLORESCU, Daniela; u. a.: "*XL: A Platform for Web Services*". In: ERCEGOVAC, Vuk: *CIDR 2003 - First Biennial Conference on Innovative Data Systems Research*. Website. University of Wisconsin-Madison, Madison, Januar 2003. http://www-db.cs.wisc.edu/cidr/program/p8.pdf (8.5.03)

[Foody 97] FOODY, Mike: "Le*t's Talk - Getting COM and CORBA talking to each other requires more than just lip service*". In: Byte, Vol. 22, Nr. 4, S. 99-102, April 1997. http://www.byte.com/art/9704/sec8/art2.htm (24.7.02)

[Fraunhofer 02] FRAUNHOFER: *IPSI-XQ - Der XQuery Demonstrator*. Website. Fraunhofer IPSI, Darmstadt, 2002. http://ipsi.fhg.de/oasys/projects/ipsi-xq/index_d.html (25.10.02)

[Freed 96] FREED, Ned; BORENSTEIN, Nathaniel S.: *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*. RFC 2046, Internet Engineering Task Force, November 1996. http://www.ietf.org/rfc/rfc2046.txt (12.4.03)

[Gaul 99] GAUL, Thilo; u. a.: "*Conctruction of Verified Software Systems with Program-Checking: An Application To Compiler Back-Ends*". In: PNUELI, A.; TRAVERSO; P: *Electronic Proceedings of the Federated Logics Conference 99 Workshop on Runtime Result Verification*, 1999. http://www.informatik.uni-kiel.de/~wg/Berichte/RTRV99.ps.gz (16.12.02)

[Goerigk 98] GOERIGK, Wolfgang; u. a.: "*Correct Programs without Proof? On Checker-Based Program Verification*". In: BERGHAMMER, R., LAKHNECH, Y: *Proceedings of the ATOOLA'98 Workshop on "Tool Support for System Specification, Development, and Verification"*. Springer Verlag, New York, S. 108-122, 1998. http://www.informatik.uni-kiel.de/~wg/Berichte/Checker-Tools98.ps.gz (13.12.02)

[Griffel 98] GRIFFEL, Frank: *Componentware - Konzepte und Techniken eines Softwareparadigmas*. Dpunkt, Heidelberg, 1998.

[Gudgin 02a] GUDGIN, Martin; u. a.: *SOAP Version 1.2 Part 1: Messaging Framework*. W3C-Working-Draft. World Wide Web Consortium (W3C), Juni 2002. http://www.w3.org/TR/2002/WD-soap12-part1-20020626/ (6.8.02)

[Gudgin 02b] GUDGIN, Martin; u. a.: *SOAP Version 1.2 Part 2: Adjuncts*. W3C-Working-Draft. World Wide Web Consortium (W3C), Juni 2002. http://www.w3.org/TR/2002/WD-soap12-part2-20020626/ (6.8.02)

[Harmelen 01] HARMELEN, Frank van; u. a.: *Reference description of the DAML+OIL (March 2001) ontology markup language*. Joint US/EU ad hoc Agent Markup Language Committee, März 2001. http://www.daml.org/2001/03/reference (7.3.03)

[Harmelen 03] HARMELEN, Frank van; u. a.: *Web Ontology Language (OWL)- Reference Version 1.0*. W3C-Working-Draft. World Wide Web Consortium (W3C), Februar 2003. http://www.w3.org/TR/2003/WD-owl-ref-20030221/ (3.3.03)

[Hoare 85] HOARE, C.A.R: *Communicating Sequential Processes*. Prentice-Hall, London, 1985.

[Hoffman 00] HOFFMAN, P.; YERGEAU, F.: *UTF-16, an encoding of ISO 10646*. RFC 2781. Internet Engineering Task Force (IETF), Februar 2000. http://www.ietf.org/rfc/rfc2781.txt (18.9.02)

[Hong 02] HONG, Tony; HONG, James: *XMethods*. Website. XMethods, San Jose, 2002. http://www.xmethods.com/ (11.9.02)

[Horrocks 00] HORROCKS, I.; u. a.: *The Ontology Inference Layer OIL*. Technical Report. On-To-Knowledge, 2000. http://www.ontoknowledge.org/oil/TR/oil.long.html (7.3.03)

[HP 03] HP: *Welcome to HP*. Website. Hewlett-Packard (HP), Palo Alto, 2003. http://www.hp.com/ (12.3.03)

[IBM 03] IBM: *IBM United States*. Website. International Business Machines Corporation (IBM), Armonk, 2003. http://www.ibm.com/ (7.5.03)

[Intalio 03] INTALIO: *Intalio, The Business Process Management Company*. Website. Intalio, San Mateo, 2003. http://www.intalio.com/ (7.5.03)

[ISO 03] ISO: *ISO - International Organization for Standardization*. Website. International Organizsation for Standardization (ISO), Genf, 2003. http://www.iso.org/ (6.5.03)

[Kay 01] KAY, Michael: *XSLT - Programmer's Reference*. Wrox Press, Birmingham, 2001.

[Kay 02] KAY, Michael H.: *Saxon - The XSLT Processor*. Website. Open Source Development Network, Boston, August 2002. http://saxon.sourceforge.net/ (25.10.02)

[Kreuz 99] KREUZ, Detlef: *Formale Semantik von Konnektoren*. Dissertation. Technische Universität Hamburg-Harburg, Mai 1999. http://www.ti5.tu-harburg.de/Publication/1999/thesis/Kreuz99/Kreuz99.pdf (6.5.03)

[Lassila 99] LASSILA, Ora; SWICK, Ralph R.: *Resource Description Framework (RDF) Model and Syntax Specification*. W3C-Recommendation. World Wide Web Consortium (W3C), Februar 1999. http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/ (7.3.03)

[LeHors 02] LE HORS, Arnaud; u. a.: *Document Object Model (DOM) Level 3 Core Specification - Version 1.0*, Working-Draft. World Wide Web Consortium (W3C), April 2002. http://www.w3.org/TR/2002/WD-DOM-Level-3-Core-20020409/ (18.9.92)

[Leymann 01] LEYMANN, Frank: *Web Services Flow Language (WSFL 1.0)*. International Business Corporation (IBM), Armonk, Mai 2001. http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf (13.2.03)

[Long 01] LONG, Matt: "*Namespaces*". In: FELL, Simon: *WSDL Issue List*. UserLand Software, Acton, Mai 2001. http://wsdl.soapware.org/stories/storyReader$15 (9.9.02)

[Malhotra 01] MALHOTRA, Ashok; u. a.: *XML Syntax for XQuery 1.0 (XQueryX)*. W3C-Working-Draft, World Wide Web Consortium (W3C), Juni 2001. http://www.w3.org/TR/2001/WD-xqueryx-20010607 (6.5.03)

[Malhotra 02] MALHOTRA, Ashok; u. a.:*XQuery 1.0 and XPath 2.0 Functions and Operators*. W3C-Working-Draft, World Wide Web Consortium (W3C), November 2002. http://www.w3.org/TR/2002/WD-xquery-operators-20021115/ (22.4.03)

[Martin 02] MARTIN, David; u. a.: *DAML-S -Semantic Markup for Web Services*. Joint US/EU ad hoc Agent Markup Language Committee, 2002. http://www.daml.org/services/daml-s/0.7/daml-s.pdf (6.3.03)

[McCarthy 02] MCCARTHY, James: *Reap the benefits of document style Web services - Web services are not exclusively designed for handling remote procedure calls*. International Business Machines Corporation (IBM), Armonk, Juni 2002. http://www-106.ibm.com/developerworks/webservices/library/ws-docstyle.html (11.9.02)

[McIlraith 03] MCILRAITH, Sheila A.; MARTIN, David L.: "*Bringing Semantics to Web Services*". In: IEEE Intelligent Systems, Vol. 18, Nr.1, S. 90- 93, Januar 2003. http://computer.org/intelligent/ex2003/x1090.pdf (3.3.03)

[Mertz 01] MERTZ, David: *Understanding ebXML - Untangling the business Web of the future*. International Business Machines Corporation (IBM), Armonk, Juni 2001. http://www-106.ibm.com/developerworks/xml/library/x-ebxml/index.html (17.2.03)

[Microsoft 02a] MICROSOFT: *Microsoft Corporation*. Website. Microsoft Corporation, Redmond, 2002. http://www.microsoft.com/ (18.7.02)

[Microsoft 02b] MICROSOFT. *Microsoft .NET*. Website. Microsoft Corporation, Redmond, 2002. http://www.microsoft.com/net/ (18.9.02)

[Microsoft 02c] MICROSOFT: *Background: COM Interop in Visual J++ 6.0*. Microsoft Corporation, Redmond, 2002. http://msdn.microsoft.com/library/en-us/dv_vjsharp/html/vjgrfbackgroundcominteropinvisualj60.asp (24.7.02)

[Microsoft 02d] MICROSOFT: *MSXML 4.0 SDK*. Microsoft Corporation, Redmond, März 2002. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/xmlsdk/htm/sdk_intro_6g53.asp (25.10.02)

[Mitra 02] MITRA, Nilo: *SOAP Version 1.2 Part 0: Primer*. W3C-Working-Draft. World Wide Web Consortium (W3C), Juni, 2002. http://www.w3.org/TR/2002/WD-soap12-part0-20020626/ (6.8.02)

[Moreau 02] MOREAU, Jean-Jacques; SCHLIMMER, Jeffrey: *Web Services Description Language (WSDL) Version 1.2: Bindings*. W3C-Working-Draft. World Wide Web Consortium (W3C), Juli 2002. http://www.w3.org/TR/2002/WD-wsdl12-bindings-20020709/ (2.9.02)

[Netzquadrat 02] NETZQUADRAT: *sms.de - dein handy lebt!* Website. Netzquadrat, Düsseldorf, 2003. http://www.sms.de/ (7.6.03)

[OASIS 03] OASIS: *OASIS*. Website. Organization for the Advancement of Structured Information Standards (OASIS), Billerica, 2003. http://www.oasis-open.org/ (7.6.03)

[OMG 02a] OMG: *Object Management Group*. Website. Object Management Group (OMG), Needham, 2002. http://www.omg.org/ (18.7.02)

[OMG 02b] OMG: *The Common Object Request Broker: Architecture and Specification*. Revision 2.6.1. Spezifikation. Object Management Group, Needham, Mai 2002. http://www.omg.org/cgi-bin/doc?formal/02-05-15 (24.7.02)

[Raggett 99] RAGGETT, Dava, u. a.: *HTML 4.01 Specification*. W3C-Recommendation, World Wide Web Consortium (W3C), Dezember 1999. http://www.w3.org/TR/1999/REC-html401-19991224/ (18.9.02)

[RefsnesData 02] REFSNES DATA: *W3Schools Online Web Tutorials*. Website. Refsnes Data, Orre, 2002. http://www.w3schools.com/ (19.9.02)

[Rogerson 97] ROGERSON, Dale: *Inside COM*. Microsoft Press, Redmond, 1997.

[RosettaNet 00] ROSETTANET: *Understanding a PIP Blueprint. Release 1.3*, RosettaNet, Santa Ana, 2000. http://www.rosettanet.org/RosettaNet/Doc/0/OGODJ61V5JA131130304UQ4J39/UG_Understanding_PIP_Blueprint.pdf (20.2.03)

[RosettaNet 01] ROSETTANET: *RosettaNet Background Information*. RosettaNet, Santa Ana, 2001. http://www.rosettanet.org/RosettaNet/Doc/0/36LROCU7QTGKT5383K9A2JL682/Background+Information.pdf (20.2.03)

[RosettaNet 03] ROSETTANET: *Welcome To RosettaNet*. Website. RosettaNet, Santa Ana, 2003. http://www.rosettanet.org/ (12.3.03)

[Rost 03] ROST, Steffen: *XL - XML Language*. Website. Technische Universität München, München, 2003. http://xl.in.tum.de/ (27.3.03)

[Rubinfeld 96] RUBINFELD, Ronitt: "*Designing checkers for programs that run in parallel*". In: Algorithmica, Vol. 15, Nr. 4, S. 287-301, April 1996. http://www.cs.cornell.edu/Info/People/ronitt/PAP/par.ps (12.12.02)

[SAP 99] SAP: *SAP@WEB Installation Guide - Release 4.6b*. SAP, Walldorf, 1999.

[SAP 02a] SAP: *SAP*. Website. SAP, Walldorf, 2002. http://www.sap.com/ (18.9.02)

[SAP 02b] SAP: *Web Service Choreography Interface (WSCI) FAQ*. SAP, Walldorf, Deutschland, 2002. http://ifr.sap.com/wsci/wsci-faq.html (14.3.03)

[SAX 02] SAX-PROJECT: *SAX*. Website. SAX-Project. 2002. http://www.saxproject.org/ (31.7.02)

[Schlimmer 02] SCHLIMMER, Jeffrey C.: *Web Service Description Requirements*. W3C-Working-Draft. World Wide Web Consortium (W3C), April 2002. http://www.w3.org/TR/2002/WD-ws-desc-reqs-20020429/ (7.8.02)

[Schwarze 97] SCHWARZE, Jochen: *Einführung in die Wirtschaftsinformatik*. 4. Auflage. Verlag Neue Wirtschafts-Briefe GmbH & Co., Herne, 1997.

[SeeBeyond 03] SEEBEYOND: *SeeBeyond Technology Corp - Beyond Integration*. Website. SeeBeyond, Monrovia, 2003. http://www.seebeyond.com/ (17.3.03)

[SemanticWeb 02] SEMANTICWEB.ORG: "*Markup Languages and Ontologies*". In: DECKER, Stefan; SINTEK, Michael: *SemanticWeb.org - The Semantic Web Community Portal*, 2002. http://www.semanticweb.org/knowmarkup.html (3.3.03)

[Siegel 00] SIEGEL, Jon: *CORBA 3 - Fundamentals and Programming*, Second Edition. Wiley Computer Publishing, New York, 2000.

[Skonnard 00] SKONNARD, Aaron; u. a: *XML Tutorial*. Developmentor, Torrance, 2000. http://www.develop.com/tutorials/xml/logon.asp (020918)

[Skonnard 02] SKONNARD, Aaron, GUDGIN, Martin: *Essential XML Quick Reference - A Programmer's Reference to XML, XPath, XSLT, XML Schema, SOAP and More*. Addison-Wesley, Boston, 2002.

[SoftwareAG 02a] SOFTWARE AG: *The XML Company - Software AG - Tamino XML Server and Web services*. Website. Software AG, Darmstadt, 2002. http://www.softwareag.com/ (18.7.02)

[SoftwareAG 02b] SOFTWARE AG: "*QuiP – Information*". In: SOFTWARE AG: *Tamino Developer Community*. Website. Software AG, Darmstadt, 2002. http://developer.softwareag.com/tamino/quip/ (25.10.02)

[Soley 01] SOLEY, Richard Mark: *Model Driven Architecture: An Introduction*. Präsentationsfolien. Object Management Group, Needham, 2001. http://www.omg.org/mda/mda_files/MDA_Seminar_Soley6.pdf (31.1.03)

[Sperberg-McQueen 02] SPERBERG-MCQUEEN, C. M.; THOMPSON, Henry: *XML Schema*. Webseite. World Wide Web Consortium (W3C), 2002. http://www.w3.org/XML/Schema (10.12.02)

[Sun 02a] SUN: *Sun Microsystems*. Website. Sun Microsystems, Santa Clara, 2002. http://www.sun.com/ (18.7.02)

[Sun 02b] SUN: *The Only Component Architecture for Java Technology*. Sun Microsystems, Santa Clara, 2002. http://java.sun.com/products/javabeans/ (18.7.02)

[Sun 02c] SUN: *The JavaBeans Bridge for ActiveX*. Sun Microsystems, Santa Clara, Februar 1998. http://java.sun.com/products/javabeans/software/bridge/ (24.7.02)

[Sun 03] SUN: *Java 2 Platform, Enterprise Edition - 1.4 Glossary*. Sun Microsystems, Santa Clara, 2003. http://java.sun.com/j2ee/1.4/docs/glossary.html (6.2.03)

[Szyperski 99] SZYPERSKI, Clemens: *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, Harlow, 1999.

[Thatte 01] THATTE, Satish: *XLANG - Web Services for Business Process Design*. Microsoft, Redmond, 2001. http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/ (13.2.03)

[Thatte 02] THATTE, Satish; u. a.: *Business Process Execution Language for Web Services, Version 1.0*. International Business Machines Corporation (IBM), Armonk, 2002. http://www-106.ibm.com/developerworks/library/ws-bpel/ (10.2.03)

[Thompson 01] THOMPSON, Henry S.; u. a.: *XML Schema Part 1: Structures*. W3C-Recommendation, World Wide Web Consortium (W3C), Mai 2001. http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/ (18.9.02)

[TU-München 03] TU-MÜNCHEN: *Technische Universität München*. Website. Technischen Universität München, München, 2003. http://www.tum.de/ (27.3.03)

[UDDI 00] UDDI.ORG: *UDDI Technical White Paper*. Organization for the Advancement of Structured Information Standards (OASIS), Billerica, September 2000. http://www.uddi.org/pubs/Iru_UDDI_Technical_White_Paper.pdf (25.2.03)

[UDDI 01] UDDI.ORG: *Version 2.0 UDDI XML Schema 2001*. XML-Schemadokument. Organization for the Advancement of Structured Information Standards (OASIS), Billerica, 2001. http://uddi.org/schema/uddi_v2.xsd (11.3.03)

[UN/CEFACT 03] UN/CEFACT: *UN/CEFACT - United Nations Centre for Trade Facilitation and Electronic Business*. Website. United Nations Centre for Trade Facilitation and Electronic Business (UN/CEFACT), Genf, 2003. http://www.unece.org/cefact/ (21.3.03).

[Unicode 03] UNICODE CONSORTIUM: *Unicode Home Page*. Website. The Unicode Consortium, Mountain View, 2003. http://www.unicode.org/ (8.5.03)

[Valk 01] VALK, Rüdiger: *Prozesse und Nebenläufigkeit (PNL)*.Vorlesungsunterlagen. Universität Hamburg, 2001. http://www.informatik.uni-hamburg.de/TGI/lehre/vl/WS0102/PNL/pnl.html (6.5.03)

[Versata 03] VERSATA: *Welcome to Versata*. Website. Versata, Oakland, 2003. http://www.versata.com/ (17.3.03)

[W3C 02a] W3C: *World Wide Web Consortium*. Website. World Wide Web Consortiums (W3C), 2002. http://www.w3.org/ (29.7.02)

[W3C 02b] W3C: *Schema defined in the SOAP Version 1.2 Part 1 specification*. XML-Schemadokument, Working-Draft. World Wide Web Consortium (W3C), Juni 2002. http://www.w3.org/2002/06/soap-envelope/ (6.5.03)

[Wasserman 97] WASSERMAN, Hal; BLUM, Manuel: "*Software Reliability via Run-Time Result-Checking*". In: Journal of the ACM, Vol. 44, Nr. 6, S. 826-849, November 1997. http://http.cs.berkeley.edu/~blum/fourier.ps (13.12.02)

[Yergeau 98] YERGEAU, F.: *UTF-8, a transformation format of ISO 10646*. RFC 2279. Internet Engineering Task Force (IETF), Januar 1998. http://www.ietf.org/rfc/rfc2279.txt (18.9.02)

Hinweis: Die Datumsangabe in Klammern hinter jeder URL gibt den letzten Zugriff des Autors dieser Arbeit auf das Dokument an. An diesem Tag war die URL also gültig.

Lebenslauf

Name		Marcus Venzke
Geburtstag		20. April 1968
Geburtsort		Hamburg
Familienstand		ledig
Staatsangehörigkeit		deutsch
Schulische Ausbildung	1974 - 1978	Grundschule
	1978 - 1988	Gymnasium abgeschlossen mit der allgemeinen Hochschulreife
Wehrdienst	1988 - 1989	
Studium	1990 - 1998	Studium der Informatik an der Universität Hamburg abgeschlossen mit dem Diplom der Informatik Diplomarbeit: „Integration von Standard-Geschäftsanwendungen in verteilte transaktionale Komponentenarchitekturen am Beispiel von SAP R/3“
	1999 - 1999	Wissenschaftliche Fortbildung an der Technischen Universität Hamburg-Harburg
Berufliche Tätigkeit	1989 - 1998	Selbständige Tätigkeit im Bereich der Softwareerstellung
	2000 - 2003	Wissenschaftlicher Mitarbeiter am Arbeitsbereich Telematik an der Technischen Universität Hamburg-Harburg (Leitung: Prof. Dr. F. H. Vogt)