

Diplomarbeit

**Ein Verfahren zur abstrakten Interpretation von  
XPath-Ausdrücken in WS-BPEL-Prozessen**

Katharina Görlach

25. März 2008



Humboldt-Universität zu Berlin  
Mathematisch-Naturwissenschaftliche Fakultät II  
Institut für Informatik

Gutachter:  
Prof. Dr. Wolfgang Reisig  
Prof. Dr. Karsten Wolf



## **Zusammenfassung**

Die Web Services Business Process Execution Language ist eine Sprache zur Modellierung von Geschäftsprozessen als Web Services. Für eine umfassende Analyse von WS-BPEL-Prozessen müssen auch die Daten der Prozesse analysiert werden. Daten werden in WS-BPEL-Prozessen mit Hilfe von XML-Schema typisiert und standardmäßig mit Hilfe von XPath manipuliert. Eine Datenanalyse für WS-BPEL muss deshalb XML-Schema berücksichtigen die im Prozess enthaltenen XPath-Ausdrücke auswerten. Eine solche Datenanalyse ermöglicht Rückschlüsse auf den Kontrollfluss und dient so beispielsweise zur Optimierung eines WS-BPEL-Prozesses.

In der vorliegenden Arbeit wird ein Verfahren zur abstrakten Interpretation von XPath-Ausdrücken in WS-BPEL-Prozessen vorgestellt. Dafür wird ein umfassendes Datenmodell für WS-BPEL-Prozesse sowie die enthaltenen XPath-Ausdrücke entwickelt. Auf Grundlage des entwickelten Datenmodells stellen wir eine statische Analyse vor, die die XPath-Ausdrücke in einem WS-BPEL-Prozess abstrakt interpretiert. Die Analyse berechnet dabei die Wertebereiche für Variablen und Bedingungen in WS-BPEL-Prozessen.



# Inhaltsverzeichnis

<b>Abbildungsverzeichnis</b>	<b>VI</b>
<b>Algorithmenverzeichnis</b>	<b>VII</b>
<b>1. Einführung</b>	<b>1</b>
1.1. Motivation . . . . .	1
1.2. Einordnung der Arbeit . . . . .	3
1.3. Ziel der Arbeit . . . . .	5
<b>2. Grundlagen</b>	<b>7</b>
2.1. Web Services Business Process Execution Language . . . . .	7
2.2. XML-Schema . . . . .	12
2.3. XPath . . . . .	25
<b>3. Ein Datenmodell für WS-BPEL-Prozesse</b>	<b>35</b>
3.1. Ein formales Datenmodell für XPath . . . . .	35
3.2. Drei-Adress-Code . . . . .	68
3.3. Concurrent-Single-Static-Assignment-Form . . . . .	73
<b>4. Statische Analyse von XPath-Ausdrücken in WS-BPEL-Prozessen</b>	<b>91</b>
4.1. Analyse des CSSA-Graphen . . . . .	92
4.2. Analyse der XPath-Ausdrücke . . . . .	111
<b>5. Zusammenfassung und Ausblick</b>	<b>141</b>
5.1. Zusammenfassung der Ergebnisse . . . . .	141
5.2. Ausblick . . . . .	144
<b>A. Mathematische Grundlagen</b>	<b>147</b>
A.1. Die Galois Korrespondenz . . . . .	148
<b>B. Hilfsalgorithmen</b>	<b>149</b>
B.1. Hilfsalgorithmen für den Algorithmus analyseCSSA() . . . . .	149
<b>C. Beispiel</b>	<b>159</b>
<b>Literaturverzeichnis</b>	<b>163</b>
<b>Erklärung</b>	<b>167</b>

## Abbildungsverzeichnis

1.1. Serviceorientierte Architektur . . . . .	3
1.2. Übersicht zur vorliegenden Arbeit . . . . .	5
2.1. Das Schema einer Adresse. . . . .	13
2.2. Ein Element vom Typ Adresse. . . . .	14
2.3. Ein XML-Schema mit einer Referenz. . . . .	16
2.4. Das Schema einer Buchbestellung . . . . .	19
3.1. Galois Korrespondenz . . . . .	36
3.2. Die SSA-Form . . . . .	74
3.3. Der CSSA-Graph für einen WS-BPEL-Prozess. . . . .	75
3.4. Der CSSA-Teilgraph für eine receive-Aktivität. . . . .	77
3.5. Der CSSA-Teilgraph für eine reply-Aktivität. . . . .	78
3.6. Der CSSA-Teilgraph für eine invoke-Aktivität. . . . .	79
3.7. Der CSSA-Teilgraph für eine assign-Aktivität. . . . .	80
3.8. Der CSSA-Teilgraph für eine if-Aktivität. . . . .	81
3.9. Der CSSA-Teilgraph für eine while-Aktivität. . . . .	81
3.10. Der CSSA-Teilgraph für eine repeatUntil-Aktivität. . . . .	82
3.11. Der CSSA-Teilgraph für eine forEach-Aktivität. . . . .	83
3.12. Der CSSA-Teilgraph für eine pick-Aktivität. . . . .	85
3.13. Der CSSA-Teilgraph für eine flow-Aktivität. . . . .	86
3.14. Der CSSA-Teilgraph für eine scope-Aktivität. . . . .	87
3.15. Die CSSA-Teilgraphen für die sequence-, empty-, wait-Aktivitäten. . . . .	88
3.16. Ein CSSA-Teilgraph mit Drei-Adress-Code. . . . .	89
C.1. Ein Ausschnitt aus einem WS-BPEL-Prozess. . . . .	159
C.2. Der vollständige CSSA-Graph für das Beispiel aus Abb. C.1. . . . .	162

## Algorithmenverzeichnis

1.	analyseCSSA(CSSA-Graph, startknoten, Werteliste, flow, schleife) . . . . .	94
2.	analysePi(knoten, Werteliste) . . . . .	101
3.	analysePhi(knoten, Werteliste) . . . . .	103
4.	widening( $W_{\text{Phi1}}$ , $W_{\text{Phi2}}$ ) . . . . .	106
5.	speichern(var, W, OUT, aktuell, inflow, schleife) . . . . .	109
6.	zusammenfassen(Werteliste) . . . . .	111
7.	analyseXPath(Ausdruck, Werteliste) . . . . .	113
8.	analyseOperation(operation, operand1, operand2, Ausdruck, Werte) . . . . .	117
9.	analyseNumerisch(operation, operand1, operand2) . . . . .	120
10.	analyseKonvertVergleich(operation, operand1, operand2) . . . . .	122
11.	analyseVergleich(operation, operand1, operand2) . . . . .	124
12.	analyseAchse(knoten, achse) . . . . .	126
13.	analyseAncestor((knoten, baum)) . . . . .	128
14.	analyseChild((knoten, baum)) . . . . .	129
15.	analyseDescendant((knoten, baum)) . . . . .	130
16.	analyseFollowing((knoten, baum)) . . . . .	132
17.	analysePreceding((knoten, baum)) . . . . .	133
18.	analyseAttributNamespace((knoten, baum), Typ) . . . . .	134
19.	analyseKnotentyp(Knotenmenge) . . . . .	135
20.	analyseKnotentest(Knotenmenge, operand2) . . . . .	136
21.	analyseFkt(funktion, zähler, Ausdruck, Werte) . . . . .	139
22.	teilgraphFlow(CSSA-Graph, startknoten) . . . . .	150
23.	startZyklus(CSSA-Graph, start, aktuell, Besucht) . . . . .	154
24.	teilgraphSchleife(CSSA-Graph, startknoten, Pfad) . . . . .	156



# 1. Einführung

Diese Arbeit ist in das Umfeld der Geschäftsprozessmodellierung und der statischen Programmanalyse einzuordnen. Wir betrachten die Modellierung verteilter Geschäftsprozesse mit Web Services unter Verwendung der *Web Services Business Process Execution Language* (WS-BPEL)<sup>1</sup>. Solche verteilten Geschäftsprozesse werden dabei oft in einer sogenannten *Choreographie* zu einem „größeren“ Geschäftsprozess miteinander kombiniert. Die *statische Analyse* einzelner modellierter Geschäftsprozesse kann verschiedene Aspekte eines Geschäftsprozesses betrachten. In dieser Arbeit analysieren wir die Dateninformationen in Geschäftsprozessen.

Die Analyse von Dateninformationen in BPEL-Prozessen ist besonders interessant, da sie Rückschlüsse für den Kontrollfluss zulässt. So kann der Kontrollfluss optimiert werden, indem beispielsweise sogenannte „tote Pfade“ eliminiert werden. In Choreographien können die Dateninformationen eines BPEL-Prozesses sogar für die Optimierung aller anderen BPEL-Prozesse der Choreographie verwendet werden. Aber auch bestehende Verifikationstechniken für BPEL-Prozesse können durch Dateninformationen verbessert werden.

In dieser Arbeit wird eine statische Analyse für Daten in einem BPEL-Prozess vorgestellt. Zunächst erstellen wir ein formales Datenmodell, das eine korrekte Abstraktion von den konkreten Datenwerten zusichert. Auf Grundlage dieses Datenmodells stellen wir eine statische Datenanalyse vor, die die Wertebereiche von Daten in einem BPEL-Prozess analysiert.

## 1.1. Motivation

Arbeiten zur statischen Analyse eines BPEL-Prozesses bezogen sich in der Vergangenheit fast ausschließlich auf den Kontrollfluss. Diese Analysen sind zahlreich und werden beispielsweise in [Sch05], [MM06], [OVA<sup>+</sup>07], [LMSW08] und [FUMK04] vorgestellt. So wird in [Sch05] z.B. die Bedienbarkeit eines BPEL-Prozesses analysiert. Dabei wird analysiert, ob ein Partnerprozess existiert, der mit dem betrachteten BPEL-Prozess kommunizieren kann. Da die Analyse der Bedienbarkeit Datenabhängigkeiten vernachlässigt, werden Verzweigungen im BPEL-Prozess durch Nichtdeterminismus abgebildet. Der Nichtdeterminismus bietet aber eine weniger präzise Grundlage für die Analyse. In den genannten Arbeiten werden aber auch andere Eigenschaften, wie z.B. die Korrektheit und Kompatibilität von BPEL-Prozessen, auf Grundlage der Kontrollflussinformationen analysiert.

---

<sup>1</sup>Im folgenden Text verwenden wir die Abkürzung BPEL für die Sprache WS-BPEL.

Es gibt nur wenige Arbeiten, die sich explizit mit der Datenanalyse für BPEL-Prozesse befassen. So wird in [God06] eine Concurrent-Single-Static-Assignment-Form (CSSA-Form) für BPEL-Prozesse vorgestellt. Eine CSSA-Form ist ein Graph, der eine kompakte Repräsentation des Kontrollflusses und der Datenabhängigkeiten ermöglicht. In [MMG<sup>+</sup>07] wird auf Grundlage dieser CSSA-Form eine statische Datenanalyse für BPEL-Prozesse vorgestellt, die die Abhängigkeiten von zu sendenden Nachrichten und empfangenen Nachrichten analysiert. Dabei werden zunächst die kommunikationsrelevanten Anweisungen im BPEL-Prozess ermittelt. Danach werden die ermittelten Anweisungen entsprechend des Kontrollflusses miteinander verknüpft, so dass entschieden werden kann, wie das Senden einer Nachricht von vorher empfangenen Nachrichten abhängt.

In den genannten Arbeiten zur Datenanalyse wird ein wesentlicher Aspekt außer Acht gelassen: die vollständige Analyse der im BPEL-Prozess enthaltenen *XPath-Ausdrücke*. Da in BPEL-Prozessen die Daten standardmäßig mit Hilfe von XPath-Ausdrücken manipuliert werden, ist aber eine Analyse dieser XPath-Ausdrücke notwendig, um die Daten in einem BPEL-Prozess umfassend analysieren zu können. Weiterhin muss eine umfassende Datenanalyse für BPEL-Prozesse *XML-Schema* berücksichtigen, da XML-Schema für die Typisierung von Daten in BPEL-Prozessen verwendet wird.

In der vorliegenden Arbeit möchten wir nun die Grundlagen für eine umfassende Datenanalyse für BPEL-Prozesse schaffen und eine solche Datenanalyse realisieren. Zunächst werden wir dafür ein Datenmodell für BPEL-Prozesse und die enthaltenen XPath-Ausdrücke entwickeln. Weiterhin stellen wir auf Grundlage dieses Datenmodells eine statische Analyse von XPath-Ausdrücken in BPEL-Prozessen vor, die auch XML-Schema berücksichtigt. Das Ziel unserer Analyse ist es, die Wertebereiche von Daten in einem BPEL-Prozess zu ermitteln.

Die Ergebnisse dieser Arbeit können danach vielfältig weiter verwendet werden. Beispielsweise können weitere Datenanalysen für BPEL-Prozesse auf dieser Arbeit aufbauen. Eine Datenanalyse, die ebenfalls XPath-Ausdrücke in BPEL-Prozessen aber andere Aspekte als den Wertebereich betrachtet, kann das entwickelte Datenmodell übernehmen. Sollen andere Ausdruckssprachen als XPath für die Datenmanipulation in BPEL-Prozessen analysiert werden, kann das entwickelte Datenmodell an die Bedürfnisse dieser Analysen angepasst werden.

Weiterhin können die Ergebnisse dieser Arbeit für die Optimierung von BPEL-Prozessen verwendet werden, da das Ergebnis einer Datenanalyse Rückschlüsse für den Kontrollfluss zulässt. Beispielsweise können sogenannte „tote Pfade“ in einem BPEL-Prozess erkannt und gelöscht werden. Die Entfernung der toten Pfade ermöglicht eine kürzere Ausführungszeit des BPEL-Prozesses. Neben der Optimierung können die Analyseergebnisse auch für die Verbesserung von bestehenden Verifikationstechniken für BPEL-Prozesse verwendet werden. Solche Verifikationstechniken werden beispielsweise in [Sch05] und [MM06] vorgestellt. Bisher vernachlässigen diese Techniken Dateninformationen und bilden deshalb Verzweigungen durch Nichtdeterminismus ab. Werden die Ergebnisse unserer Datenanalyse aber in die Betrachtungen einbezogen, kann der Nichtdeterminismus aufgelöst und die Verifikationstechniken aussagekräftiger werden.

## 1.2. Einordnung der Arbeit

Für Unternehmen ist die Automatisierung betrieblicher Abläufe unverzichtbar. Betriebliche Abläufe werden *Geschäftsprozesse* genannt und sollen nach Möglichkeit autonom und verteilt ausgeführt werden können. Weiterhin solchen verteilt ausgeführte Geschäftsprozesse zu „größeren“ Geschäftsprozessen kombiniert werden können. Das Konzept der *Serviceorientierten Architektur* bietet eine Realisierung für die Automatisierung von Geschäftsprozessen mit diesen Eigenschaften. In einer Serviceorientierten Architektur wird jeder Geschäftsprozess durch einen *Web Service* realisiert. Ein Web Service ist dabei eine Software-Komponente, die einen Geschäftsprozess implementiert.

In einer Serviceorientierten Architektur sind Web Services unabhängig und lose gekoppelt. Dadurch können Web Services verteilt ausgeführt und zu „größeren“ Web Services miteinander kombiniert werden. Abbildung 1.1 zeigt die grundlegenden Konzepte einer Serviceorientierten Architektur. Um die Funktionalität eines Web Services (*Service-Anbieter*) zur Verfügung zu stellen, veröffentlicht der Eigentümer den Web Service in einem Verzeichnis (*Service-Register*). Dabei wird eine Beschreibung der bereitgestellten

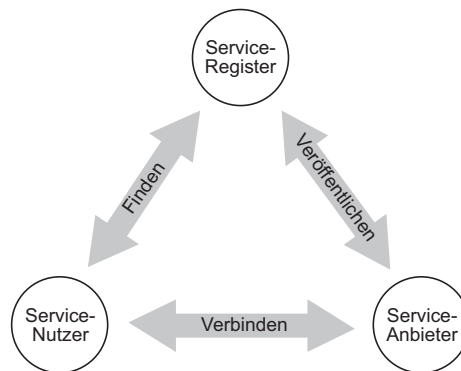


Abbildung 1.1.: Serviceorientierte Architektur

Funktionalität sowie eine Spezifikation der Schnittstelle veröffentlicht. Ein potentieller Nutzer (*Service-Nutzer*) des Web Services kann nun das Service-Register durchsuchen und anhand der Funktionalitätsbeschreibungen den passenden Web Service finden. Danach kann sich der Service-Nutzer mit Hilfe der Schnittstellenbeschreibung mit dem Service-Anbieter verbinden.

Ein Web Service kann beispielsweise mit Hilfe der *Web Services Business Process Execution Language* implementiert werden. BPEL [JE07] unterstützt dabei verschiedene Techniken, die für die Realisierung von verteilt ausgeführten und lose gekoppelten Web Services benötigt werden. So wird eine Schnittstelle in BPEL mit Hilfe der *Web Service Description Language* (WSDL) [CCMW01] beschrieben. Das Kombinieren einzelner Web Services zu einem „größeren“ Web Service wird durch die Unterstützung der *Service Component Architecture* (SCA) [OSO07] ermöglicht. Da wir uns in dieser Arbeit

auf die Datenanalyse eines BPEL-Prozesses beschränken, erläutern wir diese Techniken an dieser Stelle nicht genauer.

Zur Beschreibung von Daten unterstützt BPEL die Technologien XML-Schema und XPath. *XML-Schema* [TBMM04], [BM04] wird für die Typisierung von Daten in einem BPEL-Prozess verwendet. Dabei ermöglicht XML-Schema die Beschreibung von hierarchisch strukturierten Daten. Zur Manipulation von Daten unterstützt BPEL standardmäßig die Ausdruckssprache *XPath* [CD99]. Ein XPath-Ausdruck wird einerseits dazu verwendet, die verschiedenen Hierarchie-Stufen in den Daten zu adressieren. Andererseits können in einem XPath-Ausdruck auch Berechnungen auf den Daten vorgenommen werden.

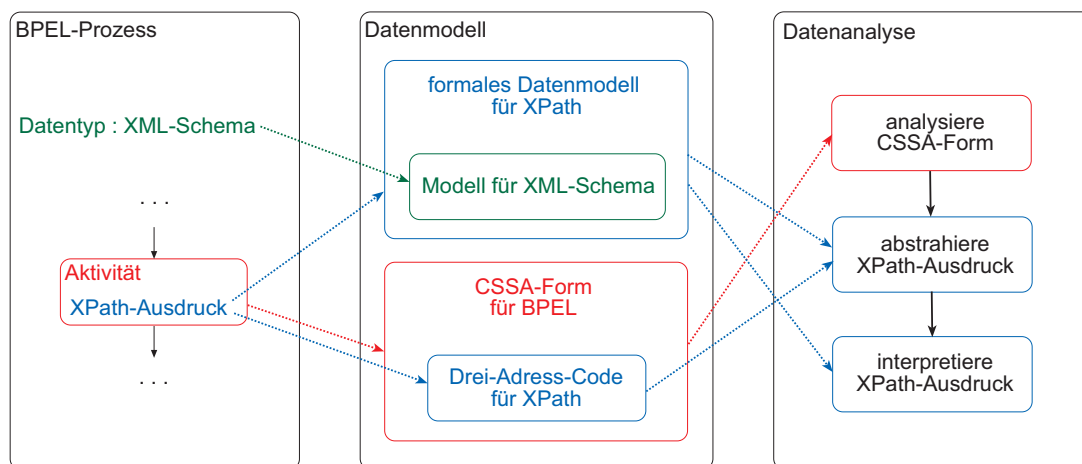
Da XPath zur Manipulation von Daten in einem BPEL-Prozess verwendet wird, müssen die XPath-Ausdrücke betrachtet werden, wenn wir eine statische Datenanalyse für BPEL-Prozesse realisieren möchten. Die *statische Datenanalyse* ist seit vielen Jahren aus dem Compilerbau bekannt. Bei einer statischen Analyse werden Implementierungen vor ihrer Ausführung analysiert. Deshalb können einige Informationen, die erst zur Laufzeit zur Verfügung stehen, in einer statischen Analyse nicht berücksichtigt werden. Ein Teilgebiet der statischen Analyse ist die *abstrakte Interpretation* [CC77]. Bei der abstrakten Interpretation wird zunächst von den konkreten Datenwerten abstrahiert. Auf Grundlage der abstrakten Datenwerte werden dann die Anweisungen in der betrachteten Implementierung ausgeführt. In dieser Arbeit stellen wir ein Verfahren zur abstrakten Interpretation für XPath-Ausdrücke vor. Dabei geben wir ein formales Datenmodell an, das von den konkreten Datenwerten in XPath abstrahiert. Auf Grundlage dieses formalen Datenmodells geben wir eine statische Analyse an, die die XPath-Ausdrücke auswertet.

Die Analyse von XPath-Ausdrücken ist vor allem durch den Kontrollfluss sinnvoll, der durch den BPEL-Prozess gegeben ist. In [Sch07] wird deshalb eine Analyse für Verzweigungen in BPEL-Prozessen vorgestellt. In der Arbeit wird überprüft, ob entschieden werden kann, dass eine Verzweigung exklusiv ausgeführt wird. Für Verzweigungen in einem BPEL-Prozess werden Bedingungen standardmäßig mit Hilfe von XPath beschrieben. Deshalb wird in der Arbeit auch ermittelt, für welche Klassen von XPath-Ausdrücken eine solche Entscheidung möglich ist.

Weiterhin gibt es viele Arbeiten, die sich mit der Analyse von XPath im Allgemeinen, d.h. unabhängig von einer anderen Technologie, beschäftigen. Beispielsweise wird in [GVD04] eine Möglichkeit zur Optimierung von XPath-Ausdrücken vorgestellt. In [GLS07] wird wiederum die Entscheidbarkeit von verschiedenen Eigenschaften für XPath-Ausdrücke analysiert. Hier überprüfen Pierre Genevès und Nabil Layaïda u.a., ob statisch entschieden werden kann, dass zwei XPath-Ausdrücke das gleiche Ergebnis liefern. Aber auch die Entscheidbarkeit, ob ein XPath-Ausdruck erfüllbar ist oder ob das Ergebnis eines XPath-Ausdrucks leer ist, wird in der Arbeit betrachtet. Wie sich Vorgaben für Daten auf die Entscheidbarkeit des Erfüllbarkeitsproblem auswirken, wird in [BFG05] überprüft. Solche Vorgaben werden in BPEL-Prozessen durch XML-Schema geliefert.

### 1.3. Ziel der Arbeit

Das Ziel dieser Arbeit ist, die Grundlagen für eine umfassende statische Datenanalyse für BPEL-Prozesse zu schaffen und eine solche Datenanalyse zu realisieren. Dazu erstellen wir im ersten Teil der Arbeit ein geeignetes Datenmodell für BPEL-Prozesse, das die Grundlage für eine umfassende Datenanalyse bildet. Abbildung 1.2 veranschaulicht dies. Zum Einen entwickeln wir im Datenmodell ein *formales Datenmodell für XPath*, auf dessen Grundlage XPath-Ausdrücke abstrakt interpretiert werden können. In diesem formalen Datenmodell berücksichtigen wir auch die Repräsentation von Daten, die mit Hilfe von XML-Schema typisiert werden. Zum Anderen beschreiben wir im Datenmodell eine *CSSA-Form* zur Repräsentation von BPEL-Prozessen. In diese CSSA-Form betten wir den *Drei-Adress-Code* als Repräsentationsform für XPath-Ausdrücke ein.



**Abbildung 1.2.:** In der vorliegenden Arbeit entwickeln wir ein Datenmodell für BPEL-Prozesse und stellen auf Grundlage dieses Datenmodells eine statische Datenanalyse für BPEL-Prozesse vor. Im Datenmodell beschreiben wir zum Einen ein formales Modell für XPath. Zum Anderen beschreiben wir eine analysierbare Repräsentation für BPEL-Prozesse und die enthaltenen XPath-Ausdrücke. Bei der statischen Datenanalyse, die in dieser Arbeit vorgestellt wird, werden die XPath-Ausdrücke im BPEL-Prozess auf Grundlage des Datenmodells abstrakt interpretiert.

Im zweiten Teil der Arbeit stellen wir eine statische Datenanalyse für BPEL-Prozesse auf Grundlage des entwickelten Datenmodells vor. Diese Analyse ermittelt den *Wertebereich* von Variablen und den Wertebereich der Ergebnisse von Bedingungen. Dazu traversieren wir über den BPEL-Prozess und ermitteln die enthaltenen XPath-Ausdrücke. Diese XPath-Ausdrücke werden dann auf Grundlage des formalen Datenmodells abstrakt interpretiert.

Die vorliegende Arbeit gliedert sich neben der Einführung in vier Kapitel. In Kapitel 2 werden die grundlegenden Technologien vorgestellt, die bei unserer statischen Datenanalyse für BPEL-Prozesse betrachtet werden. Zunächst geben wir eine Einführung in BPEL und erläutern dann die in BPEL verwendeten Sprachen zur Typisierung (XML-

Schema) und Manipulation (XPath) von Daten. In Kapitel 3 entwickeln wir danach ein Datenmodell für BPEL-Prozesse. Hier definieren wir zunächst ein formales Datenmodell für XPath. Danach beschreiben wir ein Modell zur Repräsentation der XPath-Ausdrücke in BPEL-Prozessen. Um das Datenmodell für BPEL-Prozesse zu vervollständigen, stellen wir schließlich ein Modell zur Repräsentation des Kontrollflusses und der Datenabhängigkeiten eines BPEL-Prozesses vor. In Kapitel 4 beschreiben wir die statische Analyse von Wertebereichen in BPEL-Prozessen auf Grundlage des entwickelten Datenmodells. Schließlich fassen wir in Kapitel 5 die Ergebnisse der vorliegenden Arbeit zusammen und geben einen Ausblick zu weiterführenden Arbeiten.

Zusätzlich zu den genannten Kapiteln enthält die Arbeit drei Anhänge. In Anhang A werden die mathematischen Grundlagen für das formale Datenmodell aus Kapitel 3 erläutert. Anhang B enthält einige Hilfsalgorithmen für die Datenanalyse aus Kapitel 4. Schließlich geben wir in Anhang C ein Beispiel an und stellen die Ausführung der Datenanalyse für dieses Beispiel dar.

## 2. Grundlagen

Nachdem wir im letzten Kapitel das Umfeld und das Ziel dieser Arbeit beschrieben haben, stellen wir in diesem Kapitel die grundlegenden Technologien vor, die in dieser Arbeit betrachtet werden. Zunächst geben wir in Kapitel 2.1 eine Einführung in BPEL. Danach erläutern wir in Kapitel 2.2 die Sprache XML-Schema, die für die Typisierung von Daten in BPEL-Prozessen verwendet wird. Schließlich stellen wir in Kapitel 2.3 die Sprache XPath vor, die standardmäßig zur Manipulation von Daten in BPEL-Prozessen verwendet wird. Bei der Beschreibung dieser drei Technologien beschränken wir uns auf die Konzepte, die für diese Arbeit wesentlich sind.

### 2.1. Web Services Business Process Execution Language

In diesem Kapitel stellen wir die *Web Services Business Process Execution Language 2.0* [JE07] vor. BPEL ist eine Sprache für die Beschreibung von verteilten Geschäftsprozessen. Mit Hilfe von BPEL wird dabei die Funktionalität eines Geschäftsprozesses beschrieben. Dabei können sowohl ausführbare als auch abstrakte Geschäftsprozesse modelliert werden. In dieser Arbeit betrachten wir nur ausführbare Geschäftsprozesse, da in diesen Prozessen alle Dateninformationen zugänglich sind.

Geschäftsprozesse kommunizieren miteinander über Nachrichtenaustausch. Für den Nachrichtenaustausch wird für jeden Geschäftsprozess eine Schnittstelle mit Hilfe der *Web Service Description Language* (WSDL) [CCMW01] beschrieben. Bei der Schnittstellenbeschreibung werden alle Informationen angegeben, die für die Nutzung des Geschäftsprozesses nötig sind. So werden verschiedene Operationen für einen Geschäftsprozess definiert, die von einem anderen Prozess aufgerufen werden können. Für jede Operation wird der Name der Operation, der Übergabeparameter und der Rückgabewert definiert. Der Übergabeparameter und der Rückgabewert repräsentieren die Nachrichten, die bei der Kommunikation zwischen zwei Geschäftsprozessen ausgetauscht werden.

Eine Nachricht in BPEL-Prozessen ist ein XML-Dokument. Die *Extensible Markup Language* (XML) [BPSM<sup>+</sup>04] ist eine Sprache zur Darstellung hierarchisch strukturierter Daten. Ein XML-Dokument ist eine Textdatei, in der die Daten durch sogenannte *XML-Elemente* beschrieben werden. Ein solches XML-Element kann Kindelemente, Attribute sowie freien Text enthalten. Eine Nachricht in BPEL-Prozessen besteht aus einem oder mehreren Teilen, sogenannten *Parts*. Jeder Part verweist dabei auf ein XML-Element von einem bestimmten Typ. Der Typ dieses XML-Elements wird durch ein *XML-Schema* [TBMM04] beschrieben.

Bei der Ausführung eines BPEL-Prozesses wird eine Instanz erzeugt. Um sicher zu stellen, dass eine Nachricht an die richtige Instanz adressiert wird, erhält eine Nachricht ein *CorrelationSet*. Im *CorrelationSet* wird eine ID beschrieben, die einer bestimmten Instanz des BPEL-Prozesses zugeordnet ist. Da eine Nachricht an mehrere Instanzen geschickt werden kann, können im *CorrelationSet* mehrere IDs beschrieben werden.

Nachrichten sowie andere Daten eines BPEL-Prozesses werden in *Variablen* gespeichert. Eine Variable hat immer einen eindeutigen Namen und einen ausgezeichneten Datentyp. Der Datentyp einer Variablen wird durch ein XML-Schema beschrieben. Zusätzlich können für Variablen in einem BPEL-Prozess sogenannte *properties* definiert werden. Eine *property* benennt dabei ein konkretes Datum<sup>1</sup> in einer Variablen. Um den Namen, der in der *property* angegeben ist, mit einem konkreten Datum zu verbinden, wird ein *propertyAlias* spezifiziert. In diesem *propertyAlias* wird ein XPath-Ausdruck angegeben, um das konkrete Datum zu adressieren. Für eine leichtere Datenadressierung im BPEL-Prozess kann dann die *property* verwendet werden.

Anweisungen in BPEL-Prozessen werden durch *Aktivitäten* repräsentiert. Die Aktivitäten sind in Basisaktivitäten und strukturierte Aktivitäten klassifiziert. Basisaktivitäten sind eigenständige Anweisungen. Die strukturierten Aktivitäten können wiederum andere Aktivitäten enthalten. Für die Behandlung spezieller Situationen bei der Ausführung eines BPEL-Prozesses, wie z.B. das Auftreten eines Fehlers oder anderer Ereignisse, gibt es sogenannte *Handler*. Die *Handler* beschreiben dabei das Verhalten des BPEL-Prozesses in den jeweiligen Situationen.

### 2.1.1. Aktivitäten in WS-BPEL

Im Folgenden beschreiben wir grundlegend die Aktivitäten aus der aktuellen BPEL-Spezifikation. Die Aktivitäten *throw*, *rethrow*, *compensate*, *compensateScope* und *exit* berücksichtigen wir dabei nicht, da in unserer Analyse keine *Handler* betrachtet werden. Eine genaue Beschreibung aller Aktivitäten ist in der BPEL-Spezifikation [JE07] enthalten.

#### Basisaktivitäten

**receive** empfängt und speichert eine Nachricht von einem Partnerservice<sup>2</sup>. In der Aktivität wird spezifiziert, über welche Schnittstelle die Nachricht empfangen und in welcher Variablen die Nachricht gespeichert wird. Dabei kann sowohl die gesamte Nachricht als auch nur ein Teil der Nachricht gespeichert werden.

---

<sup>1</sup>Datum benennt den Singular von Daten.

<sup>2</sup>Ein Partnerservice ist ein Service, mit dem der betrachtete BPEL-Prozess kommuniziert.

**reply** sendet eine Nachricht an einen Partnerservice. Sie spezifiziert über welche Schnittstelle die Nachricht gesendet wird und wo die Nachricht im BPEL-Prozess gespeichert ist. Eine solche Nachricht kann in einer Variablen gespeichert sein, oder aus den Daten mehrerer Variablen zusammengesetzt werden.

**invoke** sendet eine Nachricht an einen Partnerservice und empfängt ggf. eine Antwort von diesem Partnerservice. Ob der Partnerservice eine Antwort liefert, ist in der Schnittstelle spezifiziert. Wird keine Antwort vom Partnerservice geliefert, ist das Verhalten der invoke-Aktivität analog zum Verhalten der reply-Aktivität. Wird eine Antwort vom Partnerservice geliefert, wartet die invoke-Aktivität so lange, bis die Antwort empfangen wurde. Das bedeutet, dass die Ausführung des BPEL-Prozesses so lange unterbrochen wird, bis eine Antwort empfangen wird. Danach wird die gesamte Antwort oder ein Teil der Antwort in einer Variablen gespeichert.

**assign** repräsentiert eine Zuweisung im BPEL-Prozess. Mit Hilfe einer Zuweisung kann der Wert einer oder mehrerer Variablen aktualisiert werden. Für jede Aktualisierung enthält die assign-Aktivität ein *copy-Element*. In diesem copy-Element wird spezifiziert, welcher Variablen in der Aktivität welcher Wert zugewiesen wird.

**empty** repräsentiert eine leere Anweisung. Diese Aktivität verändert den Zustand des BPEL-Prozesses nicht.

**wait** repräsentiert die Unterbrechung des BPEL-Prozesses für eine bestimmte Zeitdauer. Die Zeitdauer kann in einem *for-Element* oder einem *until-Element* angegeben werden. Im *for-Element* wird eine konkrete Zeitdauer spezifiziert. Im *until-Element* wird ein Zeitpunkt spezifiziert, bis zu dem der BPEL-Prozess unterbrochen wird.

### strukturierte Aktivitäten

**sequence** repräsentiert die sequentielle Ausführung der enthaltenen Aktivitäten. Die enthaltenen Aktivitäten werden in der Reihenfolge ausgeführt, in der sie spezifiziert sind.

**if** repräsentiert eine Verzweigung im BPEL-Prozess. Dabei wird mindestens ein *if-Zweig* spezifiziert. Zusätzlich können ein oder mehrere *else-if-Zweige* sowie ein *else-Zweig* spezifiziert werden. Für jeden *if-* bzw. *else-if-Zweig* wird in einem *condition-Element* eine Bedingung für das Betreten des Zweiges angegeben. Beim Betreten eines spezifizierten Zweiges wird die Aktivität ausgeführt, die für diesen Zweig spezifiziert ist. Ist eine solche Aktivität eine strukturierte Aktivität, werden mehrere Aktivitäten im Zweig ausgeführt.

**while** repräsentiert eine Schleife, deren Bedingung für den Schleifeneintritt am Anfang der Schleife überprüft wird. Die Bedingung für den Schleifeneintritt wird in einem *condition-Element* angegeben. Der Schleifenkörper wird durch eine in der while-Aktivität enthaltene Aktivität repräsentiert. Ist die enthaltene Aktivität eine strukturierte Aktivität, werden mehrere Aktivitäten im Schleifenkörper ausgeführt.

**repeatUntil** repräsentiert eine Schleife, deren Bedingung für den Schleifeneintritt am Ende der Schleife überprüft wird. Wie bei der while-Aktivität wird die Bedingung für den Schleifeneintritt in einem *condition-Element* angegeben und der Schleifenkörper wird durch eine in der repeatUntil-Aktivität enthaltene Aktivität repräsentiert. Ist die enthaltene Aktivität eine strukturierte Aktivität, werden mehrere Aktivitäten im Schleifenkörper ausgeführt.

**scope** repräsentiert einen Gültigkeitsbereich für Variablen in einem BPEL-Prozess. In diesem Gültigkeitsbereich können lokale Variablen definiert werden. Diese lokalen Variablen stehen dann nur für die Aktivitäten innerhalb der Scope-Aktivität zur Verfügung. Weiterhin können in einem Gültigkeitsbereich Handler definiert werden.

**forEach** repräsentiert eine for-Schleife, deren Ausführung unter einer bestimmten Bedingung früher abgebrochen werden kann. Für die Laufvariable der for-Schleife wird der Name, der Startwert und der Endwert spezifiziert. Die spezifizierte Laufvariable muss allerdings nicht den Endwert erreichen. Eine bestimmte Bedingung kann die Ausführung der Schleife abbrechen. Eine solche Bedingung wird in einem optionalen *completion-Condition-Element* angegeben. Der Schleifenkörper wird durch eine in der forEach-Aktivität enthaltene Scope-Aktivität repräsentiert.

Die Schleife, die durch eine forEach-Aktivität repräsentiert wird, kann sequentiell oder parallel ausgeführt werden. Wie die Schleife ausgeführt werden soll, wird mit Hilfe des *parallel-Attributs* spezifiziert. Die sequentielle Ausführung der Schleife unterscheidet sich nicht von der Ausführung einer „normalen“ for-Schleife. Bei der parallelen Ausführung der Schleife wird die enthaltene Scope-Aktivität mehrmals kopiert und die Kopien werden dann parallel ausgeführt. Wie oft die Scope-Aktivität kopiert wird, ergibt sich aus der Differenz des Endwerts und dem Startwerts der Laufvariablen.

**pick** wartet auf genau ein Ereignis aus einer Menge von Ereignissen. Ein Ereignis kann dabei das Empfangen einer Nachricht oder eines Signals von einem Zeitzähler sein. In der pick-Aktivität wird der Empfang einer Nachricht in einem *onMessage-Element* spezifiziert. Hier kann analog zur receive-Aktivität entweder die gesamte Nachricht oder nur ein Teil der Nachricht gespeichert werden. Der Empfang eines Signals von einem Zeitzähler wird in der pick-Aktivität in einem *onAlarm-Element* spezifiziert. Dabei wird die Zeidauer des Zeitzählers analog zur wait-Aktivität in einem for-Element oder ei-

nem until-Element spezifiziert. Ist die spezifizierte Zeitdauer überschritten, sendet der Zeitzähler das Signal.

Tritt ein spezifiziertes Ereignis auf, wird die mit dem Ereignis assoziierte Aktivität ausgeführt. Eine Aktivität ist mit einem Ereignis assoziiert, wenn sie in dem zum Ereignis gehörenden onMessage- bzw. onAlarm-Element spezifiziert wird. Ist eine solche Aktivität eine struktuierte Aktivität, werden mehrere Aktivitäten beim Auftreten des Ereignisses ausgeführt.

**flow** repräsentiert die parallele Ausführung der enthaltenen Aktivitäten. Zur Synchronisierung können die enthaltenen Aktivitäten durch Links miteinander verbunden werden. Ein *Link* verbindet genau zwei Aktivitäten miteinander. Die Aktivität am Ende eines Links kann erst ausgeführt werden, wenn die Ausführung der Aktivität am Anfang des Links beendet wurde. Eine Aktivität kann der Anfang oder das Ende mehrerer Links sein. Zyklische Links sind nicht erlaubt.

Eine Aktivität am Anfang eines Links kann eine *transitionCondition* für den betrachteten Link spezifizieren. Diese Bedingung beschreibt den Status des Links. Ist der Status des Links wahr, kann ggf. die Aktivität am Ende des Links ausgeführt werden. Ist keine transitionCondition spezifiziert, wird standardmäßig der Wert *wahr* für den Linkstatus verwendet. Eine Aktivität am Ende von einem oder mehreren Links kann eine *joinCondition* spezifizieren. Diese Bedingung beschreibt, ob die Aktivität ausgeführt werden darf. In der joinCondition kann der Status der eingehenden Links verwendet werden. Wird keine joinCondition spezifiziert, wird standardmäßig die Disjunktion aller eingehenden Links als joinCondition verwendet.

### 2.1.2. Datenmanipulationen in WS-BPEL-Prozessen

Zur Datenmanipulation wird in BPEL-Prozessen standardmäßig die Sprache XPath 1.0 verwendet. In BPEL-Prozessen werden XPath-Ausdrücke in Bedingungen und Zuweisungen verwendet. So wird beispielsweise ein XPath-Ausdruck angegeben, um den Wert zu berechnen, der einer Variablen zugewiesen werden soll. Mit Hilfe eines XPath-Ausdrucks kann also die Adressierung sowie die Berechnung von Daten beschrieben werden. XPath stellt dafür einige Operationen und Funktionen zur Verfügung. Zusätzlich stellt BPEL zwei weitere Funktionen zur Verfügung, die in XPath-Ausdrücken verwendet werden dürfen. Diese beiden Funktionen werden im Folgenden beschrieben.

Die Funktion *getVariableProperty()* liefert das Datum, das in einer Variablen durch eine property benannt wird. Dazu werden der Funktion zwei Parameter übergeben. Der erste Parameter enthält den Namen der Variablen, für die die property definiert ist. Der zweite Parameter enthält den Namen der property. Anhand dieses Namens ermittelt die Funktion den propertyAlias, der für die property definiert ist. Dieser propertyAlias enthält einen XPath-Ausdruck zu Adressierung eines konkreten Datums. Die Funktion wendet

nun den XPath-Ausdruck aus dem `propertyAlias` auf die Variable aus dem ersten Parameter an. Das Resultat des XPath-Ausdrucks wird von der Funktion zurückgegeben.

Die Funktion `doXslTransform()` führt eine XSLTransformation aus. Dabei wird ein Datum von einem bestimmten Datentyp in einen anderen Typ transformiert. Die genaue Beschreibung einer XSLTransformation ist in der Spezifikation [Cla99] enthalten. Der Funktion werden mindestens zwei Parameter übergeben. Der erste Parameter enthält eine Menge von Transformationsregeln. Diese Regeln beschreiben, wie das Datum in den neuen Datentyp transformiert werden soll. Der zweite Parameter enthält das Datum, dessen Datentyp transformiert werden soll. Weiterhin können optionale Parameter angegeben werden, die als Parameter für die XSLTransformation dienen. In BPEL-Prozessen wird die Funktion `doXslTransform()` nur in Zuweisungen verwendet.

### 2.2. XML-Schema

In diesem Kapitel stellen wir *XML-Schema* vor. Ein XML-Schema beschreibt den Typ eines XML-Dokumentes. Wie bereits erwähnt, wird die Extensible Markup Language (XML) für die Beschreibung von hierarchisch strukturierten Daten verwendet. In einem XML-Dokument werden die Daten durch sogenannte XML-Elemente beschrieben. Ein XML-Element kann Kindelemente, Attribute sowie freien Text enthalten.

In BPEL-Prozessen wird XML-Schema verwendet, um die Daten im Prozess zu typisieren. Beispielsweise wird der Datentyp einer Variablen mit Hilfe von XML-Schema definiert. Aber auch der Datentyp von Parts in Nachrichten wird mit Hilfe von XML-Schema definiert. XML-Schema ist sehr komplex und wir beschränken uns deshalb in diesem Kapitel auf die Konzepte, die in der vorliegenden Arbeit benötigt werden. Eine ausführliche Beschreibung ist in der XML-Schema-Spezifikation [TBMM04] und [BM04] enthalten.

#### 2.2.1. Beispiel

Ein XML-Schema beschreibt den *Typ* von XML-Dokumenten. Ein konkretes XML-Dokument, das einem Schema entspricht, wird als *Instanzdokument* bezeichnet. Dies geschieht in Anlehnung an die Objektorientierung, in der ein Typ durch eine Klasse repräsentiert wird. Ein Objekt einer solchen Klasse wird als Instanz bezeichnet.

Der XML-Schema-Namensraum ist per Konvention mit dem Präfix `xsd` assoziiert. Beispielsweise steht das Präfix vor vordefinierten einfachen Typen, wie z.B. `xsd:string`. Aber auch Datenstrukturen können das Präfix enthalten. Dadurch wird eine Zuordnung der jeweiligen Datenstruktur zum XML-Schema-Vokabular ermöglicht.

Datenstrukturen werden in XML-Schema als *Element* oder *Attribut* dargestellt. Ein Element kann Kindelemente sowie Attribute enthalten. Attribute dürfen keine Kindelemente haben und müssen von einem einfachen Typ sein. In Abb. 2.1 sind beispielsweise vier Elemente in den Zeilen 5 – 8 deklariert. Diese Elemente besitzen jeweils ein Attribut

```

1  <xsd:schema xmlns:xsd='http://www.w3.org/2001/XMLSchema'>
2    <xsd:element name='Lieferadresse' type='Adresse' />
3    <xsd:complexType name='Adresse'>
4      <xsd:sequence>
5        <xsd:element name='Name' type='xsd:string' />
6        <xsd:element name='Strasse' type='xsd:string' />
7        <xsd:element name='Ort' type='xsd:string' />
8        <xsd:element name='PLZ' type='xsd:unsignedInt' />
9      </xsd:sequence>
10   </xsd:complexType>
11 </xsd:schema>

```

**Abbildung 2.1.:** Das Schema einer Adresse. Der Typ `Adresse` hat 4 Kindelemente mit jeweils einem Attribut `name` und `type`. Die Instanz des Schemas enthält ein Element `Lieferadresse` vom Typ `Adresse`.

`name` und ein Attribut `type`. Im Folgenden beschreiben wir das Beispiel aus Abb. 2.1 genauer.

Abbildung 2.1 definiert ein XML-Schema für eine Adresse. In Zeile 2 wird festgelegt, dass eine Instanz des Schemas ein Element `Lieferadresse` vom Typ `Adresse` enthält. Der Typ `Adresse` wird in den Zeilen 3 – 10 definiert. Wir können dort sehen, dass ein Element vom Typ `Adresse` 4 Kindelemente mit den Namen `Name`, `Strasse`, `Ort` und `PLZ` hat. Die ersten drei Kindelemente enthalten jeweils eine Zeichenkette, das vierte Kindelement enthält eine Zahl. Für ein Element vom Typ `Adresse` fordert das Schema genau die 4 Kindelemente mit den entsprechenden Namen und Typen. So wäre ein Element vom Typ `Adresse` ohne das Kindelement `PLZ` unzulässig. Mit Hilfe des Schlüsselworts `sequence` werden die deklarierten Elemente zu einer Gruppe zusammengefasst.

Die Schema-Definition aus Abb. 2.1 definiert den Typ `Adresse` unter Verwendung globaler und lokaler Elemente. *Globale Elemente* sind direkte Kindelemente des Schema-Elements (`xsd:schema`). Die Namen globaler Elemente müssen eindeutig innerhalb des Schema-Elements sein. In Abb. 2.1 ist `Lieferadresse` demnach ein globales Element. Die Elemente `Name`, `Strasse`, `Ort` und `PLZ` sind dagegen *lokale Elemente* eines Elements vom Typ `Adresse` und damit keine direkten Kindelemente des Schema-Elements. In Abb. 2.1 sind die Elemente `Name`, `Strasse`, `Ort` und `PLZ` lokale Elemente des Elements `Lieferadresse`.

Da alle Elemente im Schema aus Abb. 2.1 einen eindeutigen Namen haben, könnte das Schema auch unter Verwendung von ausschließlich globalen Elementen definiert werden. Die Verwendung lokaler Elemente bietet aber einen großen Vorteil. Beispielsweise könnte ein Buch die Kindelemente `Titel` und `Autor` enthalten. Um z.B. auch akademische Titel zu berücksichtigen, enthalte das Element `Autor` die Kindelemente `Titel` und `Name`. Bei der Verwendung lokaler Elemente kann sowohl das Element, das den Titel des Buches

beschreibt, als auch das Element, das den Titel des Autors beschreibt, den Namen Titel besitzen. Unter Verwendung globaler Elemente wäre das nicht möglich und der Name der Titlelemente müsste genauer spezifiziert werden.

Ein Instanzdokument zum Schema aus Abb. 2.1 könnte ein Element `Lieferadresse` enthalten, wie in Abb. 2.2 dargestellt. Hier hat das Element `Lieferadresse` alle im Schema geforderten Kindelemente. Der Wert eines Kindelementes ist der Inhalt des Kindelements. Beispielsweise wird der Wert des Elements `Name` als Inhalt zwischen `<Name>` und `</Name>` angegeben.

```
1 <Lieferadresse >
2     <Name> Liese von der Wiese </Name>
3     <Strasse> An der Wiese 3 </Strasse>
4     <Ort> Wiesenstadt </Ort>
5     <PLZ> 23632 </PLZ>
6 </Lieferadresse>
```

**Abbildung 2.2.:** Ein Element vom Typ `Adresse`. Das Element `Lieferadresse` enthält alle Kindelemente, die durch die Typvorgabe gefordert werden.

### Namensräume

Einem Element, Attribut sowie einem Typ kann ein *Namensraum* zugeordnet werden. Für die Zuordnung eines Namensraumes steht das Attribut `xmlns` zur Verfügung. Einem Namensraum kann zusätzlich ein *Präfix* zugeordnet werden. Der Präfix repräsentiert dann den Namensraum. Damit kann einem Element, Attribut sowie einem Typ der Namensraum auch über das entsprechende Präfix zugeordnet werden.

In Abb. 2.1 haben wir bereits den XML-Schema-Namensraum kennengelernt. Hier wird in der ersten Zeile der XML-Schema-Namensraum `http://www.w3.org/2001/XMLSchema` mit Hilfe des Attributs `xmlns` referenziert. Zusätzlich wird diesem Namensraum das Präfix `xsd` zugeordnet. Dazu wird der Name des `xmlns`-Attributs um einen Doppelpunkt und den Namen des Präfixes erweitert. Möchten wir nun den Typ `xsd:string` aus dem XML-Schema-Namensraum verwenden, können wir das Präfix gefolgt von einem Doppelpunkt vor dem Typ `xsd:string` angeben. Bei den Typangaben der Elemente `Name`, `Strasse` und `Ort` nutzen wir diese Möglichkeit.

Möchten wir die Elemente im Instanzdokument aus Abb. 2.2 einem Namensraum zuordnen, müssen wir zunächst den Namensraum und ggf. ein Präfix für diesen Namensraum angeben. Beispielsweise könnten wir dazu die erste Zeile in Abb. 2.2 durch folgende Zeile ersetzen:

```
<Lieferadresse xmlns:adr='http://www.beispiel.de/Adressen'>
```

Damit ordnen wir das Element `Lieferadresse` dem Namensraum `http://www.beispiel.de/Adressen` zu. Weiterhin definieren wir für diesen Namensraum das Präfix `adr`. Da ein Namensraum immer an Kindelemente vererbt wird, sind damit die Elemente `Name`, `Strasse`, `Ort` und `PLZ` automatisch auch diesem Namensraum zugeordnet. Allerdings können wir die Zuordnung zu diesem Namensraum auch explizit angeben. Dazu können wir nun das Präfix verwenden. So kann beispielsweise das Element `Name` explizit dem Namensraum durch folgende Zeile zugeordnet werden:

```
<adr:name> Liese von der Wiese </adr:name>
```

XML-Schema unterscheidet zwischen Deklarationen und Definitionen. Bei einer Deklaration wird ein Element bzw. Attribut beschrieben, das im Instanzdokument vorkommt. Eine Definition beschreibt einen Typ. Im Folgenden beschreiben wir die Deklaration von Elementen bzw. Attributen sowie die Typdefinition in XML-Schema genauer.

### 2.2.2. Deklarationen

Die *Deklaration* von Elementen haben wir bereits in Abb. 2.1 kennen gelernt. Dort werden fünf Elemente mit den Namen `Lieferadresse`, `Name`, `Strasse`, `Ort` und `PLZ` deklariert. Wie wir sehen können, wird ein Element mit Hilfe des Schlüsselworts `element` deklariert. Ein Attribut wird wie ein Element deklariert, allerdings wird hier das Schlüsselwort `attribute` verwendet.

Bei der Deklaration eines Elements wird grundsätzlich ein Name mit einem Typ assoziiert. Beispielsweise wird in Abb. 2.1 der Name `PLZ` mit dem Typ `unsignedInt` assoziiert. Es besteht aber auch die Möglichkeit bei einer Elementdeklaration ein globales Element zu referenzieren. Eine *Referenz* wird mit Hilfe des Attributs `ref` angegeben. Beispielsweise wird in Zeile 6 der Abb. 2.3 ein Element unter Angabe einer Referenz auf das globale Element `Kommentar` deklariert. In einem Instanzdokument zum XML-Schema aus Abb. 2.3 müssen nun zwei Elemente mit Namen `Kommentar` enthalten sein. Eines dieser Elemente ist global, das andere ist ein lokales Element von `Kunde`.

Der Wert eines deklarierten Elements bzw. Attributs wird im Instanzdokument angegeben. In Abb. 2.2 haben wir bereits gesehen, wie der Wert eines Elements angegeben wird. Beispielsweise wird hier dem Element `Name` der Wert `Liese von der Wiese` zugewiesen. Der Wert eines Attributs wird wie der Wert eines Elements angegeben.

### Konstanten

Ein Element bzw. Attribut kann im XML-Schema als *Konstante* deklariert werden. Dabei wird der Konstantenwert bereits bei der Deklaration im XML-Schema angegeben. Für die Angabe des Konstantenwert steht das Attribut `fixed` zur Verfügung. Beispielsweise können wir in Abb. 2.1 das Element `PLZ` als Konstante deklarieren, indem wir Zeile 8 durch folgende Zeile ersetzen:

```
<xsd:element name='PLZ' type='xsd:unsignedInt' fixed='12555' />
```

```
1 <xsd:schema xmlns:xsd='http://www.w3.org/2001/XMLSchema'>
2   <xsd:element name='Kunde' type='Person'/>
3   <xsd:element name='Kommentar' type='xsd:string'/>
4   <xsd:complexType name='Person'>
5     <xsd:sequence>
6       <xsd:element ref='Kommentar'>
7       <xsd:element name='Lieferadresse' type='Adresse'/>
8       <xsd:element name='Kdnummer' type='xsd:unsignedInt'/>
9     </xsd:sequence>
10  </xsd:complexType>
11 </xsd:schema>
```

**Abbildung 2.3.:** Bei der Elementdeklaration kann ein globales Element referenziert werden. In Zeile 6 wird ein Element mit Hilfe einer Referenz auf das globale Element `Kommentar` deklariert. Im Instanzdokument müssen nun zwei Elemente mit Namen `Kommentar` vorkommen. Ein Element ist global im Instanzdokument. Das andere Element ist ein lokales Element von `Kunde`.

### Häufigkeitsbeschränkungen

Bei der Deklaration von Elementen bzw. Attributen kann eine *Häufigkeitsbeschränkung* angegeben werden. Diese Häufigkeitsbeschränkung gibt an, wie oft das Element bzw. Attribut im Instanzdokument auftreten darf. Die Häufigkeitsbeschränkung eines Elements wird bei der Deklaration mit Hilfe der Attribute `minOccurs` und `maxOccurs` angegeben. Die minimale Anzahl eines Elements im Instanzdokument wird durch das Attribut `maxOccurs` angegeben. Die maximale Anzahl wird durch das Attribut `minOccurs` angegeben. Der Vorgabewert dieser Attribute ist der Wert 1. Wird also keine explizite Häufigkeitsbeschränkung für ein Element im XML-Schema angegeben, muss das Element genau einmal im Instanzdokument auftreten.

Um ein Element `Wohnadresse` vom Typ `Adresse` aus Abb. 2.1 zu deklarieren, das im Instanzdokument nicht zwingend auftreten muss, aber maximal fünf mal auftreten darf, können wir das Element folgendermaßen deklarieren:

```
<xsd:element name='Wohnadresse' type='Adresse' minOccurs='0'
maxOccurs='5'/>
```

Da die Vorgabewerte der Attribute `minOccurs` und `maxOccurs` 1 ist, müssen wir eine Beschränkung auf maximal ein bzw. minimal ein Element nicht angeben. So können wir ein Element `Zweitwohnsitz`, das im Instanzdokument nicht auftreten muss, aber maximal einmal auftreten darf, wie folgt deklarieren:

```
<xsd:element name='Zweitwohnsitz' type='Adresse' minOccurs='0'/>
```

Die Häufigkeitsbeschränkung eines Attributs wird bei der Deklaration mit Hilfe des Attributs `use` angegeben. Diesem Attribut darf einer der drei Werte `required`, `optional`

oder `prohibited` zugewiesen werden. Der Vorgabewert des Attributs `use` ist der Wert `required`. Der Wert `required` gibt an, dass das Attribut genau einmal auftreten darf. Der Wert `optional` gibt an, dass das Attribut im Instanzdokument nicht zwingend auftreten muss, aber maximal einmal auftreten darf. So können wir beispielsweise ein Attribut `Bestelldatum`, das im Instanzdokument nicht oder maximal einmal auftreten darf, folgendermaßen im XML-Schema deklarieren:

```
<xsd:attribute name='Bestelldatum' type='xsd:date' use='optional' />
```

Der Wert `prohibited` für das `use`-Attribut gibt an, dass das Attribut im Instanzdokument nicht auftreten darf.

Neben der vorgestellten Technik zur Deklaration von Elementen bzw. Attributen erlaubt XML-Schema die spezielle Technik zur Deklaration mit den Schlüsselwörtern `any` und `anyAttribute`. Bei der Deklaration eines Elements bzw. Attributs mit `any` bzw. `anyAttribute` wird jedes wohlgeformte<sup>3</sup> XML zugelassen. Diese Technik wird verwendet, wenn der Name und Typ eines Elements bzw. Attributs nicht näher spezifiziert werden soll, aber trotzdem einige Anforderungen für das Element bzw. Attribut gelten sollen. So kann für ein solches Element bzw. Attribut beispielsweise der Namensraum oder eine Häufigkeitsbeschränkung im XML-Schema vorgegeben werden. Beispielsweise kann ein beliebiges Element, das im Namensraum `http://www.beispiel.de/BspNamensraum` enthalten sein muss, folgendermaßen im XML-Schema deklariert werden:

```
<xsd:any xmlns='http://www.beispiel.de/BspNamensraum' />
```

### 2.2.3. Typdefinitionen

Wir haben bereits gesehen, dass einem Element bzw. Attribut bei seiner Deklaration ein Typ zugeordnet wird. Die Definition von solchen Typen möchten wir nun genauer vorstellen. XML-Schema unterscheidet dabei zwischen *einfachen* und *komplexen Typen*. Während Elemente sowohl einfache als auch komplexe Typen haben dürfen, darf ein Attribut nur einfache Typen haben.

Die speziellen Eigenschaften einfacher und komplexer Typen werden im Folgenden beschrieben. Sowohl einfache als auch komplexe Typen sind vom vordefinierten Typ `xsd:anyType` abgeleitet. Dieser generelle Typ kann deshalb jeden beliebigen anderen Typ repräsentieren. Für die Repräsentation eines beliebigen einfachen Typs gibt es zusätzlich den Typ `xsd:anySimpleType`, der eine Spezialisierung des Typs `xsd:anyType` darstellt. Die Typen `xsd:anyType` und `xsd:anySimpleType` dürfen in der Typangabe einer Element- bzw. Attributdeklaration verwendet werden.

<sup>3</sup>Ein XML-Dokument ist wohlgeformt, wenn es keine XML-Regel verletzt. In den XML-Regeln wird beispielsweise die Syntax für XML-Elemente festgelegt. Aber auch die Namens eindeutigkeit wird in den XML-Regeln gesichert. Weitere Informationen sind in der XML-Spezifikation [BPSM<sup>+</sup>04] enthalten.

## Einfache Typen

In XML-Schema bilden die sogenannten atomaren Typen, Listen- und Vereinigungstypen die Menge der einfachen Typen. Für die Definition einfacher Typen steht das Schlüsselwort `simpleType` zur Verfügung. Die Definition eines einfachen Typs kann mit folgender Zeile eingeleitet werden:

```
<xsd:simpleType name='...'>
```

Die *atomaren Typen* im XML-Schema sind unteilbar. Das bedeutet, dass der Wert eines atomaren Typs keine eigenständigen Teile enthält. Beispielsweise haben die einzelnen Buchstaben einer Zeichenkette allein keine Bedeutung. Erst der gesamten Zeichenkette kann eine Bedeutung wie z.B. Strasse oder Ort zugeordnet werden. Zunächst bestehen die atomaren Typen aus den vordefinierten Datentypen in XML-Schema, z.B. `xsd:string`, `xsd:integer`, `xsd:boolean`, `xsd:time`. Eine Liste aller vordefinierten Datentypen ist in der XML-Schema-Spezifikation [BM04] enthalten. Weitere atomare Typen können durch Ableitung von bestehenden atomaren Typen definiert werden. Die Ableitung neuer Typen stellen wir später genauer vor.

Der *Listentyp* ermöglicht das Zusammenfassen mehrerer Werte eines atomaren Typs zu einer Liste. Ein Listentyp kann also geteilt werden, wobei jeder Teil eine eigene Bedeutung hat. Möchten wir beispielsweise verschiedene Telefonnummern in einer Liste zusammenfassen, können wir für die einzelnen Telefonnummern den atomaren Typ `xsd:unsignedInt` verwenden. Eine Liste von Telefonnummern können wir dann als Listentyp `Telefonliste` wie folgt definieren:

```
<xsd:simpleType name='Telefonliste'>
  <xsd:list itemType='xsd:unsignedInt' />
</xsd:simpleType>
```

Das Schlüsselwort `list` zeichnet dabei den einfachen Typ `Telefonliste` als Listentyp aus. Das Attribut `itemType` bestimmt den Typ der einzelnen Listenelemente. Wie bereits erwähnt muss dieser Typ ein atomarer Typ sein. Das Element eines Listentyps ist eine Folge einzelner Werte, die durch Leerzeichen voneinander getrennt werden. Beispielsweise kann ein Element vom Typ `Telefonliste` folgendermaßen aussehen:

```
<meineTelefonliste>
  '2154638 01728634444 00128506629376520'
</meineTelefonliste>
```

Für einen Listentyp kann die Länge der Liste spezifiziert werden. Dabei ist zu beachten, dass die Länge in XML-Schema bei 1 beginnt. Beispielsweise können wir die Typdefinition für den Typ `Telefonliste` um die Zeile `<xsd:length value=3>` erweitern. Damit fordern wir, dass ein Element vom Typ `Telefonliste` genau 3 Telefonnummern enthält. Neben `length` stehen die Schlüsselwörter `minLength` und `maxLength` zur Verfügung, die die minimale bzw. maximale Länge einer Liste bestimmen.

*Vereinigungstypen* ermöglichen das Zusammenfassen mehrerer einfacher Typen zu einem Typ. Dabei können *alle* einfachen Typen vereinigt werden. Die Typdefinition eines Vereinigungstypen wird durch das Schlüsselwort `union` ausgezeichnet. Die Typen, die vereinigt werden sollen, werden im Attribut `memberTypes` angegeben. Im folgenden Beispiel vereinigen wir die einfachen Typen `PLZ` und `Ort` zu einem Typ `Bundesland`:

```
<xsd:simpleType name='Bundesland'>
  <xsd:union memberTypes='PLZ Ort' />
</xsd:simpleType>
```

## Komplexe Typen

*Komplexe Typen* zeichnen sich dadurch aus, dass Elemente dieser Typen Kindelemente und Attribute enthalten dürfen. Für die Definition eines komplexen Typs steht das Schlüsselwort `complexType` zu Verfügung. In Abb. 2.1 haben wir bereits den komplexen Typ `Adresse` definiert. Eine Adresse besteht aus den Kindelementen `Name`, `Strasse`, `Ort` und `PLZ`. Abbildung 2.4 zeigt die Definition eines weiteren komplexen Typs `Buchbestellung`. In Zeile 5 wird ein Element `Lieferadresse` deklariert, welches wiederum einen komplexen Typ hat. Zeile 7 deklariert ein Attribut `Bestelldatum` für ein Element vom Typ `Buchbestellung`.

```
1 <xsd:complexType name='Buchbestellung'>
2   <xsd:sequence>
3     <xsd:element name='ISBN' type='xsd:unsignedInt'>
4     <xsd:element name='Anzahl' type='xsd:unsignedInt'>
5     <xsd:element name='Lieferadresse' type='Adresse'>
6   </xsd:sequence>
7   <xsd:attribute name='Bestelldatum' type='xsd:date'>
8 </xsd:complexType>
```

Abbildung 2.4.: Das Schema einer Buchbestellung

Bei der Definition eines komplexen Typs gibt es ein `mixed`-Attribut, das angibt ob Text zwischen den Kindelementen oder Attributen auftreten darf. Wird dem `mixed`-Attribut der Wert `false` zugewiesen, darf kein Text zwischen den Kindelementen oder Attributen auftreten. Wird dem `mixed`-Attribut der Wert `true` zugewiesen, darf Text zwischen den

Kinderelementen oder Attributen auftreten. Möchten wir beispielsweise in einer Adresse Text zwischen den einzelnen Elementen zulassen, können wir Zeile 3 aus Abb. 2.1 durch folgende Zeile ersetzen:

```
<xsd:complexType name='Adresse' mixed='true'>
```

Ein Instanzdokument kann dann im Gegensatz zu Abb. 2.2 beispielsweise so aussehen:

```
<Lieferadresse >
  Hier werden die Daten einer Person angegeben.
  <Name> Liese von der Wiese </Name>
  <Strasse> An der Wiese 3 </Strasse>
  <Ort> Wiesenstadt </Ort>
  <PLZ> 23632 </PLZ>
</Lieferadresse>
```

Der Vorgabewert des `mixed`-Attributs ist *false*. Ist also kein expliziter Wert im XML-Schema angegeben, wird der Wert *false* verwendet.

### Anonyme Typen

Die Typdefinitionen aus Abb. 2.1 und Abb. 2.4 sind Typdefinitionen bei denen der Typ mit einem Namen verbunden ist. Es ist aber auch möglich, einen Typ ohne Namen zu definieren. Solche Typen werden als *anonyme Typen* bezeichnet. Anonyme Typen können einfach oder komplex sein. Sie werden immer dann verwendet, wenn nur ein Element des Typs deklariert werden soll.

Bei der Deklaration von Elementen mit einem anonymen Typ fehlt das `type`-Attribut. Dafür wird ein anonymer Typ innerhalb des Elements definiert. Möchten wir beispielsweise für das Element `Lieferadresse` aus Abb. 2.4 einen anonymen Typ definieren, können wir Zeile 5 aus Abb. 2.4 durch die folgenden Zeilen ersetzen:

```
<xsd:element name='Lieferadresse'>
  <xsd:complexType>
    <xsd:element name='Name' type='xsd:string' />
    <xsd:element name='Strasse' type='xsd:string' />
    <xsd:element name='Ort' type='xsd:string' />
    <xsd:element name='PLZ' type='xsd:unsignedInt' />
  </xsd:complexType>
</xsd:element>
```

## Ableitung neuer Typen

Sowohl einfache als auch komplexe Typen können durch *Ableitung* von bestehenden Typen definiert werden. Dabei können einfache Typen nur aus einfachen Typen abgeleitet werden. Komplexe Typen können sowohl aus einfachen als auch aus komplexen Typen abgeleitet werden. Bei der Ableitung eines neuen Typs wird außerdem zwischen *Einschränkung* und *Erweiterung* des bestehenden Typs unterschieden. Einfache Typen dürfen nur einschränken. Die Erweiterung eines einfachen Typs ist nicht erlaubt, da nicht mehr sichergestellt werden kann, dass der resultierende Typ ein einfacher Typ ist. Bei komplexen Typen ist sowohl Einschränkung als auch Erweiterung erlaubt. Allerdings darf bei der Ableitung eines komplexen Typs aus einem einfachen Typ keine Einschränkung vorgenommen werden, da wieder ein einfacher Typ entstehen würde.

**Einfache Typen** Ein neuer einfacher Typ kann aus einem bestehenden einfachen Typ abgeleitet werden, indem Einschränkungen an dem bestehenden Typ vorgenommen werden. Um eine solche Ableitung zu beschreiben, wird bei der Definition des neuen Typs das Schlüsselwort `restriction` verwendet. In einem Attribut `base` wird der bestehende Typ angegeben, von dem abgeleitet wird. Möchten wir beispielsweise einen Typ definieren, der alle 5-stellige Postleitzahlen beschreibt, können wir diesen Typ von dem vordefinierten Typ `xsd:integer` ableiten:

```
<xsd:simpleType name='PLZ'>
  <xsd:restriction base='xsd:integer'>
    <xsd:min-inclusive value='10000' />
    <xsd:max-inclusive value='99999' />
  </xsd:restriction>
</xsd:simpleType>
```

Durch die Werte in `min-inclusive` und `max-inclusive` wird der Wertebereich des Typs `xsd:integer` beschränkt. Für die Einschränkung kann aber auch `min-exclusive` oder `max-exclusive` verwendet werden. Die Anwendung von `min-`, `max-inclusive` bzw. `-exclusive` ist verständlicherweise auf Zahlen, Zeit und Tagesdaten beschränkt.

Neben der vorgestellten Einschränkung für Zahlen, Zeit und Tagesdaten, gibt es Einschränkungen, die für alle vordefinierten Datentypen erlaubt sind. So können alle vordefinierten Datentypen durch Angabe eines regulären Ausdrucks oder durch Aufzählung der erlaubten Werte eingeschränkt werden.

Ein regulärer Ausdruck wird mit Hilfe des Schlüsselworts `pattern` angegeben. Die Sprache für reguläre Ausdrücke in XML-Schema unterstützt Unicode. Eine Übersicht über die Sprache befindet sich im Anhang F der XML-Schema-Spezifikation [BM04]. Das folgende Beispiel schränkt den Typ `xsd:string` so ein, dass die Schulnoten  $1^+$ ,  $1^-$ ,  $2^+$ ,  $2, \dots$  durch den neuen Typ `Schulnoten` dargestellt werden.

```
<xsd:simpleType name='Schulnoten'>
  <xsd:restriction base='xsd:string'>
    <xsd:pattern value='[1 - 6][+-]?'>/>
  </xsd:restriction>
</xsd:simpleType>
```

Einen neuen Typ durch Aufzählung der zulässigen Werte zu definieren, ist für jeden vordefinierten Datentyp außer boolean erlaubt. Für die Aufzählung der einzelnen Werte, wird das Schlüsselwort `enumeration` innerhalb einer Typableitung verwendet:

```
<xsd:simpleType name='Schulnoten'>
  <xsd:restriction base='xsd:string'>
    <xsd:enumeration value='1+'>/>
    <xsd:enumeration value='1'>/>
    <xsd:enumeration value='1-'>/>
    <xsd:enumeration value='2+'>/>
    ...
  </xsd:restriction>
</xsd:simpleType>
```

**Komplexe Typen** Neue komplexe Typen können sowohl aus einfachen als auch aus komplexen Typen abgeleitet werden. So können komplexe Typen durch Erweiterung einfacher Typen entstehen, aber auch durch Einschränkung oder Erweiterung bestehender komplexer Typen.

Ein einfacher Typ wird zu einem komplexen Typ erweitert, indem Attribute hinzugefügt werden. Somit hat ein Element des neuen komplexen Typs immer noch den Inhalt des alten einfachen Typs. Allerdings kann das Element Attribute enthalten, was einem Element vom einfachen Typ nicht erlaubt ist. Möchten wir beispielsweise für ein Element mit dem oben definierten einfachen Typ PLZ ein Land angeben, in dem die Postleitzahl gilt, können wir dies in einem Attribut des Elements realisieren. Dazu müssen wir aber den einfachen Typ PLZ zu einem komplexen Typ erweitern. Der komplexe Typ `PLZmitLand` im folgenden Beispiel repräsentiert eine solche Erweiterung:

```
<xsd:complexType name='PLZmitLand'>
  <xsd:extension base='xsd:PLZ'>
    <xsd:attribute name='Land' type='xsd:string'>/>
  </xsd:extension>
</xsd:complexType>
```

Wie wir sehen, wird eine Erweiterung durch das Schlüsselwort `extension` beschrieben. Das Attribut `base` enthält wieder den bestehenden Typ, von dem abgeleitet wird.

Die Einschränkung eines komplexen Typs zu einem anderen komplexen Typ ist der Einschränkung eines einfachen Typs sehr ähnlich. Dementsprechend repräsentiert der eingeschränkte komplexe Typ eine Teilmenge des Basistyps. Die Einschränkung eines komplexen Typs wird ebenfalls durch das Schlüsselwort `restriction` beschrieben.

Folgende Einschränkungen an einen komplexen Typ können vorgenommen werden:

- Kindelemente bzw. Attribute können ausgeschlossen werden,
- Häufigkeitsbeschränkungen für Kindelemente bzw. Attribute können eingeschränkt werden,
- der Typ von Kindelementen bzw. Attributen kann genauer spezifiziert werden,
- variable Werte können zu konstanten Werten eingeschränkt werden.

Kindelemente bzw. Attribute aus dem Basistyp können ausgeschlossen werden, indem sie nicht mehr im abgeleiteten komplexen Typ aufgezählt werden. Beispielsweise können wir den Typ `Buchbestellung` in Abb. 2.4 durch Ausschließen des Elements `Anzahl` sowie des Attributs `Bestelldatum` folgendermaßen einschränken:

```
<xsd:complexType name='eingeschränkteBestellung'>
  <xsd:restriction base='Buchbestellung'>
    <xsd:sequence>
      <xsd:element name='ISBN' type='xsd:unsignedInt'>
        <xsd:element name='Lieferadresse' type='Adresse'>
      </xsd:sequence>
    </xsd:restriction>
  </xsd:complexType>
```

Weiterhin können wir Kindelemente bzw. Attribute ausschließen, indem eine entsprechende Häufigkeitsbeschränkung angegeben wird. So kann das Element `Anzahl` im Typ `Buchbestellung` aus Abb. 2.4 auch durch die Angabe der folgenden Deklaration ausgeschlossen werden:

```
<xsd:element name='Anzahl' type='xsd:unsignedInt' minOccurs='0'
maxOccurs='0'>
```

Eine Häufigkeitsbeschränkung für ein Element aus dem Basistyp wird im abgeleiteten Typ eingeschränkt, wenn beispielsweise dem Attribut `minOccurs` eine größere Zahl als im Basistyp zugewiesen wird. Dem Attribut `maxOccurs` kann bei der Einschränkung wiederum eine kleinere Zahl als im Basistyp zugewiesen werden. So können wir beispielsweise die Häufigkeitsbeschränkung für das Element `Wohnadresse` aus Kapitel 2.2.2 folgendermaßen einschränken:

```
<xsd:element name='Wohnadresse' type='Adresse' minOccurs='1'
maxOccurs='1'/>
```

Um den Typ eines Kindelements bzw. Attributs genauer zu spezifizieren, werden eingeschränkte Typen verwendet. Nehmen wir an, dass ein Kindelement im Basistyp den Typ `Buchbestellung` hat. Für dieses Kindelement können wir dann im abgeleiteten Typ den Typ `eingeschränkteBuchbestellung` verwenden, um den Typ genauer zu spezifizieren. Nehmen wir andererseits an, dass ein Kindelement oder Attribut im Basistyp einen einfachen Typ hat, der durch einen regulären Ausdruck oder einen Aufzählungstyp gegeben ist. Dann kann auch der reguläre Ausdruck bzw. die Aufzählungswerte eingeschränkt werden. Beispielsweise kann der reguläre Ausdruck `[1 - 6][+-]?` zu dem regulären Ausdruck `[1 - 5][+-]?` eingeschränkt werden. Bei der Einschränkung von Aufzählungswerten wird mindestens ein Wert nicht aufgezählt.

Schließlich kann bei der Ableitung eines komplexen Typs aus einem anderen komplexen Typ durch Einschränkung ein konstanter Wert für ein Kindelement bzw. Attribut angegeben werden, für den im Basistyp ein variabler Wert definiert ist. So können wir beispielsweise dem Kindelement `ISBN` im Typ `Buchbestellung` aus Abb. 2.4 einen konstanten Wert im eingeschränkten Typ `eingeschränkteBuchbestellung` zuweisen. Die Typdefinition `eingeschränkteBuchbestellung` kann dazu die folgende Elementdeklaration enthalten:

```
<xsd:element name='ISBN' type='xsd:unsignedInt'  
fixed='9783442153749'>
```

Die Ableitung eines komplexen Typs aus einem anderen komplexen Typ durch Erweiterung ist der Ableitung durch Erweiterung eines einfachen Typs sehr ähnlich. Wir können weitere Kindelemente sowie Attribute für den Typ hinzufügen. Wie bei der Erweiterung eines einfachen Typs wird die Erweiterung eines komplexen Typs durch das Schlüsselwort `extension` beschrieben.

Möchten wir den komplexen Typ `Buchbestellung` aus Abb. 2.4 um ein Element `Preis` und ein Attribut `Bezahlt` erweitern, können wir beispielsweise folgenden Typ definieren:

```
<xsd:complexType name='erweiterteBestellung'>  
  <xsd:extension base='Buchbestellung'>  
    <xsd:sequence>  
      <xsd:element name='Preis' type='xsd:unsignedInt'>  
    </xsd:sequence>  
    <xsd:attribute name='Bezahlt' type='xsd:boolean'>  
  </xsd:extension>  
</xsd:complexType>
```

## 2.3. XPath

In diesem Kapitel beschreiben wir die Ausdruckssprache XPath 1.0. In BPEL-Prozessen werden XPath-Ausdrücke verwendet, um Datenmanipulationen zu beschreiben. Die Sprache XPath ist dabei besonders geeignet, weil die Daten in BPEL-Prozessen meist in XML-Dokumenten enthalten sind. XPath bietet nun eine Möglichkeit, um Teile eines solchen XML-Dokumentes zu adressieren. Aber auch Werteberechnungen auf Grundlage der Daten eines BPEL-Prozesses können mit Hilfe von XPath durchgeführt werden.

XPath modelliert ein XML-Dokument als Baum. Die Knoten des Baums repräsentieren XML-Elemente und Attribute sowie Zeichenketten, Kommentare, Namensräume und Processing Instructions innerhalb des XML-Dokumentes. Die Elter-Kind-Beziehungen im Baum repräsentieren die Beziehungen der XML-Elemente, Attribute, Kommentare usw. im XML-Dokument. Zusätzlich unterstützt XPath XML-Namensräume [BHL99] in vollem Umfang.

Ein XPath-Ausdruck beschreibt einerseits die Adressierung von Daten in einem XML-Dokument. Dafür stellt XPath sogenannte Lokalisierungspfade zur Verfügung. Andererseits können auch Datenberechnungen in einem XPath-Ausdruck beschrieben werden. Dabei unterstützt XPath die vier Datentypen *boolean*, *numbers*, *string* und *node-set*. Diese Datentypen bieten die Grundlage für Operationen und Funktionen, die von XPath zur Verfügung gestellt werden. Im Folgenden beschreiben wir zunächst die vier Datentypen in XPath sowie die Operationen, die für diese Datentypen definiert sind. Danach stellen wir die Lokalisierungspfade vor, die zur Adressierung von Daten verwendet werden. Schließlich geben wir eine Übersicht über die Funktionen, die in der Grundbibliothek von XPath zur Verfügung gestellt werden. Eine vollständige Beschreibung von XPath ist in der XPath-Spezifikation [CD99] enthalten.

### 2.3.1. Boolsche Werte

Wahrheitswerte in XPath werden durch den Datentyp *boolean* repräsentiert. Ein boolescher Wert kann einen der beiden Wahrheitswerte *true* und *false* annehmen. In XPath sind für boolsche Werte die logischen Operationen Disjunktion („*or*“) und Konjunktion („*and*“) sowie die Vergleichsoperationen Gleich („*=*“) und Ungleich („*!=*“) definiert. Die Prioritäten der Operationen entsprechen der angegebenen Reihenfolge. Die Operation „*or*“ hat dabei die geringste Priorität.

### 2.3.2. Zahlen

Alle Zahlen in XPath werden durch den Datentyp *numbers* repräsentiert. XPath unterscheidet also nicht zwischen ganzen Zahlen und Dezimalzahlen. Der Typ *numbers* in XPath repräsentiert eine Gleitkommazahl nach dem Standard IEEE 754. Eine Zahl in XPath kann jeden 64-Bit Wert doppelter Genauigkeit des Formates IEEE 754 annehmen. In XPath ist eine Exponentendarstellung für Gleitkommazahlen nach Standard

IEEE 754 unzulässig. Die lexikalischen Regeln in XPath schreiben eine Dezimalzahldarstellung vor. Die Dezimalzahlen, die nach IEEE 754 repräsentiert werden können, liegen im Bereich zwischen  $2,225 \cdot 10^{-308}$  und  $1,798 \cdot 10^{308}$ .

Für Sonderfälle stellt der Standard IEEE 754 spezielle Werte zur Verfügung. Diese speziellen Werte entstehen, wenn eine Rechenoperation einen Überlauf produziert oder das Rechenergebnis außerhalb des darstellbaren Bereichs liegt. Der spezielle Wert *NaN* („Not a Number“) wird als Darstellung für explizit unmögliche Zahlen verwendet. Der Wert *NaN* entsteht z.B. nach Anwendung einer verbotenen Operation. Zu große Ergebnisse werden durch die speziellen Werte  $\infty$  und  $-\infty$  dargestellt. Schließlich muss noch erwähnt werden, dass die Zahl 0 in zwei Darstellungen als +0 und -0 existiert. Beide werden jedoch rechnerisch als identisch betrachtet.

Für Zahlen in XPath sind die numerischen Operationen Addition („+“), Subtraktion („-“), Multiplikation („\*“), Division („:“), Modulo („mod“) sowie die Vergleichsoperationen Gleich („=“), Ungleich („!=“), Kleiner („<“), Kleiner oder Gleich („<=“), Größer („>“) und Größer oder Gleich („>=“) definiert. Die Prioritäten der numerischen Operationen entsprechen den Prioritäten aus der Mathematik. Die Prioritäten der Vergleichsoperationen entsprechen der angegebenen Reihenfolge. Die Operationen „=“ und „!=“ haben somit eine geringere Priorität als die anderen Vergleichsoperationen.

### 2.3.3. Zeichenketten

Zeichenketten in XPath werden durch den Datentyp *string* repräsentiert. Eine Zeichenkette in XPath ist eine Folge von null oder mehr abstrakten Unicode-Zeichen [Con]. Ein abstraktes Unicode-Zeichen entspricht damit einem einzelnen skalaren Wert. Deshalb werden auch zusammengesetzte Zeichen, wie z.B. Umlaute, durch einen einzelnen Unicode-Wert dargestellt. Für Zeichenketten in XPath sind die Vergleichsoperationen Gleich („=“) und Ungleich („!=“) definiert.

### 2.3.4. Knotenmengen

Knotenmengen in XPath werden durch den Datentyp *node-set* repräsentiert. Eine Menge von Knoten repräsentiert in XPath eine Menge von XML-Elementen, Attributen, Kommentaren, usw.. Deshalb unterscheidet XPath sieben Typen von Knoten. Es gibt Wurzel-, Element-, Attribut-, Text-, Processing-Instruction-, Kommentar- und Namensraumknoten. Der Wurzelknoten ist ein ausgezeichnetes XML-Element. Jeder Knoten in XPath hat einen Zeichenkettenwert und einen erweiterten Namen. Der *Zeichenkettenwert* eines Knotens repräsentiert den Inhalt eines XML-Elements, eines Attributs, eines Textes, einer Processing-Instruction oder eines Kommentars. Der Zeichenkettenwert eines Namensraumknotens entspricht dem Namensraum-URI. Der *erweiterte Name* eines Knotens entspricht einem Paar, das den Namen und den Namensraum-URI des Knotens enthält.

Eine Knotenmenge in XPath ist ungeordnet. Allerdings verwendet XPath gelegentlich die *Dokumentenordnung* bzgl. eines betrachteten Knotens. Hier wird die Ordnung der Elemente, Attribute, Kommentar, usw. in dem Dokument verwendet, das den betrachteten Knoten enthält. Für Knotenmengen in XPath ist die Vereinigungsoperation „|“ definiert. Weiterhin sind die Vergleichsoperationen Gleich („=“), Ungleich („!=“), Kleiner („<“), Kleiner oder Gleich („<=“), Größer („>“) und Größer oder Gleich („>=“) definiert.

Bei einem Vergleich, an dem eine Knotenmenge teilnimmt, wird jeder Knoten der Knotenmenge verglichen. Das bedeutet für einen Vergleich zwischen zwei Knotenmengen, dass jeder Knoten der einen Knotenmenge mit jedem Knoten der anderen Knotenmenge verglichen wird. Ein Vergleich an dem mindestens eine Knotenmenge teilnimmt, ist wahr, sobald der Vergleich für *einen* Knoten wahr ist. Für den Vergleich wird ein Knoten in eine Zahl, einen booleschen Wert oder eine Zeichenkette konvertiert. Die Konvertierung ist dabei von den Typen der Operanden abhängig, die am Vergleich teilnehmen. Werden zwei Knotenmengen miteinander verglichen, werden die Zeichenkettenwerte der einzelnen Knoten miteinander verglichen. Wird eine Knotenmenge mit einer Zahl verglichen, werden die Zeichenkettenwerte der einzelnen Knoten in Zahlen konvertiert. Wird eine Knotenmenge mit einem booleschen Wert verglichen, werden die Zeichenkettenwerte der einzelnen Knoten in boolesche Werte konvertiert. In jedem anderen Fall werden die Zeichenkettenwerte der einzelnen Knoten für den Vergleich verwendet.

### 2.3.5. Lokalisierungspfade

Ein *Lokalisierungspfad* in XPath adressiert einen bestimmten Teil in einem XML-Dokument und ermittelt so eine Knotenmenge. Die Knotenmenge wird dabei relativ zu einem Kontextknoten ermittelt. Ein Kontextknoten ist ein beliebiges aber festes Element im betrachteten XML-Dokument. Ausgehend von diesem Kontextknoten werden die Knoten berechnet, die über den Lokalisierungspfad im XML-Dokument erreichbar sind. Der Kontextknoten wird dabei im Lokalisierungspfad durch ein „/“ ausgewählt. In BPEL-Prozessen wird der Kontextknoten durch eine Variable bestimmt, die vor dem „/“ angegeben wird.

Ein Lokalisierungspfad besteht aus einem oder mehreren Lokalisierungsschritten. Ein Lokalisierungsschritt ermittelt ebenfalls eine Knotenmenge. Wenn der Lokalisierungspfad aus mehreren Lokalisierungsschritten besteht, wird jeder Knoten aus der Knotenmenge des ersten Lokalisierungsschrittes als Kontextknoten für den nachfolgenden Lokalisierungsschritt verwendet. Die Knoten aus diesem Lokalisierungsschritt dienen wiederum als Kontextknoten für den folgenden Lokalisierungsschritt, usw.. Deshalb werden die verschiedenen Lokalisierungsschritte in einem Lokalisierungspfad auch durch ein „/“ voneinander getrennt.

Ein *Lokalisierungsschritt* besteht aus einer Achse, einem Knotentest und optionalen Prädikaten. Der Knotentest wird durch zwei Doppelpunkte von der Achse getrennt. Die Prädikate werden in eckigen Klammern nach dem Knotentest angegeben. Beispielsweise enthält der Lokalisierungsschritt

`child::Strasse[local-name()='An der Wiese']`

die Achse `child`, den Knotentest `Strasse` und das Prädikat `[local-name()='An der Wiese']`. Die Bestandteile eines Lokalisierungsschrittes werden im Folgenden genauer beschrieben.

### Achsen

Eine *Achse* in einem Lokalisierungsschritt spezifiziert die Beziehung zwischen dem Kontextknoten und den Knoten in der zu ermittelnden Knotenmenge. Wie bereits erwähnt wird ein XML-Dokument in XPath durch einen Baum repräsentiert. Ein XML-Element, Attribut, Kommentar, usw. in diesem XML-Dokument wird als Knoten repräsentiert. Betrachten wir einen solchen Knoten als Kontextknoten, beschreibt eine Achse, wie – ausgehend von dem Kontextknoten – über den Baum traversiert werden soll. Jeder Knoten, der im Baum durch die Achse erreicht werden kann, wird in eine Knotenmenge aufgenommen.

Im Folgenden beschreiben wir die Achsen aus XPath 1.0:

<b>self</b>	wählt die Kontextknoten selbst aus.
<b>child</b>	wählt die Kinder des Kontextknotens aus.
<b>descendant</b>	wählt die Nachkommen des Kontextknotens aus. Die Nachkommen eines Knotens sind die Kinder, deren Kinder, usw..
<b>parent</b>	wählt den Elterknoten des Kontextknotens aus.
<b>ancestor</b>	wählt die Vorfahren des Kontextknotens aus. Die Vorfahren eines Knotens sind der Elterknoten, dessen Elterknoten, usw..
<b>descendant-or-self</b>	wählt den Kontextknoten sowie dessen Nachkommen aus.
<b>ancestor-or-self</b>	wählt den Kontextknoten sowie dessen Vorfahren aus.
<b>following</b>	wählt die Knoten aus, die im gleichen Dokument wie der Kontextknoten enthalten sind und nach dem Kontextknoten definiert sind.
<b>preceding</b>	wählt die Knoten aus, die im gleichen Dokument wie der Kontextknoten enthalten sind und vor dem Kontextknoten definiert sind.
<b>following-sibling</b>	wählt die Geschwister des Kontextknotens aus, die im XML-Dokument nach dem Kontextknoten definiert sind. Die Geschwister des Kontextknotens sind die Knoten, die den gleichen Elterknoten wie der Kontextknoten haben.
<b>preceding-sibling</b>	wählt die Geschwister des Kontextknotens aus, die im XML-Dokument vor dem Kontextknoten definiert sind.

**attribute** wählt die Attribute des Kontextknotens aus.

**namespace** wählt die Namensraumknoten des Kontextknotens aus.

Die Vorfahren, Geschwister und Nachkommen eines Knotens sind niemals Attribut- und Namensraumknoten. Für die Ermittlung dieser Knoten, stehen die speziellen Achsen `attribute` und `namespace` zur Verfügung.

## Knotentest

Ein *Knotentest* spezifiziert den Knotentyp bzw. den erweiterten Namen der Knoten, die in der zu ermittelnden Knotenmenge enthalten sind. Der Knotentest folgt auf die Achse im Lokalisierungsschritt. Deshalb werden die Knoten, die über die angegebene Achse erreicht werden können, bzgl. des Knotentyps bzw. des erweiterten Namens überprüft. Für jeden Knoten, der den angegebenen Knotentyp bzw. erweiterten Namen besitzt, ist der Knotentest erfüllt. Das Resultat ist eine neue Knotenmenge, die alle Knoten enthält, die über die im Lokalisierungsschritt angegebene Achse erreicht werden können und den Knotentest erfüllen.

Im Folgenden beschreiben wir, welche Knotentypen im Knotentest spezifiziert werden können. Zusätzlich beschreiben wir, für welche Knoten der Knotentest erfüllt ist.

**node()** ist für alle Knoten eines beliebigen Typs erfüllt.

**comment()** ist für jeden Kommentarknoten erfüllt.

**text()** ist für jeden Textknoten erfüllt.

**processing-instruction()** ist für jede Processing-Instruction erfüllt.

Wird bei einem Knotentest der erweiterte Name spezifiziert, werden alle Knoten mit diesem erweiterten Namen in der Knotenmenge aufgenommen. Beispielsweise ist der Knotentest `Adressen:Strasse` für jeden Knoten im Namensraum `Adressen` mit dem Namen `Strasse` erfüllt. Der Knotentest `Adressen:*` bzw. „\*“ ist für jeden Knoten im Namensraum `Adressen` bzw. für alle Knoten erfüllt.

## Prädikate

Ein *Prädikat* filtert eine Knotenmenge, die zuvor durch den Knotentest ermittelt wurde. Für die Filterung wird im Prädikat ein XPath-Ausdruck angegeben. Bei der Auswertung dieses Ausdrucks wird jeder Knoten in der Knotenmenge als Kontextknoten betrachtet. Falls die Auswertung des Ausdrucks wahr ergibt, wird der betrachtete Kontextknoten in die Knotenmenge für den Lokalisierungsschritt aufgenommen. In unserer Analyse berücksichtigen wir die Filterung mit Hilfe von Prädikaten nicht.

### 2.3.6. Funktionen der Grundbibliothek

In diesem Abschnitt stellen wir die *vordefinierten Funktionen* vor, die in XPath-Ausdrücken vorkommen dürfen. Einige dieser Funktionen greifen auf einen Kontextknoten zu. In BPEL-Prozessen gibt es keinen voreingestellten Kontextknoten (siehe [JE07], Anhang B, SA00027). Deshalb dürfen Funktionen, die auf den Kontextknoten zugreifen, in BPEL-Prozessen nur in Prädikaten innerhalb eines Lokalisierungsschrittes verwendet werden. Wie bereits erwähnt, filtern Prädikate eine Knotenmenge. Da wir in unserer Analyse die Filterung durch Prädikate nicht berücksichtigen, brauchen wir auch keine XPath-Funktionen berücksichtigen, die auf einen Kontextknoten zugreifen.

Deshalb vernachlässigen wir im Folgenden die XPath-Funktionen `lang()`, `id()`, `last()` und `position()`, welche auf einen Kontextknoten zugreifen. Weiterhin gibt es XPath-Funktionen mit einem optionalen Argument, die auf den Kontextknoten zugreifen, wenn das Argument fehlt. Für diese Funktion vernachlässigen wir den Fall, dass das Argument fehlt. Demnach ist für uns das Argument solcher Funktionen immer obligatorisch.

Für jede berücksichtigte Funktion geben wir ihre vollständige Signatur an. Ein Argument vom Typ `object` steht für ein Argument von einem beliebigen Typ. Ein Argument gefolgt von einem Fragezeichen repräsentiert ein optionales Argument. Ein Argument gefolgt von einem Stern darf keinmal, einmal oder auch mehrmals auftreten.

#### Zahlenfunktionen

**numbers number(object?)** konvertiert das Argument in eine Zahl. Die Konvertierung ist dabei vom Typ des Arguments abhängig. Eine Zeichenkette kann nur in eine gültige Zahl überführt werden, wenn sie aus einem optionalen Minuszeichen gefolgt von einer Dezimalzahl besteht. Der boolesche Wert *true* wird in die Zahl 1 konvertiert. Der Wert *false* wird in die Zahl 0 konvertiert. Bei einer Knotenmenge  $K$  wird zunächst jeder Knoten aus  $K$  in eine Zeichenkette konvertiert. Danach werden diese Zeichenketten wie oben beschrieben in Zahlen konvertiert. Für die Konvertierung der Knotenmenge  $K$  in eine Zahl, wird schließlich die Summe aller so ermittelten Zahlen gebildet.

dieser Zeichenketten wie oben beschrieben `ko` mit der Funktion `string()` in eine Zeichenkette konvertiert. Anschließend wird diese Zeichenkette wie oben beschrieben in eine Zahl konvertiert.

Ist kein Argument angegeben, wird die Knotenmenge mit dem Kontextknoten als einziges Element als Argument verwendet. Wie bereits erwähnt, berücksichtigen wir diesen Fall im Weiteren nicht.

**numbers sum(node-set)** berechnet die Summe der Zahlen, die nach Konvertierung des Arguments entstehen. Zunächst wird für jedes Element der Knotenmenge im Argument der zugehörige Zeichenkettenwert ermittelt. Danach werden diese Zeichenkettenwerte in Zahlen konvertiert. Das bedeutet, dass auf jedes Element in der übergebenen

Knotenmenge zunächst die Funktion `string()` angewendet wird. Das Ergebnis wird danach der Funktion `number()` übergeben. Schließlich wird die Summe aller so entstandenen Zahlenwerte berechnet.

**numbers floor(numbers)** berechnet die größte ganze Zahl, die kleiner als das Argument ist. Die Funktion `floor()` rundet also ab.

**numbers ceiling(numbers)** berechnet die kleinste ganze Zahl, die größer als das Argument ist. Die Funktion `ceiling()` rundet also auf.

**numbers round(numbers)** liefert die ganze Zahl, die am nächsten an der Zahl im Argument ist. Die Funktion rundet also auf oder ab. Gibt es zwei ganze Zahlen, die dem Argument am nächsten sind, wird aufgerundet.

Bei den Rundungsfunktionen `floor()`, `ceiling()` und `round()` gibt es eine Feinheit, die wir vernachlässigen: Werte zwischen  $-0.5$  und  $0$  werden manchmal zu  $-0$  und manchmal zu  $0$  aufgerundet. Da  $-0$  und  $0$  gleich behandelt werden, bilden wir diesen Sachverhalt später nicht im formalen Modell ab. Wir runden alle Werte zwischen  $-0.5$  und  $0$  zu  $0$  auf.

## Zeichenkettenfunktionen

**string string(object?)** konvertiert das Argument in eine Zeichenkette. Wie konvertiert wird, ist vom Typ des Arguments abhängig. Eine Zahl wird in eine Zeichenkette konvertiert, indem die einzelnen Ziffern einer Zahl als Zeichen interpretiert werden. Beispielsweise wird die Zahl  $2.52$  in die Zeichenkette “2.52” konvertiert. Der Wert *NaN* wird analog in die Zeichkette “NaN” konvertiert. Sowohl  $-0$  als auch  $0$  werden in die Zeichenkette “0” konvertiert. Die Werte  $\infty$  und  $-\infty$  werden in die Zeichenketten “Infinity” und “-Infinity” konvertiert. Die boolschen Werte *true* und *false* werden in die Zeichenketten “true” und “false” konvertiert. Die Konvertierung einer Knotenmenge in eine Zeichenkette liefert den Zeichenkettenwert des ersten Knotens bzgl. der Dokumentenordnung in der Knotenmenge. Bei einer leeren Knotenmenge wird die leere Zeichenkette zurückgeliefert.

Ist kein Argument angegeben, wird die Knotenmenge mit dem Kontextknoten als einziges Element als Argument verwendet. Wie bereits erwähnt, berücksichtigen wir diesen Fall im Weiteren nicht.

**string concat(string, string, string\*)** liefert die Zeichenkette, die durch Verkettung der Argumente entsteht.

**boolean starts-with(string, string)** liefert den boolschen Wert *true*, falls die Zeichenkette im ersten Argument mit der Zeichenkette im zweiten Argument beginnt. Sonst liefert die Funktionen den Wert *false*. Ist das zweite Argument die leere Zeichenkette, wird der Wert *true* zurück gegeben.

**boolean contains(string, string)** liefert den boolschen Wert *true*, falls die Zeichenkette im ersten Argument die Zeichenkette im zweiten Argument enthält. Sonst liefert die Funktion den Wert *false*. Ist das zweite Argument die leere Zeichenkette, wird der Wert *true* zurück gegeben.

**string substring-before(string, string)** liefert den Teil der Zeichenkette im ersten Argument, der vor dem ersten Auftreten der Zeichenkette im zweiten Argument steht, z.B. `substring-before("Hallo Christian!", "Christian") = "Hallo "`. Ist das zweite Argument gar nicht im ersten Argument enthalten, wird die leere Zeichenkette zurück gegeben. Die leere Zeichenkette wird ebenfalls zurück gegeben, wenn das zweite Argument die leere Zeichenkette ist.

**string substring-after(string, string)** liefert den Teil der Zeichenkette im ersten Argument, der nach dem ersten Auftreten der Zeichenkette im zweiten Argument steht, z.B. `substring-after("Hallo Christian!", "Christian") = "!"`. Ist das zweite Argument gar nicht im ersten Argument enthalten, wird die leere Zeichenkette zurück gegeben. Ist das zweite Argument die leere Zeichenkette, wird die komplette Zeichenkette aus dem ersten Argument zurück gegeben.

**string substring(string, numbers, numbers?)** liefert einen Teil der Zeichenkette im ersten Argument. Die Position, ab der dieser Teil beginnt, wird im zweiten Argument übergeben. Dabei ist zu beachten, dass das erste Zeichen die Position 1 hat. Das dritte Argument ist optional und gibt die Länge der Teilzeichenkette an, z.B. liefert `substring("Hallo Christian!", 1, 5)` die Zeichenkette "Hallo". Fehlt das dritte Argument, wird der Teil der Zeichenkette im ersten Argument zurück gegeben, der ab der im zweiten Argument angegebenen Position beginnt und bis zum Ende der Zeichenkette im ersten Argument reicht. Enthält das zweite und dritte Argument keine ganzen Zahlen, wird der Wert mit Hilfe der Funktion `round()` gerundet.

**string string-length(string?)** liefert die Anzahl der Zeichen in der Zeichenkette, die als Argument übergeben wird.

Ist kein Argument angegeben, ermittelt sich das Argument mit Hilfe des aktuellen Kontextknotens. Wie bereits erwähnt, berücksichtigen wir diesen Fall im Weiteren nicht.

**string normalize-space(string?)** liefert die Zeichenkette, die aus der übergebenen Zeichenkette nach Normalisierung des Leerraumes entsteht. Das bedeutet, dass führende und abschließende Leerzeichen entfernt werden. Weiterhin werden mehrere aufeinander folgende Leerzeichen durch ein Leerzeichen ersetzt.

Ist kein Argument angegeben, ermittelt sich das Argument mit Hilfe des aktuellen Kontextknotens. Wie bereits erwähnt, berücksichtigen wir diesen Fall im Weiteren nicht.

**string translate(string, string, string)** liefert die Zeichenkette, die der Zeichenkette im ersten Argument entspricht, wobei jedes Vorkommen eines Zeichens aus dem zweiten Argument durch ein Zeichen aus dem dritten Argument ersetzt wird. Dabei wird ein Zeichen an Position  $x$  des zweiten Arguments durch das Zeichen an Position  $x$  des dritten Arguments ersetzt, z.B. `translate("Hallo!", "ahlo", "AHLO") = "HALLO!"`. Ist die Zeichenkette im zweiten Argument länger, als die Zeichenkette im dritten Argument, werden die letzten Zeichen des zweiten Arguments gelöscht, da es kein korrespondierendes Zeichen im dritten Argument gibt. Kommt ein Zeichen mehrmals im zweiten Argument vor, wird die Position des ersten Vorkommens verwendet.

### Boolsche Funktionen

**boolean boolean(object)** konvertiert das Argument in einen boolschen Wert. Wie konvertiert wird, ist vom Typ des Arguments abhängig. Eine Zahl wird in den Wert *true* konvertiert, wenn sie weder 0,  $-0$  noch *NaN* ist. Ansonsten ergibt die Konvertierung den Wert *false*. Eine Zeichenkette wird in den Wert *true* konvertiert, wenn die Zeichenkette nicht die leere Zeichenkette ist. Die leere Zeichenkette wird in den Wert *false* konvertiert. Eine Knotenmenge wird in den Wert *true* konvertiert, wenn die Knotenmenge nicht leer ist. Die leere Knotenmenge wird in den Wert *false* konvertiert.

**boolean not(boolean)** liefert den Wert *true*, wenn das Argument den Wert *false* enthält. Ansonsten liefert die Funktion den Wert *false*.

**boolean true()** liefert den Wert *true*. Diese Funktion steht zur Verfügung, da in XPath kein Literal für den Wert *true* definiert ist.

**boolean false()** liefert den Wert *false*. Diese Funktion steht zur Verfügung, da in XPath kein Literal für den Wert *false* definiert ist.

### Funktionen auf Knotenmengen

**numbers count(node-set)** liefert die Anzahl der Knoten in der Knotenmenge, die im Argument übergeben wird.

**string local-name(node-set?)** liefert den lokalen Teil des erweiterten Namens des ersten Knoten in der übergebenen Knotenmenge. Der erste Knoten in der Knotenmenge wird bzgl. der Dokumentenordnung ermittelt. Ist die Knotenmenge im Argument leer, wird die leere Zeichenkette geliefert. Besitzt der erste Knoten keinen erweiterten Namen, wird ebenfalls die leere Zeichenkette geliefert.

Ist kein Argument angegeben, wird als Argument die Knotenmenge mit dem Kontextknoten als einziges Element verwendet. Wie bereits erwähnt, berücksichtigen wir diesen Fall im Weiteren nicht.

**string namespace-uri(node-set?)** liefert den Namensraum-URI des erweiterten Namens des ersten Knotens in der übergebenen Knotenmenge. Der erste Knoten in der Knotenmenge wird bzgl. der Dokumentenordnung ermittelt. Die Funktion liefert die leere Zeichenkette, wenn die übergebene Knotenmenge leer ist, der erste Knoten in der Knotenmenge keinen erweiterten Namen besitzt oder der Namensraum-URI leer ist.

Ist kein Argument angegeben, wird als Argument die Knotenmenge mit dem Kontextknoten als einziges Element verwendet. Wie bereits erwähnt, berücksichtigen wir diesen Fall im Weiteren nicht.

**string name(node-set?)** liefert den erweiterten Namen des ersten Knotens in der übergebenen Knotenmenge repräsentiert. Der erste Knoten in der Knotenmenge wird bzgl. der Dokumentenordnung ermittelt.

Ist kein Argument angegeben, wird als Argument die Knotenmenge mit dem Kontextknoten als einziges Element verwendet. Wie bereits erwähnt, berücksichtigen wir diesen Fall im Weiteren nicht.

### 2.3.7. Abgekürzte Syntax

In XPath können Ausdrücke in abgekürzter Syntax beschrieben werden. So kann beispielsweise die Achse `child` in Lokalisierungspfaden weggelassen werden, da diese Achse die Standardachse darstellt. Weitere Abkürzungen sind in der XPath-Spezifikation [CD99] enthalten.

In unserer Analyse unterstützen wir die abgekürzte Syntax nicht. Soll ein BPEL-Prozess analysiert werden, müssen deshalb die XPath-Ausdrücke in abgekürzter Syntax in einen XPath-Ausdruck in vollständiger Syntax überführt werden. Jeder XPath-Ausdruck in abgekürzter Syntax kann dabei mit Hilfe der XPath-Spezifikation [CD99] in einen Ausdruck in vollständiger Syntax überführt werden. Somit können wir jeden regulären BPEL-Prozess analysieren, obwohl unsere Analyse die abgekürzte Syntax für XPath-Ausdrücke nicht unterstützt.

### 3. Ein Datenmodell für WS-BPEL-Prozesse

Im vorhergehenden Kapitel haben wir die technologischen Grundlagen erläutert, die in dieser Arbeit verwendet werden. Dabei haben wir die Sprache BPEL kennengelernt, mit der Geschäftsprozesse modelliert werden können. Weiterhin haben wir die Sprache XML-Schema vorgestellt, die für die Typisierung von Daten in BPEL-Prozessen verwendet wird. Schließlich haben wir die Sprache XPath kennengelernt, die standardmäßig für die Beschreibung von Datenmanipulationen in BPEL-Prozessen genutzt wird.<sup>1</sup>

In diesem Kapitel entwickeln wir nun ein Datenmodell für BPEL-Prozesse. Zunächst beschreiben wir ein formales Datenmodell für XPath, das die abstrakte Interpretation von XPath-Ausdrücken in BPEL-Prozessen ermöglicht. In diesem formalen Modell entwickeln wir für jeden XPath-Datentyp eine abstrakte Semantik. Auf Grundlage dieser abstrakten Semantik definieren wir im formalen Modell zusätzlich die verschiedenen Operationen und Funktionen, die XPath zur Verfügung stellt. Danach beschreiben wir ein Modell zur Repräsentation von XPath-Ausdrücken in BPEL-Prozessen. Für ein vollständiges Datenmodell für BPEL-Prozesse stellen wir schließlich noch ein Modell zur Repräsentation des Kontrollflusses und der Datenabhängigkeiten eines BPEL-Prozesses vor. In diese Repräsentation betten wir das Modell zur Repräsentation von XPath-Ausdrücken ein.

In Kapitel 3.1 definieren wir das formale Modell für Daten in einem XPath-Ausdruck. Danach stellen wir in Kapitel 3.2 den Drei-Adress-Code für die Repräsentation von XPath-Ausdrücken in BPEL-Prozessen vor. Abschließend erläutern wir in Kapitel 3.3 die Concurrent-Single-Static-Assignment-Form zur Repräsentation des Kontrollflusses und der Datenabhängigkeiten eines BPEL-Prozesses.

#### 3.1. Ein formales Datenmodell für XPath

In diesem Kapitel entwickeln wir ein formales Datenmodell für XPath 1.0. Dieses formale Modell beschreibt für jeden Datentyp in XPath eine abstrakte Semantik. Auf Grundlage dieser abstrakten Semantiken definieren wir zusätzlich die verschiedenen Operationen und Funktionen, die XPath zur Verfügung stellt. Das formale Modell bietet die Grundlage für die abstrakte Interpretation von XPath-Ausdrücken, die in der vorliegenden Arbeit vorgestellt wird.

---

<sup>1</sup>Werden die vorgestellten Techniken im Folgenden verwendet, heben wir die Schlüsselwörter mit einem speziellen Schrifttyp hervor. Beispielsweise werden die verschiedenen BPEL-Aktivitäten folgendermaßen dargestellt: `assign`-Aktivität, `while`-Aktivität, usw..

Da wir in unserer abstrakten Interpretation die Wertebereiche für XPath-Ausdrücke analysieren, definieren wir ein formales Modell, mit dem die Wertebereiche möglichst genau eingegrenzt werden können. In unserem formalen Datenmodell bleiben die Eigenschaften Sicherheit und Lebendigkeit [AS85] erhalten. Ein XPath-Ausdruck ist

- *sicher*, wenn bei seiner Auswertung kein Fehler auftritt und das korrekte Ergebnis berechnet wird;
- *lebendig*, wenn seine Auswertung terminiert.

Somit wird jeder sichere XPath-Ausdruck auch unter Verwendung der abstrakten Datenwerte ohne das Auftreten von Fehlern ausgewertet und es wird das korrekte Ergebnis geliefert. Für jeden lebendigen XPath-Ausdruck terminiert auch die Auswertung unter Verwendung der abstrakten Datenwerte. Da die beiden Eigenschaften Sicherheit und Lebendigkeit zur (totalen) Korrektheit zusammengefasst werden, können wir sagen, dass die Eigenschaft (totale) Korrektheit in unserer Abstraktion erhalten bleibt.

Um diese Eigenschaften in unserer Abstraktion zu sichern, verwenden wir *Galois Korrespondenzen*. Eine Galois Korrespondenz sichert uns zu, dass wir keine Information bei der Abstraktion verlieren. Wir erläutern diesen Informationserhalt anhand von Abb. 3.1. In

$$\begin{array}{ccc}
 & \alpha & \\
 & \longrightarrow & \\
 M_0 & & M_1 \\
 & \longleftarrow & \\
 & \gamma &
 \end{array}$$

$\alpha$  ist die Abstraktionsfunktion  
 $\gamma$  ist die Konkretisierungsfunktion

**Abbildung 3.1.:** Galois Korrespondenz

der Abbildung werden die konkreten Datenwerte durch die Menge  $M_0$  und die abstrakten Datenwerte durch die Menge  $M_1$  repräsentiert. Mit Hilfe der Abstraktionsfunktion  $\alpha$  werden die konkreten Datenwerte auf ihre abstrakte Repräsentation abgebildet. Abstrakte Datenwerte werden mit Hilfe der Konkretisierungsfunktion  $\gamma$  auf die konkreten Datenwerte abgebildet. Wenden wir nun auf eine Teilmenge  $X$  der Menge der konkreten Datenwerte  $M_0$  die Abstraktionsfunktion und auf das Ergebnis die Konkretisierungsfunktion an, sichert uns eine Galois Korrespondenz zu, dass die Ausgangsmenge  $X$  auf jeden Fall im Ergebnis enthalten ist. Es gilt also:  $X \subseteq \gamma(\alpha(X))$ . Das bedeutet wiederum, dass wir bei der Abstraktion keine Informationen über die konkreten Datenwerte verlieren. Die mathematische Definition sowie weitere Eigenschaften von Galois Korrespondenzen sind in Anhang A.1 enthalten.

Im Folgenden entwickeln wir für die Datentypen `numbers` und `string` aus XPath Galois Korrespondenzen. Dabei sind drei Schritte nötig:

1. Definiere einen vollständigen Verband  $M_0$  für die konkreten Datenwerte;
2. Definiere einen vollständigen Verband  $M_1$  für die abstrakten Datenwerte;
3. Definiere die Abstraktionsfunktion  $\alpha$  und die Konkretisierungsfunktion  $\gamma$  für die Galois Korrespondenz  $(M_0, \alpha, \gamma, M_1)$ .

Der Datentyp `boolean` aus XPath erfordert keine Galois Korrespondenz für die Abstraktion, da sein Zustandsraum hinreichend klein ist. Deshalb definieren wir für den Datentyp `boolean` lediglich einen vollständigen Verband für die abstrakten Datenwerte. Der Datentyp `node-set` aus XPath beschreibt eine Menge von XML-Elementen und Attributen. Der Typ dieser Elemente setzt sich aus den anderen genannten Typen `boolean`, `numbers` und `string` zusammen. Deshalb definieren wir für diese Elemente eine hierarchische Struktur und repräsentieren den Datentyp `node-set` als Menge dieser hierarchischen Strukturen.

Für den Datentyp `node-set` definieren wir keine Galois Korrespondenz, da wir den betrachteten BPEL-Prozess und damit auch den Zustandsraum für den Datentyp nicht kennen. Möchten wir für den Datentyp `node-set` eine Galois Korrespondenz definieren, müssen wir alle möglichen XML-Schemata und alle möglichen XPath-Lokalisierungspfade berücksichtigen. In dieser Arbeit verzichten wir darauf und definieren zur Abstraktion stattdessen eine Menge der hierarchischen Strukturen. Diese Abstraktion kann dann von einer Datenanalyse analysiert werden.

### 3.1.1. Der Datentyp `boolean`

Ein Datum vom Typ `boolean` kann den Wert `true` oder `false` annehmen. Dementsprechend einfach können wir einen endlichen Verband für den Datentyp `boolean` definieren. Sei  $B$  die Menge der Werte `true` und `false`:

$$B =_{\text{def}} \{true, false\}.$$

Dann repräsentiert die Menge  $B$  den Wertebereich eines Datums vom Typ `boolean`. Die Potenzmenge von  $B$  beschreibt alle möglichen Einschränkungen an den Wertebereich. Somit repräsentiert die Potenzmenge von  $B$  genau die Ergebnisse, die in unserer Datenanalyse berechnet werden können. Da der Zustandsraum für den Typ `boolean` sehr klein ist, benötigen wir keine Galois Korrespondenz, die eine korrekte Abstraktion zusichert. Wir definieren lediglich einen vollständigen Verband<sup>2</sup>  $B_1$  für die Potenzmenge von  $B$ . Trivialerweise ist die Abstraktion korrekt.

---

<sup>2</sup>Die Definition eines vollständigen Verbands ist im Anhang A.1 enthalten.

**Definition 1 (vollständiger Verband für boolesche Werte)**

Für die Menge  $B$  definieren wir den vollständigen Verband  $B_1$  wie folgt:

$$B_1 =_{\text{def}} (\mathcal{P}(B), \subseteq, \sqcup_{\text{bool}}, \sqcap_{\text{bool}}, \emptyset, B) \text{ mit}$$

$$\begin{aligned} \mathcal{P}(B) &- \text{Potenzmenge von } B \\ \subseteq &- \text{Teilmengenbeziehung} \\ \sqcup_{\text{bool}} &= \bigcup Y \text{ für } Y \subseteq \mathcal{P}(B) \\ \sqcap_{\text{bool}} &= \bigcap Y \text{ für } Y \subseteq \mathcal{P}(B) \end{aligned}$$

┘

Jedem konkreten booleschen Wert können wir nun einen abstrakten Wert aus dem vollständigen Verband  $B_1$  zuordnen. Für diese Abbildung definieren wir eine Repräsentationsfunktion  $\beta_{\text{Boolean}} : B \rightarrow B_1$  wie folgt:

$$\beta_{\text{Boolean}}(x) =_{\text{def}} \begin{cases} \{true\}, & \text{wenn } x = 1 \vee x = true; \\ \{false\}, & \text{wenn } x = 0 \vee x = false; \end{cases}$$

**Operationen auf booleschen Werten**

Die abstrakten Werte aus  $B_1$  repräsentieren boolesche Werte in XPath-Ausdrücken. Für solche booleschen Werte sind in XPath die logischen Operationen „or“ (Disjunktion) und „and“ (Konjunktion) sowie die Vergleichsoperationen „=“ (Gleich) und „!=“ (Ungleich) definiert. Um diese Operationen formal zu repräsentieren, definieren wir im Folgenden die Funktionen „or“, „and“, „=\_{Bool}“ und „!=\_{Bool}“ auf Grundlage des vollständigen Verbands  $B_1$ . Die Funktionen beschreiben somit die logische Verknüpfung und den Vergleich von abstrakten booleschen Werten. Dabei repräsentiert ein abstrakter boolescher Wert immer einen der beiden booleschen Werte *true* und *false* oder beide booleschen Werte.

Die Disjunktion zweier boolescher Werte ist wahr, sobald einer der beiden booleschen Werte *true* ist. Repräsentiert also ein abstrakter boolescher Wert ausschließlich den Wert *true*, ist die Disjunktion der abstrakten booleschen Werte wahr. Die Disjunktion zweier boolescher Werte ist falsch, wenn beide booleschen Werte *false* sind. Repräsentieren also beide abstrakten Werte ausschließlich den Wert *false*, liefert die Disjunktion der abstrakten Werte falsch. Repräsentiert einer der abstrakten booleschen Werte sowohl den Wert *true* als auch den Wert *false*, kann die Disjunktion sowohl wahr als auch falsch sein.

**Definition 2 (Disjunktion „or“ für boolesche Werte)**

Die Funktion  $or : B_1 \times B_1 \rightarrow B_1$  ist wie folgt definiert:

$$X \text{ or } Y =_{\text{def}} \begin{cases} \{true\}, & \text{für } X = \{true\} \vee Y = \{true\}; \\ \{false\}, & \text{für } X = Y = \{false\}; \\ \{true, false\} & \text{sonst.} \end{cases}$$

┘

Die Konjunktion zweier boolescher Werte ist wahr, wenn beide booleschen Werte *true* sind. Repräsentieren also beide abstrakten booleschen Werte ausschließlich den Wert *true*, ist die Konjunktion der abstrakten booleschen Werte wahr. Die Konjunktion zweier boolescher Werte ist falsch, sobald einer der beiden booleschen Werte falsch ist. Repräsentiert also ein abstrakter boolescher Wert ausschließlich den Wert *false*, ist die Konjunktion der abstrakten booleschen Werte falsch. Repräsentiert einer der abstrakten booleschen Werte sowohl den Wert *true* als auch den Wert *false*, kann die Konjunktion sowohl wahr als auch falsch sein.

**Definition 3 (Konjunktion „and“ für boolesche Werte)**

Die Funktion  $and : B_1 \times B_1 \longrightarrow B_1$  ist wie folgt definiert:

$$X \text{ and } Y =_{def} \begin{cases} \{true\}, & \text{für } X = Y = \{true\}; \\ \{false\}, & \text{für } X = \{false\} \vee Y = \{false\}; \\ \{true, false\} & \text{sonst.} \end{cases}$$

┘

Zwei boolesche Werte sind gleich, wenn ihre Werte gleich sind. Zwei abstrakte boolesche Werte sind also gleich, wenn sie den gleichen booleschen Wert repräsentieren. Repräsentiert einer der abstrakten booleschen Werte sowohl den Wert *true* als auch den Wert *false*, kann der Vergleich sowohl wahr als auch falsch sein.

**Definition 4 (Vergleichsoperation „=“ für boolesche Werte)**

Die Funktion  $=_{Bool} : B_1 \times B_1 \longrightarrow B_1$  ist wie folgt definiert:

$$X =_{Bool} Y =_{def} \begin{cases} \{true\}, & \text{für } X = Y = \{true\} \vee X = Y = \{false\}; \\ \{false\}, & \text{für } X = \{true\} \wedge Y = \{false\}; \\ \{false\}, & \text{für } X = \{false\} \wedge Y = \{true\}; \\ \{true, false\} & \text{sonst.} \end{cases}$$

┘

Zwei boolesche Werte sind ungleich, wenn ihre Werte ungleich sind. Zwei abstrakte boolesche Werte sind also ungleich, wenn sie einen unterschiedlichen booleschen Wert repräsentieren. Repräsentiert einer der abstrakten booleschen Werte sowohl den Wert *true* als auch den Wert *false*, kann der Vergleich sowohl wahr als auch falsch sein.

**Definition 5 (Vergleichsoperation „!“ für boolesche Werte)**

Die Funktion  $!=_{Bool} : B_1 \times B_1 \longrightarrow B_1$  ist wie folgt definiert:

$$X !=_{Bool} Y =_{def} \begin{cases} \{false\}, & \text{für } X = Y = \{true\} \vee X = Y = \{false\}; \\ \{true\}, & \text{für } X = \{true\} \wedge Y = \{false\}; \\ \{true\}, & \text{für } X = \{false\} \wedge Y = \{true\}; \\ \{true, false\} & \text{sonst.} \end{cases}$$

┘

### 3.1.2. Der Datentyp numbers

Wie in Kapitel 2.3.2 erwähnt, gibt es für alle Zahlen in XPath einen Datentyp `numbers`. Ganze Zahlen und Dezimalzahlen werden also nicht unterschieden. Der Typ `numbers` in XPath repräsentiert eine Gleitkommazahl nach dem Standard IEEE 754. Für Sonderfälle stellt der Standard IEEE 754 spezielle Werte zur Verfügung. Der Wert `NaN` („Not a Number“) wird als Darstellung für explizit unmögliche Zahlen verwendet. Zu große Ergebnisse werden durch die Werte  $\infty$  und  $-\infty$  dargestellt. Weiterhin existiert die Null in zwei verschiedenen Darstellungen  $-0$  und  $0$ , die allerdings gleichwertig behandelt werden. Deshalb repräsentieren wir im formalen Modell nur die Null ohne negatives Vorzeichen.

#### Eine Galois Korrespondenz für Zahlen

Mathematisch modellieren wir Zahlen aus XPath als die Menge der reellen Zahlen, inklusive der speziellen Werte `NaN`,  $\infty$  und  $-\infty$ :

$$G =_{def} \mathbb{R} \cup \{NaN, \infty, -\infty\}$$

Die Potenzmenge von  $G$  beschreibt alle möglichen Einschränkungen an den Wertebereich für Zahlen. Somit repräsentiert die Potenzmenge von  $G$  genau die Werte, die in einem Datum vom Typ `numbers` vorkommen können. Deshalb definieren wir für die Potenzmenge von  $G$  einen vollständigen Verband  $L_0$ :

#### Definition 6 (vollständiger Verband für Zahlen)

Für die Potenzmenge von  $G$  definieren wir den vollständigen Verband  $L_0$  wie folgt:

$$L_0 =_{def} (\mathcal{P}(G), \subseteq, \sqcup, \sqcap, \emptyset, G) \text{ mit}$$

$$\begin{aligned} \mathcal{P}(G) &- \text{Potenzmenge von } G \\ \subseteq &- \text{Teilmengenbeziehung} \\ \sqcup &= \bigcup Y \text{ für } Y \subseteq \mathcal{P}(G) \\ \sqcap &= \bigcap Y \text{ für } Y \subseteq \mathcal{P}(G) \end{aligned}$$

┘

Im formalen Modell abstrahieren wir von Zahlen in XPath durch Verwendung von Intervallen. Deshalb definieren wir die Menge aller Zahlenintervalle (Intervalle, die Zahlen enthalten) folgendermaßen:<sup>3</sup>

$$\text{Zahlenintervalle} =_{def} \{[x, y] \mid x \leq y \text{ für } x, y \in G \setminus \{NaN\}\} \cup \{[\ ]\}$$

<sup>3</sup>Wir verwenden die Intervallklammern „[“ und „]“ zur Beschreibung eines geschlossenen Intervalls.

In unserer Abstraktion müssen wir zusätzlich den speziellen Wert  $NaN$  berücksichtigen. Deshalb definieren wir nun die Menge aller Intervalle, die sowohl Zahlen als auch den speziellen Wert  $NaN$  repräsentieren. In dieser Menge ist jedes Intervall aus *Zahlenintervalle* sowie jedes Intervall aus *Zahlenintervalle* vereinigt mit dem speziellen Zahlenwert  $NaN$  enthalten. Demnach definieren wir die Menge, die alle abstrakten Zahlenwerte enthält, wie folgt:

$$\text{Intervalle} =_{def} \text{Zahlenintervalle} \cup \{I \cup NaN \mid I \in \text{Zahlenintervalle}\}$$

$I \cup NaN$  repräsentiert dabei alle Zahlenwerte aus dem Zahlenintervall  $I$  sowie den speziellen Zahlenwert  $NaN$ . Die Semantik der Operation „ $\cup$ “ entspricht hier also nicht der klassischen Mengenvereinigung, sondern der Verknüpfung eines Zahlenintervalls mit dem speziellen Wert  $NaN$ . Möchten wir im Folgenden ein Zahlenintervall  $[x_1, x_2]$  beschreiben, das entweder mit dem speziellen Wert  $NaN$  verknüpft ist oder nicht, verwenden wir die Schreibweise  $[x_1, x_2] \cup X$  mit  $X = (\emptyset \vee NaN)$ . Dabei ist  $X = \emptyset$ , wenn das Zahlenintervall *nicht* mit dem Wert  $NaN$  verknüpft ist. Sonst ist  $X = NaN$ .

Für die Menge *Intervalle*, die alle abstrakten Zahlenwerte enthält, definieren wir nun einen vollständiger Verband  $L_1$ :

**Definition 7 (vollständiger Verband für Intervalle)**

Für die Menge *Intervalle* definieren wir den vollständigen Verband  $L_1$  wie folgt:

$$L_1 =_{def} (\text{Intervalle}, \sqsubseteq_{Int}, \sqcup_{Int}, \sqcap_{Int}, [ ], [\infty, -\infty] \cup NaN) \text{ mit}$$

- Intervalle* – Menge aller Intervalle,
- die die Zahlen in XPath repräsentieren
- $\sqsubseteq_{Int}$  – die Enthaltenrelation für Intervalle
- $\sqcup_{Int}$  – die kleinste obere Schranke für Intervalle
- $\sqcap_{Int}$  – die größte untere Schranke für Intervalle

┘

Im Folgenden definieren wir die Enthaltenrelation, die kleinste obere Schranke sowie die größte untere Schranke für Intervalle. Dafür sei  $x, y \in \text{Zahlenintervalle}$ ;  $x_1, x_2, y_1, y_2 \in G \setminus NaN$ ;  $X = (\emptyset \vee NaN)$  und  $Y = (\emptyset \vee NaN)$ .

Die Enthaltenrelation  $\sqsubseteq_{Int} \subseteq \text{Intervalle} \times \text{Intervalle}$  für zwei Intervalle  $x$  und  $y$  definieren wir induktiv:

$$\begin{aligned}
 \text{(IA)} \quad & x \sqsubseteq_{Int} y \Leftrightarrow x = [ ] \\
 & [x_1, x_2] \sqsubseteq_{Int} [y_1, y_2] \Leftrightarrow x_1 \geq y_1 \wedge x_2 \leq y_2 \\
 \text{(IS)} \quad & x \sqsubseteq_{Int} y \Rightarrow x \sqsubseteq_{Int} y \cup NaN \\
 & x \sqsubseteq_{Int} y \Rightarrow x \cup NaN \sqsubseteq_{Int} y \cup NaN
 \end{aligned}$$

Im Induktionsanfang beschreiben wir die Enthaltenrelation für Elemente der Menge *Zahlenintervalle*. Das leere Zahlenintervall ist in jedem anderen Zahlenintervall enthalten. Weiterhin ist ein Zahlenintervall  $x$  in einem Zahlenintervall  $y$  enthalten, wenn die linke Grenze von  $x$  größer als die linke Grenze von  $y$  ist und die rechte Grenze von  $x$  kleiner als die rechte Grenze von  $y$  ist. Im Induktionsschritt beschreiben wir die Enthaltenrelation für Zahlenintervalle vereinigt mit  $NaN$ . Für jedes Zahlenintervall  $x$ , das in einem Zahlenintervall  $y$  enthalten ist, gilt, dass  $x$  auch im Intervall  $y \cup NaN$  enthalten ist. Zusätzlich gilt für jedes Zahlenintervall  $x$ , das in einem Zahlenintervall  $y$  enthalten ist, dass  $x \cup NaN$  auch im Intervall  $y \cup NaN$  enthalten ist.

Die kleinste obere Schranke zweier Intervalle  $x$  und  $y$  ist das kleinste Intervall, das sowohl  $x$  als auch  $y$  enthält. Erneut definieren wir die kleinste obere Schranke  $\sqcup_{Int} \subseteq \text{Intervalle} \times \text{Intervalle}$  induktiv:<sup>4</sup>

$$\begin{aligned}
 \text{(IA)} \quad & [ ] \sqcup_{Int} y =_{\text{def}} y \\
 & x \sqcup_{Int} [ ] =_{\text{def}} x \\
 & [x_1, x_2] \sqcup_{Int} [y_1, y_2] =_{\text{def}} [ \min(x_1, y_1) , \max(x_2, y_2) ] \\
 \text{(IS)} \quad & x \cup X \sqcup_{Int} y \cup Y =_{\text{def}} (x \sqcup_{Int} y) \cup X \cup Y
 \end{aligned}$$

Im Induktionsanfang definieren wir die kleinste obere Schranke für Zahlenintervalle. Im Induktionsschritt definieren wir die kleinste obere Schranke für Zahlenintervalle vereinigt mit  $NaN$ . Wird die kleinste obere Schranke auf eine Menge von Intervallen angewendet, berechnet die Operation  $\sqcup_{Int}$  die kleinste obere Schranke aller Intervalle in der Menge. Somit gilt für alle  $I_1, I_2, \dots, I_n \in \text{Zahlenintervalle}$  mit  $n \in \mathbb{N}$ :

$$\sqcup_{Int} \{I_1, I_2, \dots, I_n\} = I_1 \sqcup_{Int} I_2 \sqcup_{Int} \dots \sqcup_{Int} I_n$$

Die größte untere Schranke zweier Intervalle  $x$  und  $y$  ist das größte Intervall, das sowohl in  $x$  als auch in  $y$  enthalten ist. Analog zur kleinsten oberen Schranke definieren wir die größte untere Schranke  $\sqcap_{Int} \subseteq \text{Intervalle} \times \text{Intervalle}$  induktiv:

---

<sup>4</sup>Dabei berechnet  $\min$  bzw.  $\max$  das Minimum bzw. Maximum zweier Zahlen. Diese Symbolik wird auch im weiteren Text verwendet.

$$\begin{aligned}
 \text{(IA)} \quad & [] \sqcap_{Int} y =_{def} [] \\
 & x \sqcap_{Int} [] =_{def} [] \\
 & [x_1, x_2] \sqcap_{Int} [y_1, y_2] =_{def} [max(x_1, y_1), min(x_2, y_2)] \\
 \text{(IS)} \quad & x \cup X \sqcap_{Int} y \cup Y =_{def} (x \sqcap_{Int} y) \cup X \cup Y
 \end{aligned}$$

Um eine Galois Korrespondenz zwischen  $L_0$  und  $L_1$  zu entwickeln, definieren wir nun noch die Repräsentationsfunktion  $\beta_{Zahlen} : G \rightarrow L_1$ . Die Repräsentationsfunktion ordnet jedem Element aus  $G$  ein Intervall aus *Intervall* zu. Der spezielle Wert  $NaN$  wird auf das leere Intervall vereinigt mit  $NaN$  abgebildet.

$$\beta_{Zahlen}(x) =_{def} \begin{cases} [x, x], & \text{für } x \in G \setminus \{NaN\}; \\ [] \cup NaN, & \text{für } x = NaN. \end{cases}$$

Schließlich definieren wir nun noch die Abstraktionsfunktion  $\alpha_{Zahlen} : L_0 \rightarrow L_1$  und die Konkretisierungsfunktion  $\gamma_{Zahlen} : L_1 \rightarrow L_0$  für die Galois Korrespondenz. Dabei können wir die Repräsentationsfunktion verwenden.

$$\begin{aligned}
 \alpha_{Zahlen}(Y) &=_{def} \sqcup_{Int} \{ \beta_{Zahlen}(x) \mid x \in Y \} \text{ mit } Y \subseteq G \\
 \gamma_{Zahlen}(l) &=_{def} \{ x \in G \mid \beta_{Zahlen}(x) \sqsubseteq_{Int} l \} \text{ mit } l \in L_1
 \end{aligned}$$

Nach Satz A.1<sup>5</sup> bildet  $(L_0, \alpha_{Zahlen}, \gamma_{Zahlen}, L_1)$  eine Galois Korrespondenz. Somit ist unsere Abstraktion für Zahlenwerte in XPath korrekt.

## Operationen auf Zahlen

Die Intervalle aus  $L_1$  repräsentieren konkrete Zahlenwerte in XPath. Solche Zahlenwerte werden mit Hilfe von Operationen im XPath-Ausdruck manipuliert oder verglichen. Um diese Operationen formal zu repräsentieren, definieren wir im Folgenden Funktionen auf Grundlage des vollständigen Verbands  $L_1$ . Die Funktionen beschreiben somit die Manipulation von abstrakten Zahlenwerten.

Zunächst definieren wir die Funktionen „ $\oplus$ “, „ $\ominus$ “, „ $\otimes$ “, „ $\oslash$ “ und „ $\bigcirc_{mod}$ “ für die numerischen Operationen „+“, „-“, „\*“, „div“ und „mod“ aus XPath. Dafür sei  $X = (\emptyset \vee NaN)$  und  $Y = (\emptyset \vee NaN)$ .

Die Addition und Subtraktion zweier Zahlen können wir in der Abstraktion analog auf den Intervallgrenzen ausführen. Da der spezielle Wert  $NaN$  bei der Addition und Subtraktion immer erhalten bleibt, müssen wir diesen Wert nicht gesondert betrachten.

<sup>5</sup>Der Satz A.1 ist im Anhang A formuliert.

**Definition 8 (Addition auf Intervallen)**

Die Funktion  $\oplus : L_1 \times L_1 \longrightarrow L_1$  ist wie folgt definiert:

$$x \oplus y =_{def} \begin{cases} [x_1 + y_1, x_2 + y_2] \cup X \cup Y, & \text{für } x = [x_1, x_2] \cup X \wedge y = [y_1, y_2] \cup Y; \\ [ ] \cup X \cup Y, & \text{für } x = [ ] \cup X \vee y = [ ] \cup Y. \end{cases}$$

┘

**Definition 9 (Subtraktion auf Intervallen)**

Die Funktion  $\ominus : L_1 \times L_1 \longrightarrow L_1$  ist wie folgt definiert:

$$x \ominus y =_{def} \begin{cases} [x_1 - y_2, x_2 - y_1] \cup X \cup Y, & \text{für } x = [x_1, x_2] \cup X \wedge y = [y_1, y_2] \cup Y; \\ [ ] \cup X \cup Y, & \text{für } x = [ ] \cup X \vee y = [ ] \cup Y. \end{cases}$$

┘

Die Multiplikation und die Division erfordern aufgrund ihrer Eigenschaften eine Fallunterscheidung. So ist z.B. das Produkt zweier Zahlen positiv, wenn beide Operanden das gleiche Vorzeichen haben. Haben die beiden Operanden ein unterschiedliches Vorzeichen, ist das Produkt negativ. Um die Multiplikation für Intervalle zu definieren, müssen diese Eigenschaften beachtet werden. Ein Intervall repräsentiert den kleinsten und den größten möglichen Wert einer Variablen. Somit muss im Produkt zweier Intervalle sichergestellt sein, dass die linke Grenze des Produkts auch dem kleinsten möglichen Wert entspricht bzw. dass die rechte Grenze des Produkts auch dem größten möglichen Wert entspricht. Dies geht nur unter Beachtung der Vorzeichen.

**Definition 10 (Multiplikation auf Intervallen)**

Die Funktion  $\otimes : L_1 \times L_1 \longrightarrow L_1$  ist wie folgt definiert:

$$x \otimes y =_{def} \begin{cases} ([x_1, x_2] \otimes [y_1, y_2]) \cup X \cup Y, & \text{für } x = [x_1, x_2] \cup X \wedge y = [y_1, y_2] \cup Y; \\ [ ] \cup X \cup Y, & \text{für } x = [ ] \cup X \vee y = [ ] \cup Y. \end{cases}$$

mit:

$$[x_1, x_2] \otimes [y_1, y_2] = \begin{cases} [x_1 \cdot y_1, x_2 \cdot y_2], & \text{wenn } x_1, x_2, y_1, y_2 \text{ positiv;} \\ [x_2 \cdot y_1, x_2 \cdot y_2], & \text{wenn } x_1, x_2, y_2 \text{ positiv und } y_1 \text{ negativ;} \\ [x_2 \cdot y_1, x_1 \cdot y_2], & \text{wenn } x_1, x_2 \text{ positiv und } y_1, y_2 \text{ negativ;} \\ [x_1 \cdot y_2, x_2 \cdot y_2], & \text{wenn } x_2, y_1, y_2 \text{ positiv und } x_1 \text{ negativ;} \\ [ \min(x_1 \cdot y_2, x_2 \cdot y_1), \\ \max(x_1 \cdot y_1, x_2 \cdot y_2) ], & \text{wenn } x_2, y_2 \text{ positiv und } x_1, y_1 \text{ negativ;} \\ [x_2 \cdot y_1, x_1 \cdot y_1], & \text{wenn } x_2 \text{ positiv und } x_1, y_1, y_2 \text{ negativ;} \\ [x_1 \cdot y_2, x_2 \cdot y_1], & \text{wenn } y_1, y_2 \text{ positiv und } x_1, x_2 \text{ negativ;} \\ [x_1 \cdot y_2, x_1 \cdot y_1], & \text{wenn } y_2 \text{ positiv und } x_1, x_2, y_1 \text{ negativ;} \\ [x_2 \cdot y_2, x_1 \cdot y_1], & \text{wenn } x_1, x_2, y_1, y_2 \text{ negativ.} \end{cases}$$

┘

**Definition 11 (Division auf Intervallen)**

Die Funktion  $\odot : L_1 \times L_1 \longrightarrow L_1$  ist wie folgt definiert:

$$x \odot y =_{def} \begin{cases} ([x_1, x_2] \odot [y_1, y_2]) \cup X \cup Y, & \text{für } x = [x_1, x_2] \cup X \wedge y = [y_1, y_2] \cup Y; \\ [ ] \cup X \cup Y, & \text{für } x = [ ] \cup X \vee y = [ ] \cup Y. \end{cases}$$

mit:

$$[x_1, x_2] \odot [y_1, y_2] = \begin{cases} [x_1 \div y_2, x_2 \div y_1], & \text{wenn } x_1, x_2, y_1, y_2 \text{ positiv;} \\ [x_2 \div y_1, x_2 \div y_2], & \text{wenn } x_1, x_2, y_2 \text{ positiv und } y_1 \text{ negativ;} \\ [x_2 \div y_2, x_1 \div y_1], & \text{wenn } x_1, x_2 \text{ positiv und } y_1, y_2 \text{ negativ;} \\ [x_1 \div y_1, x_2 \div y_1], & \text{wenn } x_2, y_1, y_2 \text{ positiv und } x_1 \text{ negativ;} \\ [ \min(x_1 \div y_2, x_2 \div y_1), \\ \max(x_1 \div y_1, x_2 \div y_2) ], & \text{wenn } x_2, y_2 \text{ positiv und } x_1, y_1 \text{ negativ;} \\ [x_2 \div y_2, x_1 \div y_2], & \text{wenn } x_2 \text{ positiv und } x_1, y_1, y_2 \text{ negativ;} \\ [x_1 \div y_1, x_2 \div y_2], & \text{wenn } y_1, y_2 \text{ positiv und } x_1, x_2 \text{ negativ;} \\ [x_1 \div y_2, x_1 \div y_1], & \text{wenn } y_2 \text{ positiv und } x_1, x_2, y_1 \text{ negativ;} \\ [x_2 \div y_1, x_1 \div y_2], & \text{wenn } x_1, x_2, y_1, y_2 \text{ negativ.} \end{cases}$$

┘

Bei der Modulo-Operation von XPath ist zu beachten, dass die Operation auf Gleitkommazahlen definiert ist [CD99]. Weiterhin steht in der Spezifikation von XPath, dass der Operator „*mod*“ in XPath genau den gleichen Wert wie der Operator „%“ in Java berechnet. In Java berechnet der Operator „%“ den Rest der Division zweier Gleitkommazahlen im Dualzahlsystem. Beim Konvertieren in das gewünschte Zahlenformat entsteht ein kleiner Fehler, so dass der Wert, der im Dualzahlsystem berechnet wird, ungleich dem Wert ist, der sich bei der Berechnung im Dezimalzahlsystem ergibt.

Die Modellierung der Berechnung der Modulo-Operation im Dualzahlsystem ist sehr aufwändig. Dagegen ist der erwähnte Fehler recht gering. Deshalb wird im formalen Modell der Fehler vernachlässigt und die Modulo-Operation im Dezimalzahlsystem berechnet. Da die Modulo-Operation den Rest einer Division berechnet, können wir die Intervallgrenzen für das Ergebnis der Modulo-Operation auf Intervallen leicht berechnen. Ist der Dividend immer positiv, ist der Rest immer größer oder gleich Null und kleiner oder gleich des Dividenden. Ist der Dividend immer negativ, ist der Rest immer größer oder gleich dem Dividenden und kleiner oder gleich Null. Kann der Dividend sowohl positiv als auch negativ sein, entspricht der Rest minimal bzw. maximal dem Dividenden.

**Definition 12 (Modulo-Operation auf Intervallen)**

Die Funktion  $\bigcirc_{mod} : L_1 \times L_1 \longrightarrow L_1$  ist wie folgt definiert:

$$x \bigcirc_{mod} y =_{def} \begin{cases} ([x_1, x_2] \bigcirc_{mod} [y_1, y_2]) \cup X \cup Y, & \text{für } x = [x_1, x_2] \cup X \wedge y = [y_1, y_2] \cup Y; \\ [ ] \cup X \cup Y, & \text{für } x = [ ] \cup X \vee y = [ ] \cup Y. \end{cases}$$

mit:

$$[x_1, x_2] \bigcirc_{mod} [y_1, y_2] = \begin{cases} [0, x_2], & \text{wenn } x_1, x_2 \text{ positiv;} \\ [x_1, 0], & \text{wenn } x_1, x_2 \text{ negativ;} \\ [x_1, x_2], & \text{wenn } x_2 \text{ positiv und } x_1 \text{ negativ.} \end{cases}$$

┘

Schließlich definieren wir noch die Funktion „<Int“, „>Int“, „=Int“, „!=Int“ für die Vergleichsoperationen „<“ (Kleiner), „>“ (Größer), „=“ (Gleich) und „!=“ (Ungleich) aus XPath. Die Vergleichsoperationen „<=“ (Kleiner oder Gleich) und „>=“ (Größer oder Gleich) beschreiben wir nicht im formalen Modell, da sie auf die anderen Vergleichsoperationen und die Disjunktion boolescher Werte zurückgeführt werden können. Bei allen Vergleichsoperationen ist zu beachten, dass ein Vergleich mit dem speziellen Wert *NaN* immer falsch ist.

Die Vergleichsoperationen werden auf die Zahlenwerte angewendet, die durch die Intervalle repräsentiert werden. Zwei Intervalle können somit zwar gleich sein, der Vergleich der repräsentierten Werte kann aber auch falsch sein. Betrachten wir beispielsweise einen Vergleich zwischen den Intervallen [1, 5] und [1, 5]. Beide Intervalle sind zwar gleich, aber die repräsentierten Werte 2 und 4 sind nicht gleich. Somit kann die Gleichheit der konkreten Zahlenwerte sowohl wahr, z.B. bei 1 = 1 oder 2 = 2, als auch falsch, z.B. bei 1 = 2 oder 2 = 4, sein.

Ein Intervall *x* ist kleiner als ein Intervall *y*, wenn das Intervall *x* eine Zahl repräsentiert, die immer kleiner der Zahl ist, die vom Intervall *y* repräsentiert wird. Repräsentiert das Intervall *x* eine Zahl, die immer größer oder gleich der Zahl ist, die vom Intervall *y* repräsentiert wird, ist das Intervall *x* nie kleiner als das Intervall *y*. Sonst kann ein Vergleich der Zahlenwerte, die durch die beiden Intervalle repräsentiert werden, sowohl wahr als auch falsch sein.

**Definition 13 (Vergleichsoperation „<“ für Intervalle)**

Die Funktion  $<_{Int} : L_1 \times L_1 \longrightarrow B_1$  ist wie folgt definiert:

$$x <_{Int} y =_{def} \begin{cases} \{false\}, & \text{für } x = [ ] \vee y = [ ] \vee x = [ ] \cup NaN \vee y = [ ] \cup NaN; \\ x < y & \text{für } x \neq [ ] \wedge y \neq [ ] \wedge X \neq NaN \wedge Y \neq NaN; \\ x < y \cup \{false\} & \text{für } x \neq [ ] \wedge y \neq [ ] \wedge (X = NaN \vee Y = NaN). \end{cases}$$

mit:

$$[x_1, x_2] < [y_1, y_2] = \begin{cases} \{true\}, & \text{wenn } y_1 > x_2; \\ \{false\}, & \text{wenn } x_1 \geq y_2; \\ \{true, false\}, & \text{sonst.} \end{cases}$$

┘

Ein Intervall  $x$  ist größer als ein Intervall  $y$ , wenn das Intervall  $x$  eine Zahl repräsentiert, die immer größer der Zahl ist, die vom Intervall  $y$  repräsentiert wird. Repräsentiert das Intervall  $x$  eine Zahl, die immer kleiner oder gleich der Zahl ist, die vom Intervall  $y$  repräsentiert wird, ist das Intervall  $x$  nie größer als das Intervall  $y$ . Sonst kann ein Vergleich der Zahlenwerte, die durch die beiden Intervalle repräsentiert werden, sowohl wahr als auch falsch sein.

**Definition 14 (Vergleichsoperation „>“ für Intervalle)**

Die Funktion  $>_{Int} : L_1 \times L_1 \longrightarrow B_1$  ist wie folgt definiert:

$$x >_{Int} y =_{def} \begin{cases} \{false\}, & \text{für } x = [] \vee y = [] \vee x = [] \cup NaN \vee y = [] \cup NaN; \\ x > y & \text{für } x \neq [] \wedge y \neq [] \wedge X \neq NaN \wedge Y \neq NaN; \\ x > y \cup \{false\} & \text{für } x \neq [] \wedge y \neq [] \wedge (X = NaN \vee Y = NaN). \end{cases}$$

mit:

$$[x_1, x_2] > [y_1, y_2] = \begin{cases} \{true\}, & \text{wenn } x_1 > y_2; \\ \{false\}, & \text{wenn } y_1 \geq x_2; \\ \{true, false\}, & \text{sonst.} \end{cases}$$

┘

Ein Intervall  $x$  ist gleich dem Intervall  $y$ , wenn beide Intervalle genau eine Zahl repräsentieren. Repräsentiert das Intervall  $x$  Zahlen, die nicht vom Intervall  $y$  repräsentiert werden, sind die beiden konkreten Datenwerte nie gleich. Sonst kann ein Vergleich der konkreten Zahlenwerte, die durch die beiden Intervalle repräsentiert werden, sowohl wahr als auch falsch sein.

**Definition 15 (Vergleichsoperation „=“ für Intervalle)**

Die Funktion  $=_{Int} : L_1 \times L_1 \longrightarrow B_1$  ist wie folgt definiert:

$$x =_{Int} y =_{def} \begin{cases} \{false\}, & \text{für } x = [] \vee y = [] \vee x = [] \cup NaN \vee y = [] \cup NaN; \\ x = y & \text{für } x \neq [] \wedge y \neq [] \wedge X \neq NaN \wedge Y \neq NaN; \\ x = y \cup \{false\} & \text{für } x \neq [] \wedge y \neq [] \wedge (X = NaN \vee Y = NaN). \end{cases}$$

mit:

$$[x_1, x_2] = [y_1, y_2] = \begin{cases} \{true\}, & \text{wenn } x_1 = x_2 = y_1 = y_2; \\ \{false\}, & \text{wenn } (y_1 > x_2) \vee (y_2 < x_1); \\ \{true, false\}, & \text{sonst.} \end{cases}$$

┘

Ein Intervall  $x$  ist ungleich dem Intervall  $y$ , wenn das Intervall  $x$  Zahlen repräsentiert, die nicht vom Intervall  $y$  repräsentiert werden. Repräsentieren beide Intervalle genau eine Zahl, sind die beiden konkreten Zahlenwerte niemals ungleich. Sonst kann ein Vergleich der Zahlenwerte, die durch die beiden Intervalle repräsentiert werden, sowohl wahr als auch falsch sein.

**Definition 16 (Vergleichsoperation „!“ für Intervalle)**

Die Funktion  $! =_{Int} : L_1 \times L_1 \rightarrow B_1$  ist wie folgt definiert:

$$x ! =_{Int} y =_{def} \begin{cases} \{false\}, & \text{für } x = [] \vee y = [] \vee x = [] \cup NaN \vee y = [] \cup NaN; \\ x! = y & \text{für } x \neq [] \wedge y \neq [] \wedge X \neq NaN \wedge Y \neq NaN; \\ x! = y \cup \{false\} & \text{für } x \neq [] \wedge y \neq [] \wedge (X = NaN \vee Y = NaN). \end{cases}$$

mit:

$$[x_1, x_2] ! = [y_1, y_2] = \begin{cases} \{true\}, & \text{wenn } (y_1 > x_2) \vee (y_2 < x_1); \\ \{false\}, & \text{wenn } x_1 = x_2 = y_1 = y_2; \\ \{true, false\}, & \text{sonst.} \end{cases}$$

⌋

**3.1.3. Der Datentyp string**

Zeichenketten in XPath bestehen aus einer Folge von null oder mehr Zeichen. Ein Zeichen ist dabei ein einzelnes abstraktes Unicode-Zeichen (siehe Kapitel 2.3.3). Für die Abstraktion von konkreten Zeichenketten, entwickeln wir im Folgenden eine Galois Korrespondenz.

**Eine Galois Korrespondenz für Zeichenketten**

Zunächst definieren wir die Menge aller Zeichen, die in Zeichenketten vorkommen können. Wie in Kapitel 2.3.3 erwähnt, entspricht ein XPath-Zeichen einem einzelnen abstrakten Unicode-Zeichen:

$$Z =_{def} \text{Menge aller Unicode-Zeichen}$$

Nun können wir die Sprache  $S$  definieren. Die Sprache  $S$  bildet die Menge aller Wörter, die mit Hilfe der Zeichen aus  $Z$  gebildet werden können. Weiterhin beinhaltet die Sprache  $S$  das leere Wort  $\varepsilon$ .

$$S =_{def} \{ uv \mid u \in Z \cup \{\varepsilon\} \wedge v \in S \cup \{\varepsilon\} \}$$

Die Menge  $S$  ist endlich, da die Menge aller Zeichen endlich ist und wir davon ausgehen können, dass nur endliche Zeichenketten existieren. Ein unendlich langes XML-Dokument können wir ausschließen. Dementsprechend muss jede beliebige Zeichenkette in einem XML-Dokument endlich sein. O.B.d.A. nehmen wir also an, dass  $S$  nur endlich viele Wörter enthält. Formal:<sup>6</sup>

$$card(S) < \infty$$

Die Länge  $l$  eines Wortes  $v \in S$  ermittelt sich aus der Anzahl von Zeichen im Wort. Es gilt also:  $l(v_1 \dots v_n) = n$ . Die Länge des leeren Wortes ist 0. O.B.d.A. gilt:

$$\forall s \in S : l(s) < \infty$$

---

<sup>6</sup>  $card$  berechnet die Cardinalität einer Menge.

Die Potenzmenge von  $S$  beschreibt alle möglichen Wörter, die in einer Zeichenkette vorkommen dürfen. Deshalb definieren wir für die Potenzmenge von  $S$  einen vollständigen Verband  $S_0$ :

**Definition 17 (vollständiger Verband für Zeichenketten)**

Für die Potenzmenge von  $S$  definieren wir den vollständigen Verband  $S_0$  wie folgt:

$$S_0 =_{\text{def}} (\mathcal{P}(S), \subseteq, \sqcup, \sqcap, \emptyset, S) \text{ mit}$$

$$\begin{aligned} \mathcal{P}(S) & - \text{Potenzmenge von } S \\ \subseteq & - \text{Teilmengenbeziehung} \\ \sqcup & = \bigcup Y \text{ für } Y \subseteq \mathcal{P}(S) \\ \sqcap & = \bigcap Y \text{ für } Y \subseteq \mathcal{P}(S) \end{aligned}$$

┘

In unserem formalen Modell abstrahieren wir bei Zeichenketten von der Reihenfolge und der Anzahl der einzelnen Zeichen. Somit bilden wir jede Zeichenkette auf die enthaltenen Zeichen ab. Zur Abstraktion definieren wir einen Verband  $S_1$  für die Potenzmenge über alle Zeichen aus  $Z$ :

**Definition 18 (vollständiger Verband für Zeichen)**

Für die Potenzmenge von  $Z$  definieren wir den vollständigen Verband  $S_1$  wie folgt:

$$S_1 =_{\text{def}} (\mathcal{P}(Z), \subseteq, \sqcup_Z, \sqcap_Z, \emptyset, Z) \text{ mit}$$

$$\begin{aligned} \mathcal{P}(Z) & - \text{Potenzmenge von } Z \\ \subseteq & - \text{Teilmengenbeziehung} \\ \sqcup_Z & = \bigcup Y \text{ für } Y \subseteq \mathcal{P}(Z) \\ \sqcap_Z & = \bigcap Y \text{ für } Y \subseteq \mathcal{P}(Z) \end{aligned}$$

┘

Nun definieren wir noch die Repräsentationsfunktion  $\beta_{\text{Zeichenketten}} : S \rightarrow \mathcal{P}(Z)$ , die für ein Wort der Sprache  $S$  die enthaltenen Zeichen ermittelt:

$$\beta_{\text{Zeichenketten}}(s) =_{\text{def}} \begin{cases} \emptyset, & \text{wenn } s = \varepsilon; \\ \{ z \mid z = s_i \text{ für } i \in (1 \dots n) \wedge z \in Z \}, & \text{wenn } s = s_1 \dots s_n. \end{cases}$$

Die Funktion  $\beta_{\text{Zeichenketten}}$  ermittelt eine Menge. Für die leere Zeichenkette liefert die Funktion  $\beta_{\text{Zeichenketten}}$  die leere Menge. Für eine Zeichenkette mit mindestens einem Zeichen, wird eine Menge geliefert, die jedes Zeichen in der Zeichenkette genau einmal enthält. Beispielsweise liefert die Funktion für die Zeichenkette „Hallo!“ die Menge  $\{‘a’, ‘H’, ‘l’, ‘o’, ‘!’\}$ . Das bedeutet, dass wir die Anzahl der einzelnen Zeichen in der Zeichenkette in unserem Datenmodell vernachlässigen. Da eine Menge ungeordnet ist, vernachlässigen wir auch die Reihenfolge der einzelnen Zeichen.

Schließlich definieren wir die Abstraktionsfunktion  $\alpha_{\text{Zeichenketten}} : S_0 \longrightarrow S_1$  und die Konkretisierungsfunktion  $\gamma_{\text{Zeichenketten}} : S_1 \longrightarrow S_0$  unter Verwendung der Repräsentationsfunktion  $\beta_{\text{Zeichenketten}}$ :

$$\alpha_{\text{Zeichenketten}}(Y) =_{\text{def}} \sqcup_Z \{ \beta_{\text{Zeichenketten}}(x) \mid x \in Y \} \text{ mit } Y \subseteq S$$

$$\gamma_{\text{Zeichenketten}}(l) =_{\text{def}} \{ x \in S \mid \beta_{\text{Zeichenketten}}(x) \sqsubseteq_Z l \} \text{ mit } l \in S_1$$

Nach Satz A.1 bildet  $(S_0, \alpha_{\text{Zeichenketten}}, \gamma_{\text{Zeichenketten}}, S_1)$  eine Galois Korrespondenz. Somit ist unsere Abstraktion für Zeichenketten korrekt.

### Operationen auf Zeichenketten

Die Zeichenmengen aus  $S_1$  repräsentieren Zeichenketten in XPath-Ausdrücken. Für solche Zeichenketten sind in XPath die Vergleichsoperationen „=“ (Gleich) und „!=“ (Ungleich) definiert. Um diese Operationen formal zu repräsentieren, definieren wir im Folgenden die Funktionen „= $_Z$ “ und „!= $_Z$ “ auf Grundlage des vollständigen Verbands  $S_1$ . Die Funktionen beschreiben somit den Vergleich von abstrakten Zeichenketten.

Die Gleichheit zweier Zeichenketten wird auf Grundlage der Anzahl aller Zeichen sowie deren Reihenfolge in den beiden Zeichenketten entschieden. Der Vergleich ist wahr, wenn beide abstrakten Zeichenketten keine Zeichen enthalten. In diesem Fall repräsentieren die beiden abstrakten Zeichenketten ausschließlich das leere Wort. Der Vergleich ist falsch, wenn beide abstrakten Zeichenketten keine gemeinsamen Zeichen besitzen. In diesem Fall repräsentieren die beiden abstrakten Zeichenketten immer paarweise unterschiedliche konkrete Zeichenketten. Da wir in unserer Abstraktion die Anzahl und die Reihenfolge der einzelnen Zeichen vernachlässigen, können wir in jedem anderen Fall nur sagen, dass der Vergleich sowohl wahr als auch falsch sein kann.

#### Definition 19 (Vergleichsoperation „= $_Z$ “ für Zeichenketten)

Die Funktion  $=_Z : S_1 \times S_1 \longrightarrow B_1$  ist wie folgt definiert:

$$ZK_1 =_Z ZK_2 =_{\text{def}} \begin{cases} \{true\}, & \text{für } ZK_1 = ZK_2 = \emptyset; \\ \{false\}, & \text{für } ZK_1 \cap ZK_2 = \emptyset \wedge ZK_1 \neq \emptyset \wedge ZK_2 \neq \emptyset; \\ \{true, false\} & \text{sonst.} \end{cases}$$

┘

Die Ungleichheit zweier Zeichenketten wird analog zur Gleichheit auf Grundlage der Anzahl aller Zeichen sowie deren Reihenfolge in den beiden Zeichenketten entschieden. Der Vergleich ist wahr, wenn beide abstrakten Zeichenketten keine gemeinsamen Zeichen besitzen. Der Vergleich ist falsch, wenn beide abstrakten Zeichenketten ausschließlich das leere Wort repräsentieren. Da wir in unserer Abstraktion die Anzahl und die Reihenfolge der einzelnen Zeichen vernachlässigen, können wir in jedem anderen Fall nur sagen, dass der Vergleich sowohl wahr als auch falsch sein kann.

**Definition 20 (Vergleichsoperation „!“ für Zeichenketten)**

Die Funktion  $! =_Z : S_1 \times S_1 \rightarrow B_1$  ist wie folgt definiert:

$$ZK_1 ! =_Z ZK_2 =_{def} \begin{cases} \{true\}, & \text{für } ZK_1 \cap ZK_2 = \emptyset \wedge ZK_1 \neq \emptyset \wedge ZK_2 \neq \emptyset; \\ \{false\}, & \text{für } ZK_1 = ZK_2 = \emptyset; \\ \{true, false\} & \text{sonst.} \end{cases}$$

┘

### 3.1.4. Der Datentyp node-set

Der Datentyp `node-set` repräsentiert eine Knotenmenge. Dabei ist ein Knoten ein XML-Element oder ein Attribut eines XML-Elements. Diese XML-Elemente bzw. Attribute sind in einem oder mehreren XML-Instanzdokumenten enthalten, dessen Typen mit Hilfe von XML-Schema beschrieben werden. Da wir einen BPEL-Prozess betrachten, ist im Normalfall nur das XML-Schema bekannt, nicht aber das Instanzdokument. Deshalb repräsentieren wir im formalen Modell ein XML-Element bzw. Attribut durch das zugehörige XML-Schema und die im Schema enthaltene Deklaration des XML-Elements bzw. Attributs. Dabei repräsentieren wir alle Informationen aus dem XML-Schema, die für unsere Datenanalyse benötigt werden.

Ein XML-Schema repräsentieren wir im formalen Modell als Baum. Diese Repräsentation eignet sich aufgrund der hierarchischen Struktur eines XML-Schemas besonders gut. Die Knoten eines Baums stellen dabei die im XML-Schema deklarierten Elemente bzw. Attribute dar. Die Blattknoten im Baum repräsentieren Attribute oder Elemente mit einfachem Typ. Jeder Knoten im Baum, der Kindknoten besitzt, repräsentiert ein Element mit komplexem Typ.

Ein Baum ist ein spezieller gerichteter, azyklischer Graph. Ein XML-Element bzw. Attribut wird im Baum durch einen Knoten repräsentiert, der einer natürlichen Zahl entspricht. Die Kindelemente eines XML-Elements werden durch Kindknoten des korrespondierenden Knotens im Baum repräsentiert. Eine Kante im Baum führt immer von einem Elterknoten zu einem Kindknoten.

$$G =_{def} (V, E) \text{ mit}$$

$$V \subseteq \mathbb{N} \text{ und}$$

$$E \subseteq V \times V.$$

Da  $G$  ein Baum ist, existiert genau ein Knoten in  $G$ , der keinen Elterknoten hat. Dieser Knoten ist der Wurzelknoten des Baums  $G$ . Jeder andere Knoten hat genau einen Elterknoten. Formal gilt deshalb folgende Eigenschaft für  $G$ :

$$\begin{aligned} \exists! w \in V \forall e \in E \forall x \in V : e \neq (x, w) \wedge \\ \forall v \in V, v \neq w \exists e \in E : e = (x, v) \wedge \nexists e' \in E : e \neq e' \wedge e' = (y, v). \end{aligned}$$

In unserer Datenanalyse benötigen wir verschiedene Informationen über die XML-Elemente bzw. -Attribute. So interessiert uns der Name, Datentyp, die Häufigkeitsbeschränkungen sowie Konstantenwerte für XML-Elemente bzw. -Attribute. Im Folgenden beschreiben wir, wie der Name, die Datentypinformation sowie die Häufigkeitsbeschränkungen im Baum repräsentiert werden. Im Anschluss beschreiben wir einen Algorithmus, der die entsprechenden Informationen sowie Konstantenwerte aus dem XML-Schema in den Baum überführt.

Den Namen eines XML-Elements bzw. Attributs repräsentieren wir als Tupel. Dieses Tupel enthält als erstes Element den Namen des korrespondierenden XML-Elements bzw. Attributs ohne Präfix. Als zweites Element enthält das Tupel den URI des Namensraumes, in dem sich das korrespondierende XML-Element bzw. Attribut befindet. Um einem Knoten im Baum den entsprechenden Namen zuzuordnen, annotieren wir den Baum mit Hilfe einer Relation *Name*:<sup>7</sup>

$$G_{benannt} =_{\text{def}} (G, \textit{Name}) \text{ mit}$$

$$\textit{Name} \subseteq V \times (S \times S)$$

Weiterhin benötigen wir in unserer Datenanalyse die Information, ob ein Knoten im Baum ein XML-Element, Attribut oder einen Namensraum repräsentiert. Deshalb annotieren wir die Knoten im Baum um diese Information. Formal repräsentieren wir die Information durch eine Relation *Typ*, die jedem Knoten ein Element aus der Menge  $\{A, E, N\}$  zuordnet. Ein Knoten, dem das Element  $A$  zugeordnet ist, repräsentiert ein Attribut. Ein Knoten, dem das Element  $E$  zugeordnet ist, repräsentiert ein Element. Ein Knoten, dem das Element  $N$  zugeordnet ist, repräsentiert einen Namensraum.

Da wir zusätzlich die Information des Datentyps eines XML-Elements bzw. Attributs in unserer Datenanalyse benötigen, erweitern wir die Relation *Typ* um die Datentypinformation. Der einfache Datentyp eines XML-Elements bzw. Attributs wird entweder auf den XPath-Datentyp `boolean`, `numbers` oder `string` abgebildet. Diese Datentypen repräsentieren wir gerade durch die vollständigen Verbände  $B_1, L_1$  und  $S_1$ . Für die Repräsentation eines komplexen Typs im Baum führen wir die Symbole  $\Lambda$  und  $\Omega$  ein. Ein komplexer Typ für den das mixed-Attribut den Wert *false* hat, wird durch ein  $\Lambda$  repräsentiert. Ein komplexer Typ für den das mixed-Attribut den Wert *true* hat, wird durch ein  $\Omega$  repräsentiert. Einem Knoten wird nun, je nach Datentyp, ein Element der Verbände  $B_1, L_1$  oder  $S_1$  oder das Symbol  $\Lambda$  zugeordnet.

---

<sup>7</sup>S beschreibt hier die Menge aller Zeichenketten wie in Kapitel 3.1.3 definiert.

Formal erweitern wir den Graph  $G$  um die Relation  $Typ$  wie folgt:

$$G_{getypt} =_{\text{def}} (G_{benannt}, Typ) \text{ mit} \\ Typ \subseteq V \times \{‘A’, ‘E’, ‘N’\} \times (L_1 \cup S_1 \cup B_1 \cup \{\Lambda, \Omega\})$$

Häufigkeitsbeschränkungen für XML-Elemente bzw. -Attribute geben die minimale und maximale Anzahl der XML-Elemente bzw. Attribute vor. Im Baum repräsentieren wir diese Information mit Hilfe von Intervallen. Dabei erweitern wir den Baum um eine Relation  $Card$ , die einem Knoten ein Intervall zuordnet, das die minimale und maximale Anzahl des korrespondierenden XML-Elements bzw. -Attributs repräsentiert.

$$G_{annotiert} =_{\text{def}} (G_{getypt}, Card) \text{ mit} \\ Card \subseteq V \times \text{Zahlenintervalle}$$

Ein Datum vom Typ **node-set** ist eine Menge von XML-Elementen bzw. Attributen, die durch ein oder mehrere XML-Schemata beschrieben werden. Ein XML-Element bzw. Attribut, das durch ein XML-Schema beschrieben wird, ist ein ausgezeichneter Knoten  $x$  im entsprechenden Baum  $y$ . Um ein betrachtetes XML-Element bzw. Attribut zu einem XML-Schema zuzuordnen, verwenden wir das Tupel  $(x, y)$ . Ein Datum vom Typ **node-set** ist demnach eine Menge von Tupeln  $(x, y)$ .

Ein Datum vom Typ **node-set** repräsentieren wir im formalen Modell also als Menge von annotierten Bäumen:

**Definition 21 (formale Knotenmenge)**

*Eine Knotenmenge definieren wir formal als:*

$$K =_{\text{def}} \{ (x, G_{annotiert}) \mid \text{Lokalisierungspfad liefert ein Datum } x, \text{ das durch ein} \\ \text{XML-Schema beschrieben ist, welches durch den Baum} \\ G_{annotiert} \text{ repräsentiert wird.} \}$$

┘

Die Kardinalität dieser Menge berechnet sich aus den Annotationen der Knoten  $x$  in den entsprechenden Bäumen  $y$ . Da die maximale Anzahl eines XML-Elements bzw. Attributs unbeschränkt sein kann, bedeutet das, dass die Kardinalität unendlich sein kann. Allerdings können wir ein unendlich langes Instanzdokument ausschließen. Aus diesem Grund können wir o.B.d.A. annehmen, dass die Kardinalität der Menge  $K$  endlich ist.

Der Zustandsraum der Menge  $K$  für ein Datum vom Typ **node-set** wird durch die XML-Schemata der in  $K$  enthaltenen Elemente beschrieben. Ein XML-Schema legt fest, welche Elemente vorkommen dürfen und in welcher Anzahl die Elemente auftreten dürfen. Somit legt ein XML-Schema die Anzahl der Knoten im entsprechenden Baum  $G_{annotiert}$  fest. Im Baum  $G_{annotiert}$  werden Elemente mit einem einfachen Typ durch Blattknoten repräsentiert, die mit dem jeweiligen Typ annotiert sind. Elemente mit komplexen Typen werden durch Knoten mit Kindknoten repräsentiert. Der Zustandsraum eines komplexen Elements berechnet sich somit aus den enthaltenen Elementen mit einfachem Typ.

## Repräsentation eines XML-Schemas

Um die Informationen eines XML-Schemas in einem Baum  $G_{annotiert}$  zu repräsentieren, müssen wir das XML-Schema analysieren. Dabei beginnen wir mit dem Schema-Element, das wir durch den Wurzelknoten repräsentieren. Weiterhin repräsentiert der Wurzelknoten ein Instanzdokument des XML-Schemas zur Laufzeit. Die Elemente, die im Instanzdokument enthalten sind, werden durch die Kindelemente des Wurzelknotens repräsentiert.

Ausgehend vom Schema-Element traversieren wir über das XML-Schema, wobei wir eine Tiefensuche anwenden. Bei der Traversierung wird für jedes XML-Element bzw. -Attribut ein Knoten im Baum erzeugt. Hat ein XML-Element Kindelemente oder Attribute, so werden die Elter-Kind-Beziehungen durch Kanten zwischen entsprechenden Knoten im Baum dargestellt. Wird im XML-Schema ein Namensraum definiert, so wird dieser durch Knoten im Baum repräsentiert. Dabei wird für jedes XML-Element dieses Namensraumes ein eigener Knoten erzeugt, der den Namensraum repräsentiert. Dieser Namensraumknoten ist dann ein Kindknoten des Knotens, der das betrachtete XML-Element repräsentiert. Dabei ist zu beachten, dass Namensräume auf Kindelemente vererbt werden. Sollte also für ein XML-Element kein Namensraum explizit spezifiziert sein, müssen die Vorfahren auf Namensraumspezifikationen untersucht werden.

Ein XML-Element darf maximal einem Namensraum zugordnet sein. Allerdings kann ein XML-Element mehrere Namensräume bzw. einen Namensraum mit verschiedenen Präfixen definieren. Dies wird z.B. verwendet, wenn Kindelemente einem anderen Namensraum zugeordnet werden sollen, der Namensraum aber nicht auf der Kindebene definiert wird. In diesem Fall hat der Knoten im Baum, der das XML-Element mit mehreren Namensraumdefinitionen bzw. Präfixdefinitionen repräsentiert, genau so viele Namensraumknoten als Kinder wie im XML-Element definiert werden.

Bei der Erzeugung eines Knotens wird dieser zusätzlich mit den Informationen annotiert, die in unserer Datenanalyse benötigt werden. Für einen Knoten, dessen Element mit Hilfe einer Referenz deklariert wird, annotieren wir dabei die Informationen des referenzierten Elements. Im Folgenden beschreiben wir, wie die Relationen *Name*, *Typ* und *Card* aus dem XML-Schema ermittelt werden können.

Die Relation *Name* wird definiert, indem jedem Knoten ein Tupel  $(x,y)$  zugeordnet wird. Für einen Knoten, der ein XML-Element bzw. Attribut repräsentiert ist  $x$  der Inhalt des `name`-Attributs. Für einen Knoten, der ein Attribut repräsentiert, ist  $y$  die leere Menge. Für ein XML-Element wird in  $y$  der Namensraum-URI gespeichert. Dabei ist zu beachten, dass der Namensraum-URI auf Kindelemente vererbt wird. Wird der Namensraum-URI eines XML-Elements durch ein Präfix angegeben, so wird der dem Präfix zugeordnete Namensraum-URI im XML-Schema ermittelt und im Tupel gespeichert. Wird der Namensraum-URI durch das `xmlns`-Attribut spezifiziert, so wird der Wert, der dem `xmlns`-Attribut zugewiesen ist, in  $y$  gespeichert. Ist kein Namensraum für das betrachtete XML-Element angegeben, ist der Namensraum-URI leer. Einem Knoten

im Baum, der einen Namensraum repräsentiert, wird mit Hilfe der Relation *Name* das Tupel  $(x, \emptyset)$  zugeordnet. Dabei ist  $x$  das Präfix, das dem Namensraum zugeordnet ist.

Um die Relation *Typ* zu definieren, wird zunächst jedem Knoten, der ein XML-Element repräsentiert, ein 'E' zugeordnet bzw. jedem Knoten, der ein XML-Attribut repräsentiert, ein 'A' zugeordnet. Einem Knoten, der einen Namensraum repräsentiert, wird ein 'N' zugeordnet. Weiterhin wird in der Relation *Typ* der Datentyp für jeden Knoten folgendermaßen annotiert:

Für einen Knoten, der ein XML-Element mit einem komplexen Typ repräsentiert, müssen wir folgende Fallunterscheidung vornehmen: Ist der Wert des mixed-Attribut für den komplexen Typ *true*, wird der Datentyp mit  $\Omega$  annotiert. Ist der Wert des mixed-Attributs *false*, überprüfen wir noch ob der komplexe Typ durch Ableitung definiert wurde. Wurde der komplexe Typ ohne Ableitung oder durch Ableitung von einem anderen komplexen Typ definiert, annotieren wir den Datentyp mit  $\Lambda$ . Wurde der komplexe Typ durch Ableitung von einem einfachen Typ definiert, wird der Datentyp wie für ein XML-Element mit einem einfachen Typ annotiert.

Ein Knoten, der ein XML-Element mit einem einfachen Typ oder ein XML-Attribut repräsentiert, wird der Datentyp folgendermaßen annotiert:

- Konstantenwerte werden mit Hilfe der entsprechenden Repräsentationsfunktion in ihre Abstraktion überführt. Welche Repräsentationsfunktion verwendet wird, hängt vom betrachteten Typ ab. Diese Abstraktion wird dann in der Relation *Typ* annotiert.
- Für die vordefinierten einfachen Typen in XML-Schema werden die zugeordneten Abstraktionen in der Relation *Typ* annotiert, wie in Tabelle 3.1 dargestellt.
- Wird ein neuer Zahlentyp unter Verwendung von *min-inclusive*, *max-inclusive*, *min-exclusive* oder *max-exclusive* definiert, werden die angegebenen Werte als Intervallgrenzen verwendet. Der angegebene minimale Wert bildet die linke Intervallgrenze, der maximale Wert bildet die rechte Intervallgrenze. So wird z.B. der Typ *PLZ* von Seite 21 (Kapitel 2.2.3) auf das Intervall  $[10000, 99999]$  abgebildet. Bei Angabe von *min-exclusive* und *max-exclusive* verwenden wir die angegebenen Werte als Intervallgrenzen, da wir keine offene Intervalle angeben können. Handelt es sich um Gleitkommazahlen, so können wir die konkrete minimale bzw. maximale Zahl nicht berechnen. Wir verwenden die angegebenen Werte, um in der Abstraktion keine darstellbare Zahl zu verlieren.
- Wird der einfache Typ durch Angabe eines regulären Ausdrucks definiert, wird jedes Zeichen, das sowohl im regulären Ausdruck als auch im Typ, von dem abgeleitet wird, enthalten ist, mit Hilfe der entsprechenden Repräsentationsfunktion in ihre Abstraktion überführt. Welche Repräsentationsfunktion verwendet wird, hängt von dem Typ ab, von dem der betrachtete Typ abgeleitet ist. Beispielsweise wird der einfache Typ *Schulnoten* von Seite 22 (Kapitel 2.2.3) vom Typ *xsd:string* abgeleitet. In diesem Fall wird also die Repräsentationsfunktion  $\beta_{\text{Zeichenketten}}$  verwen-

det. Schließlich werden die Abstraktionen aller Zeichen, die im regulären Ausdruck enthalten sind, vereinigt. Die Vereinigung wird dann in der Relation *Typ* annotiert.

- Bei einem Aufzählungstyp werden die aufgezählten Werte ebenfalls mit Hilfe der entsprechenden Repräsentationsfunktion in ihre Abstraktion überführt. Welche Repräsentationsfunktion verwendet wird, hängt wieder von dem Typ ab, von dem der Aufzählungstyp abgeleitet ist. Die Vereinigung der Abstraktionen für alle Aufzählungswerte wird schließlich in der Relation *Typ* annotiert.
- Bei Listentypen berechnen wir die Abstraktion des Typs, der im Attribut `itemType` angegeben ist. Diese Abstraktion wird in der Relation *Typ* annotiert.
- Bei Vereinigungstypen berechnen wir die Abstraktionen der einzelnen Typen, die im Attribut `memberTypes` angegeben sind. Danach werden zunächst die Typen vereinigt, die durch die gleiche Abstraktion repräsentiert werden. Die Vereinigung wird dabei mit Hilfe der kleinsten oberen Schranken für die Abstraktionen gebildet. Schließlich vereinigen wir noch die unterschiedlichen Abstraktionen.

Nehmen wir beispielsweise an, dass in einem Vereinigungstyp zwei unterschiedliche Zahlenbereiche und ein Zeichenkettentyp vereinigt werden. Dann ermitteln wir zuerst die Intervalle der Zahlenbereiche und die Abstraktion für den Zeichenkettentyp. Danach bilden wir die Vereinigung der Intervalle mit Hilfe der Operation  $\sqcup_{Int}$ . Schließlich bilden wir eine Vereinigungsmenge, die sowohl das Ergebnis Intervallvereinigung als auch die Abstraktion des Zeichenkettentyps enthält.

Den XML-Schema-Datentyp `xsd:anyType` müssen wir gesondert betrachten. Ein XML-Element bzw. Attribut dieses Typs kann jeden Datentyp annehmen, der in einem XML-Schema definiert wird. Da wir keinerlei Informationen über den Datentyp haben, speichern wir ihn in der Relation *Typ* als Menge

$$\{[-\infty, \infty] \cup NaN\} \cup Z \cup \{true, false\} \cup \{\Lambda, \Omega\}.$$

In dieser Menge ist jeder einfache und komplexe Datentyp enthalten. Den XML-Schema-Datentyp `xsd:anySimpleType` speichern wir analog in der Relation *Typ* als Menge

$$\{[-\infty, \infty] \cup NaN\} \cup Z \cup \{true, false\}.$$

Für einen Namensraumknoten repräsentieren wir im Datentyp den Namensraum-URI. Deshalb wird der Namensraum-URI mit Hilfe der Funktion  $\beta_{Zeichenketten}$  in eine Teilmenge von *Z* überführt. Diese Teilmenge wird dann für den Namensraumknoten in der Relation *Typ* als Datentyp gespeichert.

Schließlich müssen wir noch die Relation *Card* definieren. Für jeden Knoten, der ein XML-Element repräsentiert, wird überprüft, ob das XML-Schema eine Spezifikation der `minOccurs`- und `maxOccurs`-Werte enthält. Falls keine Werte spezifiziert sind, werden die Vorgabewerte für die Annotation verwendet. In diesem Fall wird also das Intervall  $[1, 1]$  in der Relation *Card* annotiert. Ist eine Häufigkeitsbeschränkung spezifiziert, werden die angegebenen Werte für die Annotation verwendet. Beispielsweise wird für das Element

**Tabelle 3.1.:** Abstraktion der vordefinierten einfachen Typen aus XML-Schema. Die Abstraktion enthält alle Zahlen, booleschen Werte bzw. Zeichen, die in einem konkreten Wert des vordefinierten einfachen Typs vorkommen dürfen.

einfacher Typ	Abstraktion
byte	$[-128, 127]$
unsignedByte	$[0, 127]$
base64Binary	$[-\infty, \infty]$
hexBinary	$[-\infty, \infty]$
integer, decimal	$[-\infty, \infty]$
positiveInteger	$[1, \infty]$
negativeInteger	$[-\infty, -1]$
NonNegativeInteger	$[0, \infty]$
NonPositiveInteger	$[-\infty, 0]$
int	$[-2147483648, 2147483647]$
unsignedInt	$[0, 4294967295]$
long	$[-9223372036854775808, 9223372036854775807]$
unsignedLong	$[0, 18446744073709551615]$
short	$[-32768, 32767]$
unsignedShort	$[0, 65536]$
float, double	$[-\infty, \infty] \cup NaN$
boolean	$\{true, false\}$
string, normalizedString, token	$Z$
time	$\{(0-9), ':", 'Z', '-', '+\}$
date	$\{(0-9), '-', '+', 'Z\}$
dateTime	$\{(0-9), '-', ':', 'T', 'Z', '+\}$
duration	$\{'P', 'Y', 'M', 'W', 'D', 'T', 'H', 'M', 'S', (0-9), ':", ':', '-\}$
gDay, gMonth, gYear	$\{(0-9), '-', '+', 'Z\}$
gYearMonth, gMonthDay	$\{(0-9), '-', '+', 'Z\}$
Name, QName, anyURI, language	$Z$
NCName, ID, IDREF, IDREFS	$Z \setminus \{': '\}$
ENTITY, ENTITIES	$Z \setminus \{': '\}$
NOTATION	$Z$
NMTOKEN, NMTOKENS	$Z$

**Wohnadresse** von Seite 16 (Kapitel 2.2.2) das Intervall  $[0, 5]$  in der Relation *Card* annotiert. Für das Element **Zweitwohnsitz** von Seite 16 ist nur die linke Intervallgrenze im XML-Schema spezifiziert. Die rechte Intervallgrenze ergibt sich aus dem Vorgabewert, da der **maxOccurs**-Wert nicht im XML-Schema angegeben ist. Für das Element **Zweitwohnsitz** wird demnach das Intervall  $[0, 1]$  in der Relation *Card* annotiert.

Für jeden Knoten, der ein Attribut repräsentiert, wird überprüft, ob das XML-Schema eine Spezifikation des **use**-Attributs enthält. Falls das **use**-Attribut nicht im XML-Schema enthalten ist, wird das Intervall  $[0, 1]$  in der Relation *Card* annotiert. Wenn das **use**-Attribut spezifiziert ist, wird in der Relation *Card* das Intervall

$[0, 0]$  annotiert, wenn dem **use**-Attribut der Wert **prohibited** zugewiesen ist,

$[0, 1]$  annotiert, wenn dem **use**-Attribut der Wert **optional** zugewiesen ist und

$[1, 1]$  annotiert, wenn dem **use**-Attribut der Wert **required** zugewiesen ist.

Für jeden Knoten, der einen Namensraum repräsentiert, wird in der Relation *Card* das Intervall  $[1, 1]$  annotiert.

### Operationen auf Knotenmengen

Für die Vereinigung von Knotenmengen gibt es in XPath die Operation „|“. Dabei handelt es sich um eine einfache Mengenvereinigung. Somit können wir im formalen Modell den Operator „|“ durch die Mengenvereinigung „ $\cup$ “ beschreiben.

Die Vergleichsoperationen „ $<$ “ (Kleiner), „ $\leq$ “ (Kleiner oder Gleich), „ $>$ “ (Größer), „ $\geq$ “ (Größer oder Gleich), „ $=$ “ (Gleich) und „ $\neq$ “ (Ungleich) für Knotenmengen werden auf Vergleiche der Datentypen **boolean**, **numbers** und **string** zurückgeführt. Deshalb müssen wir die Vergleichsoperationen für Knotenmengen im formalen Modell nicht beschreiben.

Weitere Operationen auf Knotenmengen sind in XPath mit Hilfe von Funktionen realisiert. Diese Funktionen sind in der Grundbibliothek von XPath enthalten (siehe Kapitel 2.3.6). Im nächsten Kapitel erläutern wir die formale Beschreibung dieser Funktionen.

#### 3.1.5. Funktionen aus der Grundbibliothek von XPath

In diesem Abschnitt stellen wir die formale Beschreibung der Funktionen aus der Grundbibliothek von XPath vor. Die einzelnen Funktionen haben wir bereits in Kapitel 2.3.6 beschrieben. In diesem Abschnitt wiederholen wir für jede Funktion die Eigenschaften, die für die Berechnung wichtig sind. Für die formale Definition einer Funktion verwenden wir die Definitionen 1, 7 und 18 der XPath-Datentypen aus den vorhergehenden Abschnitten.

Im Datenmodell berücksichtigen wir keine Funktionen in XPath, die auf einen Kontextknoten zugreifen. Diese Funktionen können wir vernachlässigen, da es in BPEL-Prozessen

keinen voreingestellten Kontextknoten gibt (siehe [JE07], Anhang B, SA00027). Werden in BPEL-Prozessen XPath-Funktionen verwendet, die auf einen Kontextknoten zugreifen, muss sichergestellt sein, dass ein solcher Kontextknoten zur Verfügung steht. In BPEL-Prozessen steht ein Kontextknoten nur in Prädikaten innerhalb eines Lokalisierungspfads zur Verfügung. Da wir die Filterung mit Prädikaten in unserer Analyse nicht berücksichtigen, brauchen wir auch keine XPath-Funktionen berücksichtigen, die auf einen Kontextknoten zugreifen. Deshalb sind solche Funktionen nicht in unserem Datenmodell enthalten.

## Zahlenfunktionen

Im Folgenden sei  $X = (\emptyset \vee NaN)$ .

**numbers number(object?)** konvertiert das Argument in eine Zahl. Wie konvertiert wird, ist vom Typ des Arguments abhängig. Deshalb definieren wir vier Funktionen, die die vier Datentypen `boolean`, `numbers`, `string` und `node-set` auf Zahlen abbilden.

Eine Zahl wird nicht konvertiert. Deshalb repräsentieren wir diese Funktion als Identitätsfunktion.

$number : L_1 \longrightarrow L_1$

$$number(x) = x$$

Eine Zeichenkette kann nur in eine gültige Zahl überführt werden, wenn sie aus einem optionalen Minuszeichen gefolgt von einer Dezimalzahl besteht. Wenn der Wertebereich der Zeichenkette die erforderlichen Zeichen „0“ bis „9“ nicht enthält, so kann die Zeichenkette auf keinen Fall in eine gültige Zahl konvertiert werden. In diesem Fall können wir also den Wert *NaN* zurückgeben. Wie bereits in Kapitel 3.1.2 erwähnt, wird der Wert *NaN* auf das leere Intervall vereinigt mit *NaN* abgebildet. In jedem anderen Fall können wir das zu berechnende Intervall nicht sicher einschränken und geben das größte Element aus der Menge *Intervalle* zurück. Betrachten wir beispielsweise die abstrakte Zeichenkette {9}. Da diese abstrakte Zeichenkette alle möglichen Zeichenketten „9“, „99“, „999“, usw. repräsentiert, können wir das zu berechnende Intervall nicht einschränken. Deshalb liefert die Funktion in diesem Fall das größte Element aus der Menge *Intervalle* zurück.

$number : S_1 \longrightarrow L_1$

$$number(ZK) = \begin{cases} [] \cup NaN, & ZK \setminus \{0, \dots, 9\} = ZK; \\ [-\infty, \infty] \cup NaN, & \text{sonst.} \end{cases}$$

Der boolesche Wert *true* wird in die Zahl 1 konvertiert. Der Wert *false* wird in die Zahl 0 und den speziellen Wert konvertiert. Die Zahl die bei der Konvertierung eines booleschen Wertes in eine Zahl entstehen kann, liegt also im Intervall  $[0, 1]$ .

$number : B_1 \longrightarrow L_1$

$$number(Bool) = \begin{cases} [1, 1], & Bool = \{true\}; \\ [0, 0], & Bool = \{false\}; \\ [0, 1], & \text{sonst.} \end{cases}$$

Eine Knotenmenge  $K$  wird in eine Zahl konvertiert, indem der Zeichenkettenwert jedes Knotens in  $K$  ermittelt und danach in eine Zahl konvertiert wird. Schließlich wird die Summe aller so ermittelten Zahlen berechnet.

$number : K \longrightarrow L_1$

$$number(K) = \sum_{i=1}^{|K|} number(string(\{k_i\})), \text{ für } k_i \in K.$$

Wird beim Aufruf der Funktion  $number()$  kein Argument angegeben, ermittelt sich das Argument mit Hilfe des aktuellen Kontextknotens. Wie bereits erwähnt, berücksichtigen wir den Zugriff auf einen Kontextknoten nicht.

**numbers sum(node-set)** berechnet die Summe der Zahlen, die durch Konvertierung der Zeichenkettenwerte der einzelnen Knoten in der übergebenen Knotenmenge entstehen (siehe Kapitel 2.3.6).

$sum : K \longrightarrow L_1$

$$sum(K) = \sum_{i=1}^{|K|} number(string(k_i)), \text{ für } k_i \in K.$$

**numbers floor(numbers)** rundet die Zahl im Argument ab.<sup>8</sup>

$floor : L_1 \longrightarrow L_1$

$$floor(x) = \begin{cases} [ [x_1], [x_2] ] \cup X, & x = [x_1, x_2] \cup X \\ x, & x = [ ] \cup NaN. \end{cases}$$

**numbers ceiling(numbers)** rundet die Zahl im Argument auf.

$ceiling : L_1 \longrightarrow L_1$

$$ceiling(x) = \begin{cases} [ [x_1], [x_2] ] \cup X, & x = [x_1, x_2] \cup X \\ x, & x = [ ] \cup NaN. \end{cases}$$

---

<sup>8</sup>Die verwendete Symbolik für das Runden ist in Anhang A definiert und wird auch im weiteren Text verwendet.

**numbers round(numbers)** rundet auf oder ab, je nachdem welche ganze Zahl näher am Argument liegt. Gibt es zwei ganze Zahlen, die dem Argument am nächsten sind, wird aufgerundet. Da wir ein Intervall mit maximal zwei Zahlen betrachten, ergeben sich 5 Fälle. Vier Fälle behandeln die Zahlen im Intervall. Der letzte Fall betrachtet ein Argument, das den einzigen Zahlenwert *NaN* enthält.

$round : L_1 \longrightarrow L_1$

$$round(x) = \begin{cases} \lceil [x_1], [x_2] \rceil \cup X, & x = [x_1, x_2] \cup X \wedge \lceil x_1 \rceil - x_1 \leq 0.5 \wedge \lceil x_2 \rceil - x_2 \leq 0.5 \\ \lfloor [x_1], [x_2] \rfloor \cup X, & x = [x_1, x_2] \cup X \wedge \lfloor x_1 \rfloor - x_1 \leq 0.5 \wedge \lfloor x_2 \rfloor - x_2 > 0.5 \\ \lceil [x_1], [x_2] \rceil \cup X, & x = [x_1, x_2] \cup X \wedge \lceil x_1 \rceil - x_1 > 0.5 \wedge \lceil x_2 \rceil - x_2 \leq 0.5 \\ \lfloor [x_1], [x_2] \rfloor \cup X, & x = [x_1, x_2] \cup X \wedge \lfloor x_1 \rfloor - x_1 > 0.5 \wedge \lfloor x_2 \rfloor - x_2 > 0.5 \\ x, & x = [] \cup NaN. \end{cases}$$

### Zeichenkettenfunktionen

**string string(object?)** konvertiert das Argument in eine Zeichenkette. Wie konvertiert wird, ist vom Typ des Arguments abhängig. Deshalb definieren wir vier Funktionen, die die vier Datentypen **boolean**, **numbers**, **string** und **node-set** auf Zeichenketten abbilden.

Eine Zahl wird in eine Zeichenkette konvertiert, indem die einzelnen Ziffern einer Zahl als Zeichen interpretiert werden (siehe Kapitel 2.3.6). In der Funktion bilden wir ein Intervall auf eine Zeichenmenge ab. Das leere Intervall bilden wir auf die leere Menge ab. Das Intervall  $[] \cup NaN$  repräsentiert den speziellen Zahlenwert *NaN*. Deshalb bilden wir dieses Intervall auf die Menge  $\{N, a\}$  ab. Das Intervall  $[-\infty, \infty]$  kann alle darstellbaren Zahlen enthalten. Dementsprechend enthält die Menge, auf die das Intervall abgebildet wird, alle Ziffern von 0 bis 9 sowie das Zeichen „-“ für negative Zahlen und das Zeichen „.“ für Dezimalzahlen. Da die Werte  $\infty$  und  $-\infty$  auf die Zeichenketten *Infinity* und *-Infinity* abgebildet werden, müssen die Zeichen *I*, *n*, *f*, *i*, *t* und *y* ebenso in der Menge enthalten sein. Alle anderen Intervalle werden analog abgebildet.

$string : L_1 \longrightarrow S_1$

$$string(x) = \begin{cases} \emptyset, & x = []; \\ \{a, N\}, & x = [] \cup NaN; \\ \{-, ., 0, \dots, 9, f, I, i, n, t, y\}, & x = [-\infty, \infty]; \\ \{-, ., 0, \dots, 9, a, f, I, i, N, n, t, y\}, & x = [-\infty, \infty] \cup NaN; \\ \{-, ., 0, \dots, 9\}, & x = [a, b] \wedge a \neq -\infty \wedge b \neq \infty; \\ \{-, ., 0, \dots, 9, a, N\}, & x = [a, b] \cup NaN \wedge a \neq -\infty \wedge b \neq \infty; \end{cases}$$

Eine Zeichenkette wird nicht konvertiert. Deshalb repräsentieren wir diese Funktion als Identitätsfunktion.

$string : S_1 \longrightarrow S_1$

$$string(ZK) = ZK$$

Der boolesche Wert *true* wird in die Zeichenkette „true“ konvertiert. Der boolesche Wert *false* wird in die Zeichenkette „false“ konvertiert. Die Zeichenkette, die bei der Konvertierung eines booleschen Wertes in eine Zeichenkette entstehen kann, enthält also maximal die Zeichen *t, r, u, e, f, a, l, s*.

$string : B_1 \longrightarrow S_1$

$$string(Bool) = \begin{cases} \{e, r, t, u\}, & Bool = \{true\}; \\ \{a, e, f, l, s\}, & Bool = \{false\}; \\ \{a, e, f, l, r, s, t, u\}, & \text{sonst.} \end{cases}$$

Eine Knotenmenge wird in eine Zeichenkette konvertiert, indem der Zeichenkettenwert des ersten Knotens in Dokumentenordnung in der Menge zurück gegeben wird. Da wir die Dokumentenordnung nicht kennen, ermitteln wir den Zeichenkettenwert jedes Knotens mit Hilfe der Funktion *str()* und vereinigen die Ergebnisse.

$string : K \longrightarrow S_1$

$$string(K) = \bigcup_{i=1}^{|K|} str(k_i) \text{ mit } k_i \in K$$

Die Funktion *str()* ermittelt den Zeichenkettenwert eines Knotens. Repräsentiert dieser Knoten ein XML-Element bzw. Attribut mit einem einfachen Typ (mit Ausnahme der Typen `xsd:anyType` und `xsd:anySimpleType`), wird der Zeichenkettenwert durch den gespeicherten Datentyp repräsentiert. Repräsentiert der Knoten einen Namensraum, entspricht der Zeichenkettenwert dem Namensraum-URI. In diesen Fällen können wir also den gespeicherten Datentyp zurückgeben. Bei Vereinigungstypen beachten wir dabei, ob die Typen mit unterschiedlichen Repräsentationen im formalen Modell vereinigt werden. Ist dies der Fall, können wir das Ergebnis der Funktion nicht mehr einschränken und wir geben die Menge *Z* aller möglichen Zeichen zurück.

Repräsentiert der Knoten ein XML-Element mit einem komplexen Typ, für den das mixed-Attribut den Wert *false* enthält, entspricht der Zeichenkettenwert dieses Knotens der Verkettung der Zeichenkettenwerte aller Kindknoten. Sonst repräsentiert der Knoten ein XML-Element mit einem komplexen Typ mit dem mixed-Attribut *true*, oder ein XML-Element bzw. Attribut mit den Typen `xsd:anyType` oder `xsd:anySimpleType`. In diesen Fällen können wir keine Einschränkung an den erwarteten Zeichenkettenwert vornehmen und geben die Menge *Z* zurück.

$str((k, ((V, E), Name, Typ, Card))) =$

$$\begin{cases} y, & (k, y, typ) \in Typ \wedge (y \in B_1 \vee y \in L_1 \vee y \in S_1) \\ \bigcup_{\forall l} string(\{l\}), & (k, y, typ) \in Typ \wedge typ = \Lambda \wedge (k, l) \in E \\ & \wedge (l, 'E', b) \in Typ; \\ Z & \text{sonst.} \end{cases}$$

Wird beim Aufruf der Funktion `string()` kein Argument angegeben, ermittelt sich das Argument mit Hilfe des aktuellen Kontextknotens. Wie bereits erwähnt, berücksichtigen wir den Zugriff auf einen Kontextknoten nicht.

**string concat(string, string, string\*)** berechnet die Verkettung der Argumente. Deshalb vereinigen wir die Zeichenmengen der einzelnen Argumente. Wir definieren hier nur eine Funktion `concat()` mit zwei Argumenten. Die Verkettung von mehr als zwei Argumenten stellen wir durch die Verkettung der Funktion `concat()` dar, z.B.  $\text{concat}(ZK_1, ZK_2, ZK_3) = \text{concat}(\text{concat}(ZK_1, ZK_2), ZK_3)$ .

$\text{concat} : S_1 \times S_1 \longrightarrow S_1$

$$\text{concat}(ZK_1, ZK_2) = ZK_1 \cup ZK_2$$

**boolean starts-with(string, string)** liefert den booleschen Wert *true*, falls die Zeichenkette im ersten Argument mit der Zeichenkette im zweiten Argument beginnt. Sonst liefert die Funktion den Wert *false*. Sind im zweiten Argument keine Zeichen enthalten, wird nur die leere Zeichenkette repräsentiert. In diesem Fall liefert die Funktion den Wert *true*. Ist im zweiten Argument mindestens ein Zeichen enthalten, wird sowohl die leere Zeichenkette als auch eine nichtleere Zeichenkette repräsentiert. In diesem Fall können wir das Ergebnis der Funktion nur ohne Einschränkung voraussagen. Deshalb liefert die Funktion in diesem Fall sowohl den Wert *true* als auch den Wert *false*.

$\text{starts-with} : S_1 \times S_1 \longrightarrow B_1$

$$\text{starts-with}(ZK_1, ZK_2) = \begin{cases} \{true\}, & ZK_2 = \emptyset; \\ \{true, false\}, & \text{sonst.} \end{cases}$$

**boolean contains(string, string)** liefert den booleschen Wert *true*, falls die Zeichenkette im ersten Argument die Zeichenkette im zweiten Argument enthält. Sonst liefert die Funktion den Wert *false*. Wie bereits bei der Funktion `starts-with()` liefert die Funktion `contains()` den Wert *true*, falls das zweite Argument die leere Menge ist. In jedem anderen Fall können wir das Ergebnis der Funktion nur ohne Einschränkung voraussagen.

$\text{contains} : S_1 \times S_1 \longrightarrow B_1$

$$\text{contains}(ZK_1, ZK_2) = \begin{cases} \{true\}, & ZK_2 = \emptyset; \\ \{true, false\}, & \text{sonst.} \end{cases}$$

**string substring-before(string, string)** liefert den Teil der Zeichenkette im ersten Argument, der vor dem ersten Auftreten der Zeichenkette im zweiten Argument steht. Ist das zweite Argument gar nicht im ersten Argument enthalten, wird die leere Zeichenkette zurück gegeben. Da wir bei Zeichenketten von der Reihenfolge und Anzahl der Zeichen

abstrahieren, können wir das Ergebnis der Funktion nicht voraussagen. Demensprechend liefert die Funktion die Menge aller Zeichen, die auch im ersten Argument enthalten sind.

$substring-before : S_1 \times S_1 \longrightarrow S_1$

$$substring-before(ZK_1, ZK_2) = ZK_1$$

**string substring-after(string, string)** liefert den Teil der Zeichenkette im ersten Argument, der nach dem ersten Auftreten der Zeichenkette im zweiten Argument steht. Ist das zweite Argument gar nicht im ersten Argument enthalten, wird die leere Zeichenkette zurück gegeben. Wie bei der Funktion `substring-before()` können wir das Ergebnis der Funktion nicht voraussagen.

$substring-after : S_1 \times S_1 \longrightarrow S_1$

$$substring-after(ZK_1, ZK_2) = ZK_1$$

**string substring(string, numbers, numbers?)** liefert einen Teil der Zeichenkette im ersten Argument. Die Position, ab der dieser Teil beginnt, wird im zweiten Argument übergeben. Das dritte Argument ist optional und gibt die Länge der Teilzeichenkette an. Auch bei dieser Funktion können wir das Ergebnis der Funktion nicht voraussagen.

$substring : S_1 \times L_1 \times L_1 \longrightarrow S_1$

$$substring(ZK_1, x, y) = ZK_1$$

**numbers string-length(string?)** liefert die Anzahl der Zeichen in der Zeichenkette, die als Argument übergeben wird. Da wir von der Anzahl der Zeichen abstrahieren, können wir diese Information nicht mehr aus dem Argument ermitteln. Wir wissen aber, dass die Länge auf jeden Fall positiv ist. Deshalb liefern wir ein Intervall, dessen linke Grenze den Wert 0 besitzt.

$string-length : S_1 \longrightarrow L_1$

$$string-length(ZK) = [0, \infty]$$

Wird beim Aufruf der Funktion `string-length()` kein Argument angegeben, ermittelt sich das Argument mit Hilfe des aktuellen Kontextknotens. Wie bereits erwähnt, berücksichtigen wir den Zugriff auf einen Kontextknoten nicht.

**string normalize-space(string?)** liefert die Zeichenkette, die aus der übergebenen Zeichenkette nach Normalisierung des Leerraumes entsteht (siehe Kapitel 2.3.6). Da diese Funktion lediglich mehrfache Leerzeichen entfernt, ändert sich die übergebene Zeichenmenge für uns nicht.

*normalize-space* :  $S_1 \longrightarrow S_1$

$$\text{normalize-space}(ZK_1) = ZK_1$$

Wird beim Aufruf der Funktion `normalize-space()` kein Argument angegeben, ermittelt sich das Argument mit Hilfe des aktuellen Kontextknotens. Wie bereits erwähnt, berücksichtigen wir den Zugriff auf einen Kontextknoten nicht.

**string translate(string, string, string)** liefert die Zeichenkette, die der Zeichenkette im ersten Argument entspricht, wobei jedes Vorkommen eines Zeichens aus dem zweiten Argument durch das korrespondierende Zeichen aus dem dritten Argument ersetzt wird (siehe Kapitel 2.3.6). Da die Zeichen des zweiten Arguments aus der Zeichenkette entfernt werden, können wir diese Zeichen aus der Zeichenmenge im ersten Argument ausschließen. Die Zeichen des dritten Arguments erweitern ggf. die Zeichenmenge.

*translate* :  $S_1 \times S_1 \times S_1 \longrightarrow S_1$

$$\text{translate}(ZK_1, ZK_2, ZK_3) = (ZK_1 \setminus ZK_2) \cup ZK_3$$

## Boolsche Funktionen

**boolean boolean(object)** konvertiert das Argument in einen boolschen Wert. Wie konvertiert wird, ist vom Typ des Arguments abhängig. Deshalb definieren wir vier Funktionen, die die vier Datentypen `boolean`, `numbers`, `string` und `node-set` auf boolsche Werte abbilden.

Eine Zahl wird in den Wert *true* konvertiert, wenn sie weder 0,  $-0$  oder *NaN* ist. Ansonsten ergibt die Konvertierung den Wert *false*. Die Funktion liefert demnach die Menge mit dem einzigen Wert *true*, wenn das Intervall im Argument ausschließlich den Wert 1 repräsentiert. Die Funktion liefert die Menge mit dem einzigen Wert *false*, wenn das Intervall im Argument nur die Werte 0 oder *NaN* repräsentiert. In jedem anderen Fall liefert die Funktion eine Menge mit beiden boolschen Werten.

*boolean* :  $L_1 \longrightarrow B_1$

$$\text{boolean}(x) = \begin{cases} \{true\}, & x = [1, 1]; \\ \{false\}, & x = [0, 0] \vee x = [0, 0] \cup NaN \vee x = [ ] \cup NaN; \\ \{true, false\}, & \text{sonst.} \end{cases}$$

Eine Zeichenkette wird in den Wert *true* konvertiert, wenn die Zeichenkette nicht die leere Zeichenkette ist. Die leere Zeichenkette wird in den Wert *false* konvertiert. Repräsentiert das Argument ausschließlich die leere Zeichenkette, liefert die Funktion also eine Menge, die nur den Wert *false* enthält. In jedem anderen Fall repräsentiert das Argument sowohl die leere Zeichenkette als auch eine Zeichenkette mit mindestens einem Zeichen. Hier liefert die Funktion eine Menge mit beiden boolschen Werten.

$boolean : S_1 \longrightarrow B_1$

$$boolean(ZK) = \begin{cases} \{false\}, & ZK = \emptyset; \\ \{true, false\}, & \text{sonst.} \end{cases}$$

Ein boolscher Wert wird nicht konvertiert. Deshalb repräsentieren wir diese Funktion als Identitätsfunktion.

$boolean : B_1 \longrightarrow B_1$

$$boolean(Bool) = Bool$$

Eine Knotenmenge wird in den Wert *true* konvertiert, wenn die Knotenmenge nicht leer ist. Die leere Knotenmenge wird in den Wert *false* konvertiert. Ist die Menge *K* im Argument leer, ist auch die Knotenmenge leer. Eine leere Menge *K* wird demnach in eine Menge mit dem einzigen Wert *false* konvertiert. Ist die Menge *K* nicht leer, müssen wir noch die Kardinalitäten der Knoten in *K* überprüfen. Wenn jeder Knoten in *K* null mal vorkommen darf, kann die Knotenmenge ebenfalls leer sein. In diesem Fall konvertieren wir die Knotenmenge *K* in die Menge mit beiden boolschen Werten. Sonst liefert die Funktion eine Menge mit dem einzigen Wert *true*.

$boolean : K \longrightarrow B_1$

$$boolean(K) = \begin{cases} \{false\}, & K = \emptyset; \\ \{true, false\}, & \forall (k, ((V, E), Name, Typ, Card)) \in K : \\ & c = [0, n] \text{ mit } (k, c) \in Card \text{ und } n \in \mathbb{R}; \\ \{true\}, & \text{sonst.} \end{cases}$$

**not(boolean)** liefert den Wert *true*, wenn das Argument den Wert *false* enthält. Ansonsten liefert die Funktion den Wert *false*. Enthält die Menge im Argument beide boolschen Werte, liefert die Funktion auch die Menge mit beiden boolschen Werten.

$not : B_1 \longrightarrow B_1$

$$not(Bool) = \begin{cases} \{true\}, & Bool = \{false\}; \\ \{false\}, & Bool = \{true\}; \\ \{true, false\}, & \text{sonst.} \end{cases}$$

**boolean true()** liefert den Wert *true*. Da in XPath kein Literal für den Wert *true* definiert ist, wird diese Funktion verwendet, wenn der Wert *true* benötigt wird. Somit repräsentieren wir diese Funktion im formalen Modell nicht durch eine Funktion, sondern durch die Menge:

$$\{true\}.$$

**boolean false()** liefert den Wert *false*. Analog zur Funktion **true()** repräsentieren wir die Funktion **false()** im formalen Modell durch die Menge:

$$\{false\}.$$

## Funktionen auf Knotenmengen

**numbers count(node-set)** liefert die Anzahl der Knoten in der Knotenmenge, die im Argument übergeben wird. Die Anzahl der Knoten in einer Knotenmenge  $K$  berechnen wir mit Hilfe der Annotation *Card* für die Knoten. Ein Knoten in der Menge  $K$  ist ein Tupel  $(x, y)$ , wobei  $x$  einen Knoten im Baum  $y$  ist. Ein solcher Knoten  $x$  ist im Baum mit der Relation *Card* annotiert. Um die Anzahl der Knoten in der Menge  $K$  zu erhalten, summieren wir die Annotationen der Knoten  $x$  über alle Tupel  $(x, y)$  in der Menge  $K$ .

$count : K \longrightarrow L_1$

$$count(K) = \sum_{i=1}^{|K|} c_i \text{ mit}$$

$$(x_i, c_i) \in Card \wedge x_i \in V \wedge (x_i, ((V, E), Name, Typ, Card)) \in K$$

**string local-name(node-set?)** liefert den lokalen Namen ohne Präfix des ersten Knotens bzgl. der Dokumentenordnung in der übergebenen Knotenmenge. Die Funktion liefert die leere Zeichenkette, wenn die übergebene Knotenmenge leer ist oder der erste Knoten in der Knotenmenge keinen lokalen Namen besitzt. Da wir die Position eines Knotens bzgl. der Dokumentenordnung nicht entscheiden können, liefert die Funktion die Vereinigung über die lokalen Namen aller Knoten in der Knotenmenge.

$local-name : K \longrightarrow S_1$

$$local-name(K) = \bigcup_{i=1}^{|K|} \beta_Z(name_i) \text{ mit}$$

$$(x_i, (name_i, ns_i)) \in Name \wedge x_i \in V \wedge (x_i, ((V, E), Name, Typ, Card)) \in K$$

Wird beim Aufruf der Funktion `local-name()` kein Argument angegeben, ermittelt sich das Argument mit Hilfe des aktuellen Kontextknotens. Wie bereits erwähnt, berücksichtigen wir den Zugriff auf einen Kontextknoten nicht.

**string namespace-uri(node-set?)** liefert den Namensraum-URI des erweiterten Namens des ersten Knotens bzgl. der Dokumentenordnung in der übergebenen Knotenmenge. Die Funktion liefert die leere Zeichenkette, wenn die übergebene Knotenmenge leer ist, der erste Knoten in der Knotenmenge keinen erweiterten Namen besitzt oder der Namensraum-URI leer ist. Da wir die Position eines Knotens bzgl. der Dokumentenordnung nicht entscheiden können, liefert die Funktion die Vereinigung über die Namensraum-URIs aller Knoten in der Knotenmenge.

$namespace-uri : K \longrightarrow S_1$

$$\text{namespace-uri}(K) = \bigcup_{i=1}^{|K|} \beta_Z(ns_i) \text{ mit}$$

$$(x_i, (name_i, ns_i)) \in Name \wedge x_i \in V \wedge (x_i, ((V, E), Name, Typ, Card)) \in K$$

Wird beim Aufruf der Funktion `namespace-uri()` kein Argument angegeben, ermittelt sich das Argument mit Hilfe des aktuellen Kontextknotens. Wie bereits erwähnt, berücksichtigen wir den Zugriff auf einen Kontextknoten nicht.

**string name(node-set?)** liefert den erweiterten Namen des ersten Knotens bzgl. der Dokumentenordnung in der übergebenen Knotenmenge. Da wir die Position eines Knotens bzgl. der Dokumentenordnung nicht entscheiden können, liefert die Funktion die Vereinigung über die erweiterten Namen aller Knoten in der Knotenmenge.

$$name : K \longrightarrow S_1$$

$$name(K) = \bigcup_{i=1}^{|K|} \beta_Z(name_i) \cup \{\{, \}\} \cup \beta_Z(ns_i) \text{ mit}$$

$$(x_i, (name_i, ns_i)) \in Name \wedge x_i \in V \wedge (x_i, ((V, E), Name, Typ, Card)) \in K$$

Wird beim Aufruf der Funktion `name()` kein Argument angegeben, ermittelt sich das Argument mit Hilfe des aktuellen Kontextknotens. Wie bereits erwähnt, berücksichtigen wir den Zugriff auf einen Kontextknoten nicht.

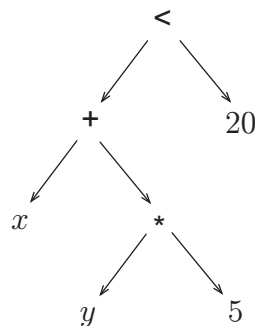
### 3.2. Drei-Adress-Code

In diesem Kapitel stellen wir den *Drei-Adress-Code* als Repräsentationsform für einen XPath-Ausdruck vor. Der Drei-Adress-Code ist eine gängige Repräsentationsform für Zwischencode im Compilerbau. Deshalb können wir hier die bewährten Techniken zur Erstellung des Drei-Adress-Codes anwenden. Eine ausführliche Beschreibung ist beispielsweise in [ASU99] enthalten. Hier stellen wir die Erstellung des Drei-Adress-Codes nur noch konzeptionell vor.

Der Drei-Adress-Code ist eine linearisierte Darstellung des *abstrakten Syntaxbaums*. Eine linearisierte Darstellung des abstrakten Syntaxbaums ermöglicht eine schnelle Traversierung. Deshalb haben wir den Drei-Adress-Code für die Repräsentation eines XPath-Ausdrucks gewählt. Der abstrakte Syntaxbaum eines XPath-Ausdrucks beschreibt die enthaltenen Operationen bzw. Funktionen sowie dessen Operanden bzw. Argumente. Dabei wird eine Operation bzw. Funktion als Knoten repräsentiert. Die Nachfolger dieses Knotens repräsentieren die Operanden bzw. Argumente. Durch die Baumrepräsentation werden auch die Prioritäten im XPath-Ausdruck repräsentiert. Syntaxbäume sind aus dem Compilerbau bekannt. Mehr Informationen zu Syntaxbäumen sind in [ASU99] enthalten.

Der Drei-Adress-Code besteht aus einer Folge von Anweisungen. Eine solche *Anweisung* enthält eine Operation bzw. Funktion und maximal zwei Bezeichner. Ein Bezeichner ist entweder eine Variable oder eine Konstante. Das Ergebnis einer Anweisung wird in einer temporären Variablen gespeichert. Zusammengesetzte Ausdrücke sind im Drei-Adress-Code nicht erlaubt. Deshalb wird ein zusammengesetzter Ausdruck durch eine Folge von Anweisungen repräsentiert. Im Folgenden beschreiben wir, wie ein zusammengesetzter Ausdruck in den Drei-Adress-Code überführt wird.

Um einen XPath-Ausdruck in Form des Drei-Adress-Codes zu repräsentieren, müssen wir zunächst den abstrakten Syntaxbaum für den XPath-Ausdruck aufstellen. Da die Prioritäten der Operationen im XPath-Ausdruck im abstrakten Syntaxbaum repräsentiert werden, enthält er alle Informationen, die bei der Ausführung des XPath-Ausdrucks beachtet werden müssen. Betrachten wir beispielsweise den XPath-Ausdruck  $x + y * 5 < 20$ , der durch den folgenden Syntaxbaum repräsentiert wird:



Der betrachtete Ausdruck beschreibt einen Vergleich zwischen zwei Zahlen. Da die Vergleichsoperation eine geringere Priorität als die Multiplikation oder die Addition hat, muss zunächst die linke Seite für den Vergleich ausgewertet werden. Hier muss beachtet werden, dass die Multiplikation eine höhere Priorität als die Addition hat. Deshalb wird zuerst die Multiplikation zwischen der Variablen  $y$  und dem Wert 5 ausgeführt. Danach wird die Variable  $x$  mit dem ermittelten Produkt addiert. Wie wir sehen können, werden die Prioritäten der verwendeten Operationen im Syntaxbaum repräsentiert.

Da wir nun den Syntaxbaum für den XPath-Ausdruck  $x + y * 5 < 20$  aufgestellt haben, können wir den Drei-Adress-Code für den Ausdruck ermitteln. Zunächst wird dabei jede Variable und jede Konstante im XPath-Ausdruck mit Hilfe der Operation *id* identifiziert. Wie bereits erwähnt wird das Ergebnis in einer temporären Variablen gespeichert. Für den betrachteten XPath-Ausdruck werden also zunächst die folgenden Anweisungen erzeugt:

$$\begin{aligned}
 tmp_1 &\leftarrow id\ x, \\
 tmp_2 &\leftarrow id\ y, \\
 tmp_3 &\leftarrow id\ 5, \\
 tmp_4 &\leftarrow id\ 20
 \end{aligned}$$

Nun traversieren wir über den Syntaxbaum und ermitteln die Operationen, die ausgeführt werden. Der Wurzelknoten repräsentiert die Vergleichsoperation. Da wir die beiden Operanden der Vergleichsoperation noch nicht ermittelt haben, können wir noch keine Anweisung für diese Operation bestimmen. Nun traversieren wir in einer Tiefensuche über den Syntaxbaum, um die Operanden zu ermitteln. Betrachten wir nun die Additionsoperation. Hier haben wir nur den ersten Operanden  $x$  ermittelt. Deshalb müssen wir weiter über den Syntaxbaum traversieren. So gelangen wir schließlich zur Multiplikationsoperation. Für diese Operation haben wir bereits beide Operanden ermittelt. Deshalb können wir die folgende Anweisung für die Repräsentation der Multiplikationsoperation erzeugen:

$$tmp_5 \leftarrow tmp_2 * tmp_3$$

Dabei werden die beiden Operanden als Verweise auf die zugehörigen temporären Variablen angegeben. Die Traversierung mit einer Tiefensuche gelangt nun zur Additionsoperation zurück. Da wir jetzt für das Ergebnis der Multiplikation eine temporäre Variable  $tmp_5$  erzeugt haben, sind die beiden Operanden für die Additionsoperation ermittelt. Somit können wir die folgende Anweisung für die Repräsentation der Additionsoperation erzeugen:

$$tmp_6 \leftarrow tmp_1 + tmp_5$$

Wiederum werden die beiden Operanden als Verweise auf die zugehörigen temporären Variablen angegeben. Nun gelangt die Traversierung mit einer Tiefensuche zur Vergleichsoperation zurück. Da wir jetzt beide Operanden für die Vergleichsoperation ermittelt haben, können wir die folgende Anweisung erzeugen:

$$tmp_7 \leftarrow tmp_6 < tmp_4$$

Damit haben wir den Syntaxbaum vollständig traversiert und der XPath-Ausdruck ist vollständig in Form des Drei-Adress-Codes repräsentiert.

Alle anderen numerischen Operationen aus XPath werden analog zur Addition und Multiplikation im Drei-Adress-Code repräsentiert. Auch die anderen Vergleichsoperationen aus XPath werden analog zur Operation „<“ im Drei-Adress-Code repräsentiert. Damit haben wir gesehen, wie numerische Operationen und Vergleichsoperationen aus XPath im Drei-Adress-Code repräsentiert werden. Die Vereinigungsoperation „|“ für Knotenmengen repräsentieren wir ebenfalls analog zur vorgestellten Repräsentation für numerische bzw. Vergleichsoperationen.

Wird eine Funktion in einem XPath-Ausdruck aufgerufen, repräsentieren wir das im Drei-Adress-Code durch mehrere Anweisungen. Zunächst werden die Parameter für den Funktionsaufruf durch eine Folge von Anweisungen repräsentiert. Danach können wir

den Funktionsaufruf durch eine Anweisung repräsentieren. So wird beispielsweise der Funktionsaufruf  $f(x_1, \dots, x_n)$  durch folgende Anweisungen repräsentiert:

$$\begin{aligned} tmp_1 &\leftarrow id\ x_1 \\ &\dots \\ tmp_n &\leftarrow id\ x_n \\ tmp_{n+1} &\leftarrow param\ tmp_1 \\ &\dots \\ tmp_{n+n} &\leftarrow param\ tmp_n \\ tmp_{2n+1} &\leftarrow call\ f\ n \end{aligned}$$

In den ersten  $n$  Zeilen werden wieder die Variablen  $x_1, \dots, x_n$  identifiziert. Die folgenden  $n$  Zeilen weisen diese Variablen als Parameter aus. Dazu wird die Operation *param* verwendet und der Parameter wird als Operand angegeben. Die Anweisung in der letzten Zeile repräsentiert nun den Funktionsaufruf. Hier verwenden wir die Operation *call* und geben die Funktion als ersten Operanden an. Der zweite Operand gibt die Anzahl der Parameter für den Funktionsaufruf an.

Damit wir die Parameter aus den Zeilen  $n + 1$  bis  $n + n$  dem Funktionsaufruf zuordnen können, erweitern wir diese Anweisungen. Da der zweite Operand für die Parameter nicht benötigt wird, geben wir hier die temporäre Variable an, in der das Ergebnis des Funktionsaufrufs gespeichert wird. Somit ersetzen wir die Zeilen  $n + 1$  bis  $n + n$  durch folgende Zeilen:

$$\begin{aligned} tmp_{n+1} &\leftarrow param\ tmp_1\ tmp_{2n+1} \\ &\dots \\ tmp_{n+n} &\leftarrow param\ tmp_n\ tmp_{2n+1} \end{aligned}$$

Die Funktionen *true()* und *false()* aus XPath repräsentieren die booleschen Werte *true* und *false*. Da diese Funktionen keine Parameter haben, werden sie im formalen Modell durch Mengen repräsentiert. Zur Vereinfachung unserer Analyse ersetzen wir deshalb einen Aufruf dieser Funktionen durch die entsprechende Menge aus dem formalen Modell. Beispielsweise wird der Funktionsaufruf *true()* im Drei-Adress-Code durch folgende Anweisung repräsentiert:

$$tmp_1 \leftarrow id\ \{true\}$$

Der Zugriff auf eine Variable wird in BPEL-Prozessen durch die Operation „\$“ beschrieben. Wir verwenden diese Operation bei der Repräsentation eines solchen Variablenzugriffs. Einen Variablenzugriff  $\$x$  in BPEL-Prozessen repräsentieren wir demnach durch folgende Anweisungen:

$$\begin{aligned} tmp_1 &\leftarrow id\ x \\ tmp_2 &\leftarrow \$\ tmp_1 \end{aligned}$$

Abschließend müssen wir noch betrachten, wie wir XPath-Lokalisierungspfade im Drei-Adress-Code repräsentieren. Wie bereits in Kapitel 2.3.5 erwähnt, besteht ein Lokalisierungspfad immer aus einem oder mehreren Lokalisierungsschritten. Außerdem wird ein Lokalisierungspfad immer bzgl. eines Kontextknotens ausgewertet. Der Kontextknoten wird im Lokalisierungspfad vor dem Zeichen „/“ angegeben. Da es in BPEL-Prozessen keinen voreingestellten Kontextknoten gibt, muss vor dem Zeichen „/“ immer ein Variablenzugriff oder ein Lokalisierungsschritt stehen. Betrachten wir beispielsweise den Lokalisierungspfad  $\$x/\text{child}::\text{Strasse}[\text{local-name()='An der Wiese'}]$ . Hier bestimmt die Variable  $x$  den Kontextknoten. Nach dem Zeichen „/“ beginnt ein Lokalisierungsschritt. Am Anfang des Lokalisierungsschrittes wird die Achse angegeben. Danach folgen zwei Doppelpunkte, die die Achse vom folgenden Knotentest trennen. Schließlich wird am Ende des Lokalisierungsschrittes das Prädikat in eckigen Klammern angegeben. Das Prädikat enthält wiederum einen XPath-Ausdruck. Den Lokalisierungspfad  $\$x/\text{child}::\text{Strasse}[\text{local-name()='An der Wiese'}]$  repräsentieren wir im Drei-Adress-Code durch folgende Anweisungen:

```

tmp1 ← id x
tmp2 ← id child
tmp3 ← id Strasse
tmp4 ← id "An der Wiese"
tmp5 ← $ tmp1
tmp6 ← / tmp5 tmp2
tmp7 ← :: tmp6 tmp3
tmp8 ← call local-name 0
tmp9 ← = tmp8 tmp4
tmp10 ← [ ] tmp7 tmp9

```

In den ersten vier Anweisungen werden die Variable und die Konstanten im Lokalisierungspfad identifiziert. Die fünfte Anweisung repräsentiert den Variablenzugriff im BPEL-Prozess. In der sechsten Anweisung wird nun der Kontextknoten für den Lokalisierungspfad bestimmt und die Achse ausgeführt. Als Operation verwenden wir dazu das Zeichen „/“ und geben als ersten Operanden den Kontextknoten an. Der zweite Operand enthält die Achse, die im Lokalisierungsschritt angegeben ist. Nach der Achse wird im Lokalisierungsschritt der Knotentest ausgeführt. Dafür wird in der siebten Anweisung die Operation „::“ angegeben und das Ergebnis der Achsenausführung als erster Operand angegeben. Der zweite Operand enthält den Knotentest. Im Lokalisierungsschritt wird nach dem Knotentest das Prädikat ausgewertet. Die achte und neunte Anweisung repräsentieren dafür den XPath-Ausdruck im Prädikat. Danach können wir in der letzten Anweisung das Prädikat repräsentieren, das bzgl. der Ergebnismenge des Knotentests ausgewertet wird. Als Operation wird dabei „[ ]“ angegeben und der erste Operand enthält das Ergebnis des Knotentests. Der zweite Operand enthält das Prädikat.

Wir haben nun gesehen, wie ein beliebiger XPath-Ausdruck in Form des Drei-Adress-Codes repräsentiert wird. Den ermittelten Drei-Adress-Code für einen XPath-Ausdruck speichern wir als eine Menge von Quadrupeln. Jedes Quadrupel repräsentiert dabei eine Anweisung im Drei-Adress-Code. Das erste Element des Quadrupels enthält die temporäre Variable, in der das Ergebnis der Anweisung gespeichert wird. Das zweite Element des Quadrupels enthält die Operation. Das dritte und vierte Element des Quadrupels enthält den ersten und zweiten Operanden. Bei einer unären Operation, wie z.B. *id* wird nur der erste Operand angegeben. Somit speichern wir den ermittelten Drei-Adress-Code für den XPath-Ausdruck  $x + y * 5 < 20$  in der folgenden Menge:

$$\{ (tmp_1, id, x, \emptyset), (tmp_2, id, y, \emptyset), (tmp_3, id, 5, \emptyset), (tmp_4, id, 20, \emptyset), (tmp_5, *, tmp_2, tmp_3), (tmp_6, +, tmp_1, tmp_5), (tmp_7, <, tmp_6, tmp_4) \}$$

### 3.3. Concurrent-Single-Static-Assignment-Form

In diesem Kapitel stellen wir die *Concurrent-Single-Static-Assignment-Form* (CSSA-Form) [LMP97] als Repräsentationsform für BPEL-Prozesse vor. Wir haben diese Repräsentation für BPEL-Prozesse gewählt, da eine CSSA-Form sowohl den Kontrollfluss als auch die Datenabhängigkeiten in einer kompakten Form repräsentiert. Weiterhin wurde in [God06] und [MMG<sup>+</sup>07] bereits eine CSSA-Form für BPEL-Prozesse vorgestellt, auf die wir in dieser Arbeit zurückgreifen.

Bei der Single-Static-Assignment-Form (SSA-Form) wird jeder Variablen genau ein Wert zugewiesen. Damit zeigt eine Variable in der SSA-Form das Verhalten einer Konstanten. Da einer Variablen  $x$  im BPEL-Prozess mehrere Werte zugewiesen werden können, erhält die Variable  $x$  in der SSA-Form einen Index. Sobald der Variablen  $x$  im BPEL-Prozess ein neuer Wert zugewiesen wird, erhöht sich der Index in der SSA-Form. Betrachten wir beispielsweise die Verzweigung in Abb. 3.2(a). Hier wird die Variable  $x$  im linken Zweig um 10 und im rechten Zweig um 20 erhöht. Die Abb. 3.2(b) zeigt die SSA-Form für die Verzweigung. Hier hat die Variable  $x$  nun einen Index, der bei jeder Wertzuweisung für die Variable  $x$  erhöht wird. Wie wir sehen können, wird jeder einzelnen Variablen  $x_1, x_2$  und  $x_3$  in der SSA-Form genau ein Wert zugewiesen.

Die SSA-Form beschreibt einen Graphen, in dem der Kontrollfluss sowie die Datenabhängigkeiten repräsentiert werden. In dieser Arbeit verwenden wir die um Parallelität erweiterte Single-Static-Assignment-Form, da BPEL die parallele Ausführung von Aktivitäten unterstützt. Die um Parallelität erweiterte Single-Static-Assignment-Form heißt Concurrent-Single-Static-Assignment-Form.



(a) Eine Verzweigung in einem BPEL-Prozess.

(b) Die Verzweigung in der SSA-Form.

**Abbildung 3.2.:** In einem BPEL-Prozess kann einer Variablen  $x$  mehrere Werte zugewiesen werden. Im SSA-Graphen wird jeder Variablen genau ein Wert zugewiesen. Deshalb erhält die Variable  $x$  einen Index, der die Unterscheidung zwischen den Variablen  $x_1, x_2, x_3, \dots$  ermöglicht.

### Spezielle Knoten in der CSSA-Form

In der CSSA-Form gibt es zwei spezielle Knoten:  $\phi$ - und  $\pi$ -Knoten. Ein  $\phi$ -Knoten repräsentiert das Zusammenführen von Kontrollflüssen. Beispielsweise muss der Kontrollfluss nach einer Verzweigung wieder zusammengeführt werden. Bei einer Verzweigung werden die Daten entweder in dem einen Zweig oder im anderen Zweig berechnet. Dies muss im CSSA-Graphen repräsentiert werden, da er sowohl Informationen über den Kontrollfluss als auch über die Datenabhängigkeiten enthält. Die Vereinigung von Kontrollflüssen wird in der CSSA-Form mit einem  $\phi$ -Knoten realisiert, der Informationen über die Daten enthält. Betrachten wir wieder die Verzweigung aus Abb. 3.2(b). Möchten wir den Kontrollfluss nach der Verzweigung zusammenführen, fügen wir am Ende der Verzweigung einen  $\phi$ -Knoten ein. Dieser  $\phi$ -Knoten definiert nun eine neue Variable  $x_4 = \phi(x_2, x_3)$ , die ihren Wert sowohl von  $x_2$  als auch von  $x_3$  erhalten kann.

Der  $\pi$ -Knoten repräsentiert in der CSSA-Form den parallelen Datenzugriff. Bei der parallelen Ausführung von Pfaden, muss beachtet werden, dass ein Datum evtl. bereits durch einen anderen parallelen Pfad verändert wurde. Deshalb wird für jedes Datum, das in einem parallelen Pfad verwendet wird, ein  $\pi$ -Knoten eingefügt, der das Verändern dieses Datums in den anderen parallelen Pfaden berücksichtigt. Betrachten wir beispielsweise zwei Pfade, die parallel ausgeführt werden. Im ersten Pfad wird die Variable  $x$  um den Wert 10 erhöht. Im zweiten Pfad wird die Variable  $x$  mit 20 multipliziert. Im CSSA-Graphen wird dann im ersten Pfad ein Knoten mit beispielsweise  $x_3 = x_0 + 10$  eingefügt. Im zweiten Pfad wird beispielsweise ein Knoten mit  $x_4 = x_0 * 20$  eingefügt. Die Variable  $x_0$  repräsentiert dabei den Wert der Variablen  $x$  vor der parallelen Ausführung der Pfade. Da die Pfade parallel ausgeführt werden, müssen wir aber auch berücksichtigen, dass die Zuweisung im ersten Pfad bereits ausgeführt sein kann, wenn die Zuweisung im zweiten Pfad ausgeführt wird. Weiterhin müssen wir berücksichtigen, dass die Zuweisung im zweiten Pfad bereits ausgeführt sein kann, wenn die Zuweisung im ersten Pfad ausgeführt wird. Das Datum in der Variablen  $x$ , das bei der Zuweisung im ersten Pfad verwendet wird, kann also vor der Ausführung der Pfade oder im zweiten Pfad berechnet worden sein. Deshalb fügen wir im ersten Pfad vor die Zuweisung einen  $\pi$ -

Knoten ein, der eine zusätzliche Variable  $x_2 = \pi(x_0, x_4)$  definiert. Dieser Knoten gibt an, dass die Variable  $x_2$  ihren Wert sowohl von  $x_0$  als auch von  $x_4$  erhalten kann. In der Zuweisung im ersten Pfad verwenden wir nun die Variable  $x_2$  statt  $x_0$ . Somit ist die neue Zuweisung im ersten Pfad  $x_3 = x_2 + 10$ . Im Gegensatz zur vorherigen Zuweisung wird berücksichtigt, dass das verwendete Datum auch vom zweiten Pfad berechnet sein kann. Analog fügen wir im zweiten Pfad vor die Zuweisung einen  $\pi$ -Knoten ein, der die zusätzliche Variable  $x_3 = \pi(x_0, x_2)$  definiert. Die Zuweisung im zweiten Pfad wird durch die Zuweisung  $x_4 = x_3 * 20$  ersetzt. Ein Beispiel für einen CSSA-Graphen mit  $\pi$ -Knoten ist in Anhang C enthalten.

### Der CSSA-Graph für WS-BPEL-Prozesse

Wie bereits erwähnt, wurde in [God06] eine CSSA-Form für BPEL-Prozesse vorgestellt, auf die wir in dieser Arbeit zurückgreifen. Dabei erweitern wir die bestehende CSSA-Form um benötigte Datenaspekte. Im Folgenden beschreiben wir die CSSA-Form, die wir in dieser Arbeit zur Repräsentation von BPEL-Prozessen verwenden. Dabei geben wir für jede Aktivität in BPEL einen CSSA-Teilgraphen an. Diese Teilgraphen werden entsprechend des Kontrollflusses im BPEL-Prozesses durch gerichtete Kanten miteinander verbunden. Abbildung 3.3 zeigt den CSSA-Graphen für einen BPEL-Prozess mit Namen `beispielProzess`. Im CSSA-Graphen wird der Anfang bzw. das Ende des Pro-

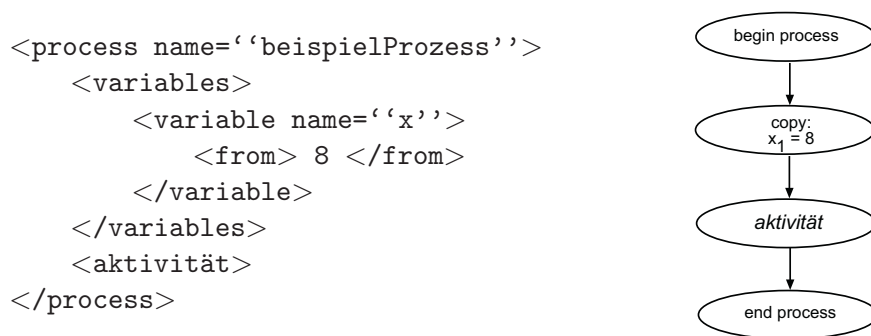


Abbildung 3.3.: Der CSSA-Graph für einen BPEL-Prozess.

zesses durch spezielle Knoten vom Typ *begin process* bzw. *end process* repräsentiert. Für jede Variableninitialisierung im BPEL-Prozess wird danach ein Knoten vom Typ *copy* eingefügt. In diesen Knoten ist die Information enthalten, welche Variable mit welchem Wert initialisiert wird. Die Knoten vom Typ *copy* werden als Sequenz nach dem Knoten vom Typ *begin process* eingefügt, da Variablen gleich zu Anfang der Ausführung eines BPEL-Prozesses initialisiert werden. Schließlich wird der CSSA-Teilgraph für die im BPEL-Prozess angegebene Aktivität ermittelt und im CSSA-Graphen zwischen der Variableninitialisierung und dem Ende des BPEL-Prozesses eingefügt.

Die CSSA-Teilgraphen für die einzelnen BPEL-Aktivitäten beschreiben wir im Folgenden. Wie in Abb. 3.3 geben wir für eine betrachtete BPEL-Aktivität eine beispielhafte

Beschreibung in XML-Notation an. Handler werden in unserem CSSA-Graphen nicht repräsentiert. Deshalb benötigen wir auch keine Repräsentation der Aktivitäten `throw`, `rethrow`, `compensate`, `compensateScope` und `exit`.

### Die Receive-Aktivität

Eine `receive`-Aktivität speichert eine Nachricht, die von einem Partnerservice<sup>9</sup> geliefert wird. Besteht die Nachricht aus einem Part, kann die gesamte Nachricht in einer Variablen gespeichert werden. Besteht die Nachricht aus mehreren Parts, kann auch nur ein Teil der Nachricht in einer Variablen gespeichert werden. Wir unterscheiden diese beiden Fälle im CSSA-Teilgraphen für eine `receive`-Aktivität. Wird die gesamte Nachricht gespeichert, verwenden wir den Teilgraphen in Abb. 3.4(a). Hier erzeugen wir einen Knoten vom Typ `receive`, der die Information enthält, woher die zu speichernde Nachricht stammt und wo sie gespeichert werden soll. Werden Teile einer Nachricht gespeichert, verwenden wir den Teilgraphen in Abb. 3.4(b). Hier erzeugen wir zunächst jeweils einen Knoten vom Typ `begin receive` und `end receive`. Zwischen diesen beiden Knoten erzeugen wir für jeden zu speichernden Teil einer Nachricht einen Knoten vom Typ `fromPart`. Ein solcher Knoten enthält dann die Information, woher der zu speichernde Teil der Nachricht stammt und wo er gespeichert werden soll.

Um eine Nachricht, die gespeichert werden soll, im CSSA-Graphen zu repräsentieren, verwenden wir eine Knotenmenge, wie in Kapitel 3.1.4 definiert. Die Knotenmenge für eine Nachricht ermitteln wir mit Hilfe des Datentyps der Nachricht und des Datentyps der Variablen, in der die Nachricht gespeichert werden soll. Da diese Variable einen generelleren Datentyp als den Datentyp der Nachricht haben kann, müssen wir beide Datentypen betrachten. Allerdings haben wir durch die Betrachtung beider Datentypen keinen Informationsverlust. Beide Datentypen sind durch ein XML-Schema gegeben.

Um den Datentyp einer Nachricht zu ermitteln, müssen wir die Operation betrachten, die von dem Partnerservice aufgerufen wird. Diese Operation ist in der Schnittstellenbeschreibung für den betrachteten BPEL-Prozess definiert. Wie bereits erwähnt wird in der Schnittstellenbeschreibung für jede Operation der Übergabeparameter angegeben. Der Übergabeparameter enthält einen oder mehrere Parts und repräsentiert eine Nachricht, die beim Aufruf der Operation empfangen wird. Somit können wir den angegebenen Datentyp eines Parts als Datentyp für die zu speichernde Nachricht verwenden.

In Kapitel 3.1.4 haben wir bereits angegeben, wie ein XML-Schema in einen Baum überführt wird. Dieses Verfahren wenden wir nun auf den Datentyp der Variablen an und ermitteln einen Baum  $baum_{var}$ . Außerdem wenden wir das Verfahren auf den Datentyp des zu speichernden Parts der Nachricht an und ermitteln den Baum  $baum_{msg}$ . Haben wir die beiden Bäume ermittelt, verändern wir den Baum  $baum_{var}$  so, dass er die speziellen Typinformationen aus dem Baum  $baum_{msg}$  enthält.

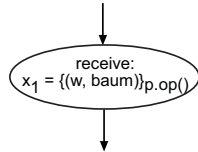
---

<sup>9</sup>Ein Partnerservice ist ein Service, mit dem der betrachtete BPEL-Prozess kommuniziert.

```

<receive
  partnerLink='p',
  operation='op',
  variable='x'>
</receive>

```

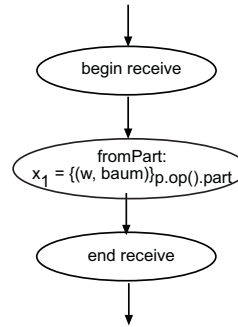


(a) Eine `receive`-Aktivität, die eine gesamte Nachricht speichert.

```

<receive
  partnerLink='p',
  operation='op'>
  <fromPart part='part',
    toVariable='x' />
</receive>

```



(b) Eine `receive`-Aktivität, die nur einen Teil einer Nachricht speichert.

**Abbildung 3.4.:** Eine `receive`-Aktivität wird in die angegebenen CSSA-Teilgraphen überführt. Wird die gesamte Nachricht gespeichert, wird der Teilgraph in 3.4(a) erzeugt. Dabei repräsentiert  $(w, \text{baum})_{p.op()}$  die Nachricht, die von Partner  $p$  bei Aufruf der Operation  $op$  empfangen wird. Wird nur ein Teil der Nachricht in einer Variablen  $x$  gespeichert, wird der Teilgraph in 3.4(b) erzeugt. Für jeden gespeicherten Teil wird ein Knoten vom Typ `fromPart` erzeugt. Dabei repräsentiert  $(w, \text{baum})_{p.op().part}$  den Teil  $part$  der Nachricht, die von Partner  $p$  bei Aufruf der Operation  $op$  empfangen wird.

Die Korrespondenz zwischen einem Knoten im Baum  $baum_{var}$  und  $baum_{msg}$  können wir mit Hilfe der Namen bilden. Die Namen für Knoten sind jeweils in den Baumannotationen  $Name_{var}$  bzw.  $Name_{msg}$  enthalten. Für die Korrespondenz verwenden wir aber nur den angegebenen Namen ohne den Namensraum. Um im Baum  $baum_{var}$  die speziellen Typinformationen aus dem Baum  $baum_{msg}$  zu berücksichtigen, betrachten wir nun für jeden Knoten  $x$  im Baum  $baum_{var}$  den korrespondierenden Knoten  $y$  im Baum  $baum_{msg}$ .

Zuerst speichern wir die spezielleren Namensrauminformationen: Besitzt der Knoten  $x$  Namensraumknoten als Kindknoten, löschen wir diese Namensraumknoten aus dem Baum  $baum_{var}$  und ersetzen sie durch die Namensraumknoten, die Kindknoten des Knotens  $y$  sind. Um den Namensraum auch für den betrachteten Knoten  $x$  spezifischer zu beschreiben, ersetzen wir das Element für  $x$  in der Annotation  $Name_{var}$  durch das Element für  $y$  in der Annotation  $Name_{msg}$ .

Nun speichern wir noch die spezielleren Datentypinformationen: Dabei betrachten wir die Annotation *Typ* für die Knoten *x* und *y*. Wenn die korrespondierenden Knoten *x* und *y* ein XML-Element bzw. Attribut repräsentieren, ersetzen wir das Element für *x* in der Annotation *Typ<sub>var</sub>* durch das Element für *y* in *Typ<sub>msg</sub>*.

Damit haben wir im Baum *baum<sub>var</sub>* die speziellen Typinformationen aus dem Baum *baum<sub>msg</sub>* gespeichert. Um diesen Baum im CSSA-Graphen als Knotenmenge verwenden zu können, müssen wir nur noch den Wurzelknoten *w* im Baum *baum<sub>var</sub>* ermitteln. Im CSSA-Graphen können wir nun die Nachricht, die in der **receive**-Aktivität gespeichert werden soll, durch die Knotenmenge  $\{(w, baum_{var})\}$  repräsentieren. Diese Knotenmenge ist in Abb. 3.4 durch  $\{(w, baum)\}_{p.op()}$  bzw.  $\{(w, baum)\}_{p.op().part}$  repräsentiert. Der Index an der Knotenmenge in Abb. 3.4 beschreibt den Partner, von dem die Nachricht empfangen wird, und die Operation, bei dessen Aufruf die Nachricht empfangen wird. Gegenenfalls beschreibt der Index auch den zu speichernden Part der empfangenen Nachricht.

### Die Reply-Aktivität

Eine **reply**-Aktivität sendet eine Nachricht an einen Partnerservice. Der CSSA-Teilgraph für eine **reply**-Aktivität ist in Abb. 3.5 dargestellt. Hier erzeugen wir einen Knoten vom Typ *reply*, der die Information enthält, welche Nachricht wohin gesendet wird.

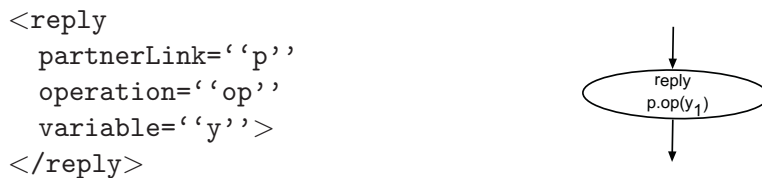


Abbildung 3.5.: Der CSSA-Teilgraph für eine **reply**-Aktivität.

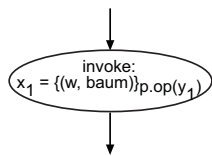
### Die Invoke-Aktivität

Eine **invoke**-Aktivität sendet eine Nachricht an einen Partnerservice und speichert ggf. die Antwort. Das Verhalten der **invoke**-Aktivität ist analog zum Verhalten der Aktivitäten **reply** und **receive**. Deshalb ist die Repräsentation der **invoke**-Aktivität als CSSA-Teilgraph analog zu diesen Aktivitäten.

Empfängt die **invoke**-Aktivität keine Antwort, ist der CSSA-Teilgraph der **invoke**-Aktivität der selbe wie der CSSA-Teilgraph der **reply**-Aktivität. Allerdings wird kein Knoten vom Typ *reply*, sondern ein Knoten vom Typ *invoke* erzeugt. Die Informationen dieses Knotens sind die selben wie beim Knoten vom Typ *reply*.

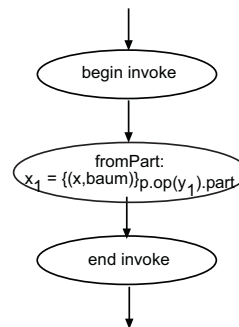
Empfängt die `invoke`-Aktivität eine Antwort, entspricht der CSSA-Teilgraph der `invoke`-Aktivität der Vereinigung der CSSA-Teilgraphen für die Aktivitäten `reply` und `receive`. Abbildung 3.6 beschreibt den CSSA-Teilgraphen für eine `invoke`-Aktivität, die eine Antwort empfängt. Analog zu den CSSA-Teilgraphen der Aktivitäten `reply` und `receive` enthalten die erzeugten Knoten die Informationen über die Nachrichten, die gesendet bzw. empfangen werden.

```
<invoke
  partnerLink='p',
  operation='op',
  inputVariable='x',
  outputVariable='y'>
</invoke>
```



(a) Eine `invoke`-Aktivität, die eine gesamte Antwort speichert.

```
<invoke
  partnerLink='p',
  operation='op',
  outputVariable='y'>
  <fromPart part='part',
    toVariable='x' />
</invoke>
```



(b) Eine `invoke`-Aktivität, die nur einen Teil einer Antwort speichert.

**Abbildung 3.6.:** Eine `invoke`-Aktivität wird in die angegebenen CSSA-Teilgraphen überführt. Wird die gesamte Nachricht gespeichert, wird der Teilgraph in 3.6(a) erzeugt. Dabei repräsentiert  $(w, \text{baum})_{p.op(y_1)}$  die Nachricht, die von Partner  $p$  bei Aufruf seiner Operation  $op$  mit dem Übergabeparameter  $y_1$  als Antwort gesendet wird. Wird nur ein Teil der Nachricht in einer Variablen  $x$  gespeichert, wird der Teilgraph in 3.6(b) erzeugt. Für jeden gespeicherten Teil wird ein Knoten vom Typ `fromPart` erzeugt. Dabei repräsentiert  $(w, \text{baum})_{p.op(y_1).part}$  den Teil  $part$  der Nachricht, die von Partner  $p$  bei Aufruf seiner Operation  $op$  mit dem Übergabeparameter  $y_1$  als Antwort gesendet wird.

### Die Assign-Aktivität

Eine `assign`-Aktivität repräsentiert eine Zuweisung in BPEL-Prozessen. Der CSSA-Teilgraph für eine `assign`-Aktivität ist in Abb. 3.7 dargestellt. Wir erzeugen zunächst jeweils einen Knoten vom Typ *begin assign* und *end assign*. Zwischen diesen beiden Knoten erzeugen wir für jedes `copy`-Element in der `assign`-Aktivität einen Knoten vom Typ *copy*. Dieser enthält dann die Information, welcher Variablen welcher Wert zugewiesen wird.

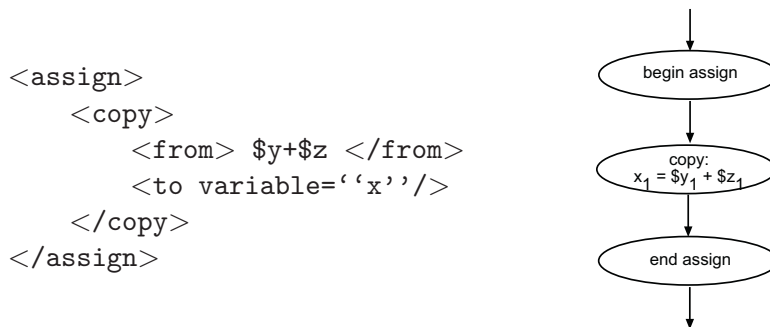


Abbildung 3.7.: Der CSSA-Teilgraph für eine `assign`-Aktivität.

### Die If-Aktivität

Eine `if`-Aktivität repräsentiert eine Verzweigung in BPEL-Prozessen. In jedem Zweig wird eine Aktivität angegeben, die ggf. ausgeführt wird. Bei einer `if`-Aktivität wird zunächst eine Bedingung für den `if`-Zweig angegeben. Ist diese Bedingung wahr, wird die Aktivität im `if`-Zweig ausgeführt. Zusätzlich können in einer `if`-Aktivität mehrere `elseif`-Zweige bzw. ein `else`-Zweig enthalten sein. Wenn die Bedingung für den `if`-Zweig nicht wahr ist, werden ggf. die Bedingungen für die `else-if`-Zweige ausgewertet. Ist keine solche Bedingung vorhanden, oder jede dieser Bedingungen ist nicht wahr, wird der `else`-Zweig ausgeführt, sofern dieser vorhanden ist. In Abb. 3.8 ist der CSSA-Teilgraph für eine `if`-Aktivität abgebildet.

Eine Bedingung repräsentieren wir im CSSA-Graphen durch einen Knoten vom Typ *condition*. Dieser Knoten enthält den Ausdruck, der die Bedingung angibt. Da die Bedingung wahr oder falsch sein kann, hat ein Knoten vom Typ *condition* immer zwei Nachfolger. Ein Nachfolger repräsentiert die Aktivität im betrachteten Zweig. Der andere Nachfolger repräsentiert einen weiteren Zweig oder das Ende der Verzweigung.

Da jede Bedingung in der Verzweigung zwei Pfade aufspaltet, fügen wir am Ende dieser beiden Pfade einen  $\phi$ -Knoten ein. Dieser  $\phi$ -Knoten repräsentiert das Zusammenführen des Kontrollfluss am Ende dieser beiden Pfade. Weiterhin enthält ein solcher  $\phi$ -Knoten die Dateninformationen der beiden Pfade. Am Anfang dieses Kapitels haben wir bereits beschrieben, wie die Dateninformationen zweier Pfade in einem  $\phi$ -Knoten repräsentiert werden.

```

<if>
  <condition> $x<$y </condition>
  <if aktivität>
  <elseif>
    <condition> $x=$y </condition>
    <else if aktivität>
  </elseif>
  <else>
    <else aktivität>
  </else>
</if>

```

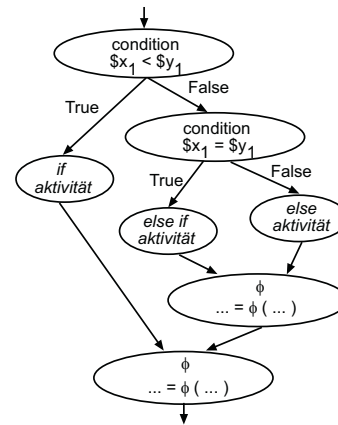


Abbildung 3.8.: Der CSSA-Teilgraph für eine if-Aktivität.

### Die While-Aktivität

Eine `while`-Aktivität repräsentiert eine Schleife, deren Bedingung für den Schleifeneintritt am Anfang der Schleife überprüft wird. Der CSSA-Teilgraph für eine `while`-Aktivität ist in Abb. 3.9 dargestellt.

```

<while>
  <condition> $x<$y </condition>
  <aktivität>
</while>

```

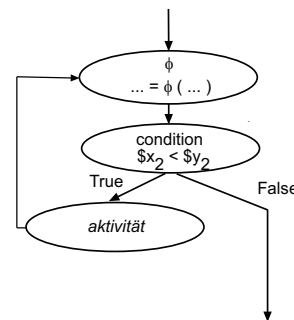


Abbildung 3.9.: Der CSSA-Teilgraph für eine while-Aktivität.

Beim Schleifeneintritt kommt der Kontrollfluss entweder von der Aktivität, die vor der Schleife ausgeführt wird, oder vom Schleifenkörper. Um die beiden Kontrollflüsse zu vereinigen, wird im CSSA-Teilgraphen zuerst ein  $\phi$ -Knoten eingefügt. Danach wird die Bedingung für den Schleifeneintritt durch einen Knoten vom Typ *condition* repräsentiert. Wie bereits erwähnt, hat ein Knoten vom Typ *condition* zwei Nachfolger. Ist die Bedingung wahr, wird die Schleife betreten und der Nachfolgeknoten repräsentiert den Schleifenkörper. Ist die Bedingung falsch, wird die erste Aktivität nach der Schleife ausgeführt.

### Die RepeatUntil-Aktivität

Eine `repeatUntil`-Aktivität repräsentiert eine Schleife, deren Bedingung für den Schleifeneintritt am Ende der Schleife überprüft wird. Der CSSA-Teilgraph für eine `repeatUntil`-Aktivität ist in Abb. 3.10 dargestellt. Da die beiden Aktivitäten `while` und `repeatUntil` ähnliche Schleifen repräsentieren, werden beide CSSA-Teilgraphen analog erstellt. Bei der `repeatUntil`-Aktivität befindet sich der Knoten vom Typ *condition* allerdings am Ende der Schleife, da die Bedingung für den Schleifeneintritt am Ende der Schleife überprüft wird.

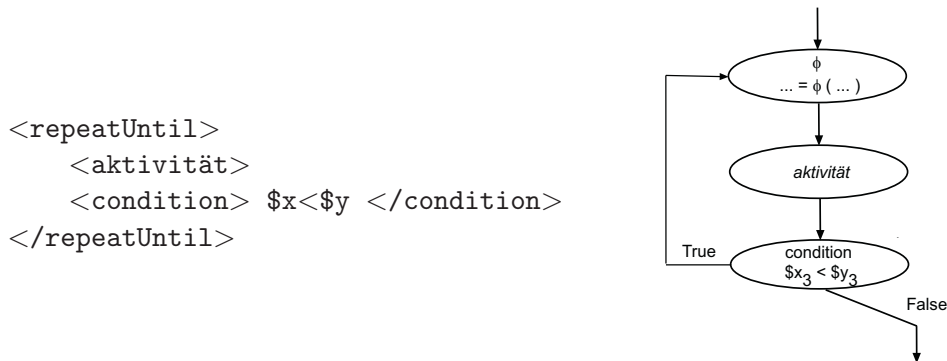


Abbildung 3.10.: Der CSSA-Teilgraph für eine `repeatUntil`-Aktivität.

### Die ForEach-Aktivität

Eine `forEach`-Aktivität repräsentiert eine `for`-Schleife, deren Ausführung unter einer bestimmten Bedingung (*completionCondition*) früher abgebrochen werden kann. Die Schleife, die durch eine `forEach`-Aktivität repräsentiert wird, kann sequentiell oder parallel ausgeführt werden. Im CSSA-Graphen unterscheiden wir diese beiden Fälle. Wird die Schleife sequentiell ausgeführt, wird der CSSA-Teilgraph in Abb. 3.11(a) erzeugt. Hier wird die *scope*-Aktivität in der Schleife in mehreren Iterationen ausgeführt. Wird die Schleife parallel ausgeführt, wird der CSSA-Teilgraph in Abb. 3.11(b) erzeugt. Hier wird die *scope*-Aktivität entsprechend der Differenz zwischen dem End- und dem Startwert der Laufvariablen kopiert.

Um die Laufvariable für die `for`-Schleife zu initialisieren, wird zu Beginn des CSSA-Teilgraphen ein Knoten vom Typ *initialize* eingefügt. Wird die Schleife sequentiell ausgeführt, wird der restliche CSSA-Teilgraph analog zur `while`-Aktivität erstellt. Der Knoten vom Typ *condition* repräsentiert hier ebenfalls die Abbruchbedingung für die Schleife. Solange die *completionCondition* nicht erfüllt ist und die Laufvariable im vorgegebenen Bereich ist, wird die Schleife ausgeführt. Deshalb beschreiben wir im Knoten vom Typ *condition* die Abbruchbedingung für die Schleife durch die Konjunktion der Verneinung der *completionCondition* und der Abbruchbedingung unter Betrachtung der Laufvariablen.

```

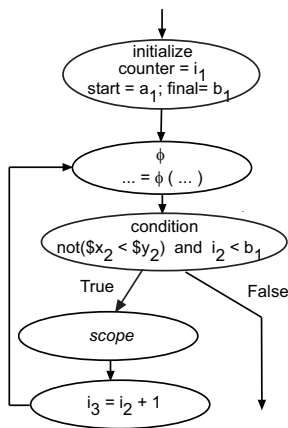
<forEach
  counterName='i'
  parallel='no'>
  <startCounterValue>
    $a
  </startCounterValue>
  <finalCounterValue>
    $b
  </finalCounterValue>
  <completionCondition>
    $x < $y
  </completionCondition>
  <scope> ...</scope>
</forEach>

```

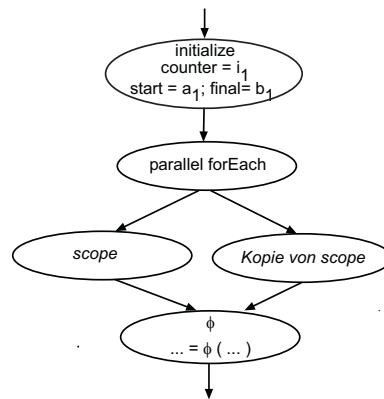
```

<forEach
  counterName='i'
  parallel='yes'>
  <startCounterValue>
    $a
  </startCounterValue>
  <finalCounterValue>
    $b
  </finalCounterValue>
  <completionCondition>
    $x < $y
  </completionCondition>
  <scope> ...</scope>
</forEach>

```



(a) Eine sequentielle `forEach`-Aktivität.



(b) Eine parallele `forEach`-Aktivität.

**Abbildung 3.11.:** Eine `forEach`-Aktivität wird in die angegebenen CSSA-Teilgraphen überführt. Wird die `forEach`-Aktivität sequentiell ausgeführt, wird der Teilgraph in 3.11(a) erzeugt. Wird die `forEach`-Aktivität parallel ausgeführt, wird der Teilgraph in 3.11(b) erzeugt.

Wird die Schleife parallel ausgeführt, enthält der CSSA-Teilgraph keinen Zyklus. Stattdessen wird die `scope`-Aktivität so oft kopiert, wie sie bei einer sequentiellen Ausführung der Schleife ausgeführt werden würde. Um den Kontrollfluss nach der parallelen Ausführung wieder zu vereinigen, wird am Ende ein  $\phi$ -Knoten eingefügt. Die `completionCondition` repräsentieren wir für eine parallel ausgeführte `forEach`-Aktivität nicht.

### Die Pick-Aktivität

Eine `pick`-Aktivität wartet auf genau ein Ereignis aus einer Menge von Ereignissen. Ein Ereignis kann dabei das Empfangen einer Nachricht sein, oder das Signal eines Zeitzählers sein. Tritt ein solches Ereignis auf, wird die mit dem Ereignis assoziierte Aktivität ausgeführt. Der Empfang einer Nachricht wird in der `pick`-Aktivität durch ein `onMessage`-Element repräsentiert. Beim Empfang einer Nachricht kann analog zur `receive`-Aktivität entweder die gesamte Nachricht oder nur ein Teil der Nachricht gespeichert werden. Dementsprechend ist der CSSA-Teilgraph für ein `onMessage`-Element analog zur `receive`-Aktivität. Wird in einem `onMessage`-Element die gesamte Nachricht gespeichert, erzeugen wir den CSSA-Teilgraphen in Abb. 3.12(a). Wird in einem `onMessage`-Element nur ein Teil einer Nachricht gespeichert, erzeugen wir den CSSA-Teilgraphen in Abb. 3.12(b).

In einer `pick`-Aktivität wird ein Zeitzähler in einem `onAlarm`-Element definiert. Ist die angegebene Zeitdauer überschritten, wird die assoziierte Aktivität ausgeführt. Für ein `onAlarm`-Element erzeugen wir im CSSA-Teilgraphen eine Knoten vom Typ `onAlarm`. Dieser Knoten enthält die Information über die Zeitdauer. Nach der Ausführung der mit einem Ereignis assoziierten Aktivität müssen wir den Kontrollfluss für alle Ereignisse wieder vereinigen. Deshalb wird am Ende des CSSA-Teilgraphen ein  $\phi$ -Knoten eingefügt.

### Die Flow-Aktivität

Eine `flow`-Aktivität repräsentiert die parallele Ausführung der enthaltenen Aktivitäten. Zur Synchronisierung können die enthaltenen Aktivitäten durch Links miteinander verbunden werden. Der CSSA-Teilgraph für eine `flow`-Aktivität ist in Abb. 3.13 dargestellt.

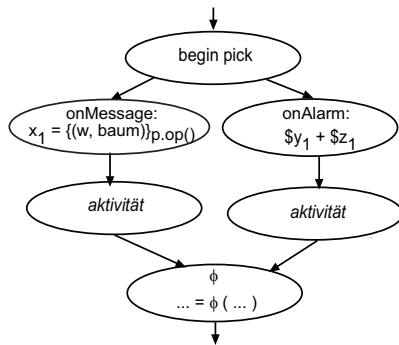
Zu Beginn der `flow`-Aktivität wird im CSSA-Teilgraphen ein Knoten vom Typ `begin flow` eingefügt. Jede Aktivität im Flow wird dann als Nachfolger dieses Knotens angegeben. Um den Kontrollfluss nach Ausführung aller Aktivitäten in der `flow`-Aktivität wieder zu vereinigen, fügen wir nach den Aktivitäten einen  $\phi$ -Knoten ein. Schließlich fügen wir am Ende einen Knoten vom Typ `end flow` ein.

Sind Links in der `flow`-Aktivität definiert, werden diese durch zusätzliche Kanten zwischen den entsprechenden Aktivitäten repräsentiert. Weiterhin wird eine `transitionCondition` durch einen Knoten vom Typ `tCondition` im CSSA-Teilgraphen repräsentiert. Der Vorgänger dieses Knotens ist die Aktivität, von der der betrachtete Link wegführt.

```

<pick>
  <onMessage
    partnerLink='p',
    operation='op',
    variable='x'>
    <aktivität>
  </onMessage>
  <onAlarm
    <for> $y + $z </for>
    <aktivität>
  </onAlarm>
</pick>

```

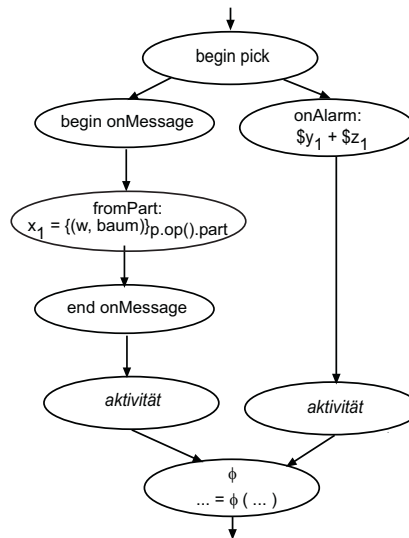


(a) Eine pick-Aktivität, die eine gesamte Nachricht speichert.

```

<pick>
  <onMessage
    partnerLink='p',
    operation='op'>
    <fromPart part='part',
    toVariable='x'>
    <aktivität>
  </onMessage>
  <onAlarm
    <for> $y + $z </for>
    <aktivität>
  </onAlarm>
</pick>

```



(b) Eine pick-Aktivität, die nur einen Teil einer Nachricht speichert.

**Abbildung 3.12.:** Eine pick-Aktivität wird in die angegebenen CSSA-Teilgraphen überführt. Wird in einem onMessage-Element die gesamte Nachricht gespeichert, wird der Teilgraph in 3.12(a) erzeugt. Wird in einem onMessage-Element nur ein Teil der Nachricht in einer Variablen x gespeichert, wird der Teilgraph in 3.12(b) erzeugt. Für jeden gespeicherten Teil wird ein Knoten vom Typ fromPart erzeugt.

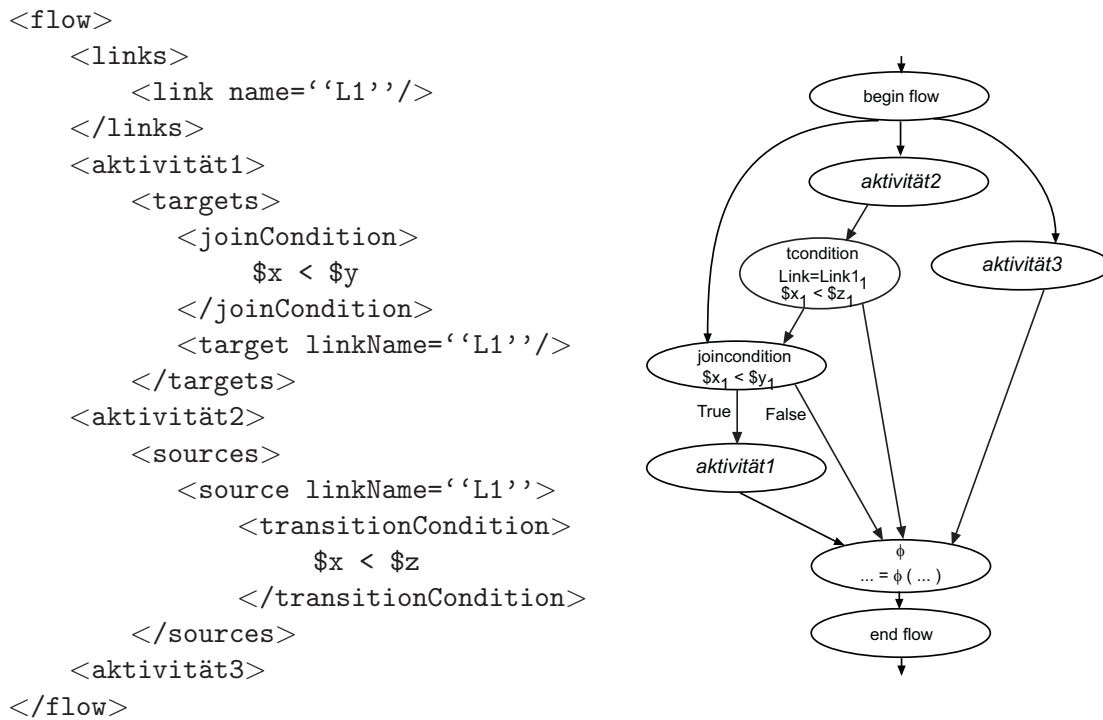


Abbildung 3.13.: Der CSSA-Teilgraph für eine flow-Aktivität.

Der Nachfolger dieses Knotens ist die Aktivität, in die der betrachtete Link führt. Eine *joinCondition* wird durch einen Knoten vom Typ *joinCondition* im CSSA-Teilgraphen repräsentiert. Die Vorgänger dieses Knotens sind alle Vorgänger der Aktivität, für die die *joinCondition* definiert ist. Diese Aktivität ist der Nachfolger dieses Knotens für den Fall, dass die *joinCondition* wahr ist. Für den Fall, dass die *joinCondition* falsch ist, ist der Nachfolger dieses Knotens der  $\phi$ -Knoten am Ende des Flows.

Wurde der CSSA-Teilgraph aus Abb. 3.13 für eine *flow*-Aktivität erzeugt, werden in diesen Teilgraphen noch  $\pi$ -Knoten eingefügt. Das Einfügen der  $\pi$ -Knoten haben wir bereits zu Anfang dieses Kapitels beispielhaft beschrieben. Dabei wird jeder Knoten im CSSA-Teilgraphen überprüft. Verwendet ein Knoten ein Datum  $x_i$ , wird ein  $\pi$ -Knoten erzeugt, der ein Datum  $x_{i-j}$  definiert. Bei der Definition wird jedes Datum  $x_k$  verwendet, das im CSSA-Teilgraphen definiert wird. Eine ausführliche Beschreibung für das Einfügen der  $\pi$ -Knoten im CSSA-Teilgraphen für eine *flow*-Aktivität ist in [God06] enthalten. Ein konkretes Beispiel für den CSSA-Teilgraphen einer *flow*-Aktivität mit  $\pi$ -Knoten ist im Anhang in Abb. C.1 dargestellt.

## Die Scope-Aktivität

Eine **scope**-Aktivität repräsentiert einen Gültigkeitsbereich im BPEL-Prozess. In diesem Gültigkeitsbereich können lokale Variablen definiert werden, die nur im Gültigkeitsbereich sichtbar sind. Der CSSA-Teilgraph für eine **scope**-Aktivität ist in Abb. 3.14 dargestellt.

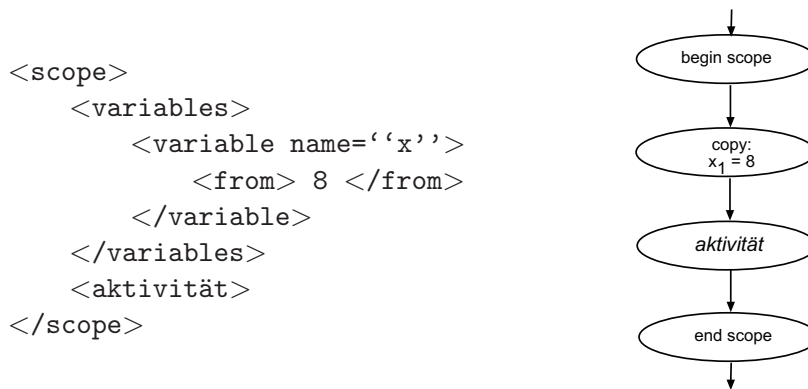


Abbildung 3.14.: Der CSSA-Teilgraph für eine **scope**-Aktivität.

Den Anfang bzw. das Ende einer **scope**-Aktivität repräsentieren wir durch einen Knoten vom Typ *begin scope* bzw. *end scope*. Für jede lokale Variable im Gültigkeitsbereich, die bei ihrer Deklaration auch initialisiert wird, fügen wir einen Knoten vom Typ *copy* ein. Dieser Knoten ist analog zu einem Knoten vom Typ *copy* in der **assign**-Aktivität. Durch die Indizes einer Variablen im CSSA-Graphen unterscheiden wir bereits zwischen lokalen Variablen, die in verschiedenen Namensräumen enthalten sind.

## Die Sequence-, Empty- und Wait-Aktivität

Die **sequence**-Aktivität repräsentiert eine sequentielle Ausführung der enthaltenen Aktivitäten. Die **empty**-Aktivität repräsentiert eine leere Anweisung. Die **wait**-Aktivität repräsentiert die Unterbrechung eines BPEL-Prozesses für eine bestimmte Zeitdauer. Die CSSA-Teilgraphen für diese Aktivitäten sind in Abb. 3.15 abgebildet.

Die **sequence**-Aktivität wird im CSSA-Teilgraphen durch zwei Knoten vom Typ *begin sequence* und *end sequence* repräsentiert. Diese beiden Knoten enthalten keine weiteren Dateninformationen. Die in der **sequence**-Aktivität enthaltenen Aktivitäten werden zwischen diesen beiden Knoten eingefügt. Die **empty**-Aktivität wird im CSSA-Teilgraphen durch einen Knoten vom Typ *empty* repräsentiert. Dieser Knoten enthält keine weiteren Informationen. Die **wait**-Aktivität wird im CSSA-Teilgraphen durch einen Knoten vom Typ *wait* repräsentiert. Dieser Knoten enthält die Zeitdauer, für die der BPEL-Prozess unterbrochen wird. Die Zeitdauer wird analog zur **pick**-Aktivität repräsentiert.

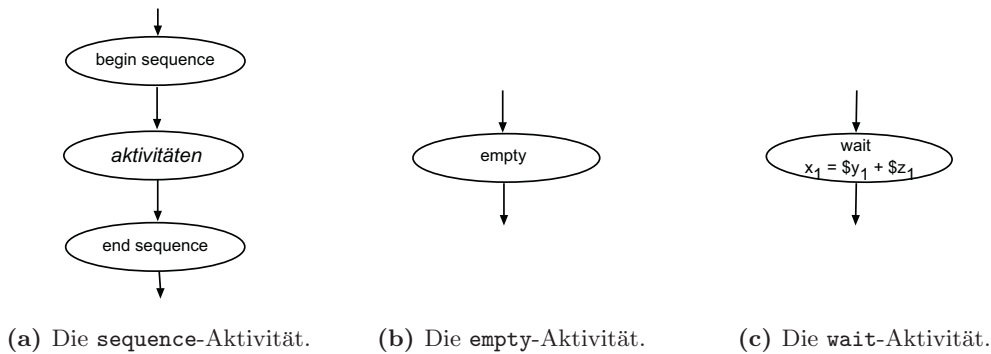


Abbildung 3.15.: Die CSSA-Teilgraphen für die sequence-, empty-, wait-Aktivitäten.

### 3.3.1. Erweiterungsfunktionen aus WS-BPEL

Zur Datenmanipulation stellt BPEL zusätzlich zu den Funktionen aus der Grundbibliothek von XPath zwei weitere Funktionen zur Verfügung. Diese Funktionen werden nicht in unserer Analyse ausgewertet. Stattdessen werden wir einen Aufruf dieser Funktionen im CSSA-Graphen geeignet repräsentieren. Im Folgenden beschreiben wir, wie ein Funktionsaufruf der Funktionen `getVariableProperty()` und `doXslTransform()` im CSSA-Graphen für einen BPEL-Prozess repräsentiert werden.

**getVariableProperty(string, string)** liefert das Datum, das in einer Variablen durch eine property benannt wird. Der erste Parameter benennt die Variable, in der das Datum enthalten ist. Der zweite Parameter benennt die property, durch die das Datum spezifiziert wird. Die Funktion ermittelt den `propertyAlias`, der für die betrachtete property definiert ist. Im `propertyAlias` ist nun ein XPath-Lokalisierungspfad angegeben, der das Datum in der Variablen adressiert. Im CSSA-Graphen ersetzen wir den Aufruf der Funktion `getVariableProperty()` durch den Zugriff auf die Variable und den im `propertyAlias` angegebenen XPath-Lokalisierungspfad. Beispielsweise wird Funktionsaufruf `getVariableProperty('var', 'prop')`, wobei für die property mit dem Namen 'prop' ein `propertyAlias` mit dem XPath-Lokalisierungspfad 'child::\*' definiert ist, durch `$var/child::*` ersetzt.

**doXslTransform(string, node-set, (string, object)\*)** führt eine XSLTransformation aus. Der erste Parameter enthält die Transformationsregeln. Der zweite Parameter enthält eine Knotenmenge, die genau einen Knoten enthalten darf. Die folgenden optionalen Parameter dienen als Parameter für die XSLTransformation. Die Funktion `doXslTransform()` wird in BPEL-Prozessen nur in Zuweisungen verwendet. Eine statische Analyse für eine XSLTransformation existiert noch nicht und übersteigt den Rahmen dieser Arbeit. Deshalb werden wir den Aufruf dieser Funktion nicht analysieren. Stattdessen ermitteln wir das Ergebnis der Funktion über den Datentyp der Variablen, der das Ergebnis der XSLTransformation zugewiesen wird. Da das Ergebnis der Funktion

einer Variablen zugewiesen werden muss, dessen Datentyp eine solche Zuweisung zulässt, ermitteln wir somit kein falsches Ergebnis. Allerdings haben wir unter Umständen einen größeren Informationsverlust, als bei der Analyse der XSLTransformation.

Wird in einem BPEL-Prozess die Funktion `doXslTransform()` in einer Zuweisung aufgerufen, ersetzen wir im CSSA-Graphen also den gesamten Ausdruck, der den Wert für die Zuweisung bestimmt. Dieser Ausdruck wird durch eine Knotenmenge ersetzt, die mit Hilfe des Datentyps der Variablen ermittelt wird, der ein Wert zugewiesen wird. Der Datentyp dieser Variablen ist immer durch ein XML-Schema definiert. In Kapitel 3.1.4 haben wir bereits angegeben, wie das XML-Schema der betrachteten Variable in einen Baum *baum* überführt werden kann. Danach ermitteln wir noch den Wurzelknoten *w* in diesem Baum. Damit können wir in der Zuweisung den Ausdruck für die Berechnung des Wertes durch die Knotenmenge  $\{(w, baum)\}$  ersetzen.

### 3.3.2. Einbettung des Drei-Adress-Codes in den CSSA-Graphen

Wir haben nun vorgestellt, wie ein BPEL-Prozess durch einen CSSA-Graphen repräsentiert wird. Dabei sind die XPath-Ausdrücke, die im BPEL-Prozess enthalten sind, in den Knoten enthalten. Um den Drei-Adress-Code für einen XPath-Ausdruck in den CSSA-Graphen einzubetten, ersetzen wir jeden XPath-Ausdruck im CSSA-Graphen durch den entsprechenden Drei-Adress-Code. Beispielsweise wird der CSSA-Teilgraph aus Abb. 3.7 in den folgenden CSSA-Teilgraphen geändert:

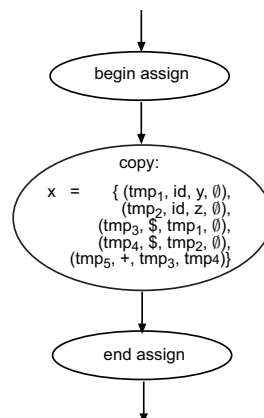


Abbildung 3.16.: Der CSSA-Teilgraph für eine assign-Aktivität mit der Repräsentation des enthaltenen XPath-Ausdruck im Drei-Adress-Code.

## Zusammenfassung

In diesem Kapitel haben wir ein Datenmodell für BPEL-Prozesse entwickelt. Dabei haben wir zunächst ein formales Datenmodell für XPath vorgestellt. Dieses formale Datenmodell ermöglicht eine Abstraktion von den konkreten Datenwerten in XPath-Ausdrücken. Zusätzlich haben wir die Operationen und Funktionen, die in XPath zur Verfügung stehen, für die Abstraktionen definiert.

Weiterhin haben wir in diesem Kapitel den Drei-Adress-Code zur Repräsentation von XPath-Ausdrücken in BPEL-Prozessen vorgestellt. Diese Repräsentationsform ermöglicht eine schnelle Auswertung eines XPath-Ausdrucks und ist deshalb in Hinblick auf unsere Analyse sehr geeignet. Danach haben wir die CSSA-Form zur Repräsentation eines BPEL-Prozesses vorgestellt. Diese Repräsentationsform bietet eine kompakte Darstellung des Kontrollflusses sowie der Datenabhängigkeiten eines BPEL-Prozesses. Den Drei-Adress-Code für XPath-Ausdrücke haben wir schließlich in die CSSA-Form eingebettet. Damit haben wir ein vollständiges Datenmodell für BPEL-Prozesse und die enthaltenen XPath-Ausdrücke vorgestellt. Auf der Grundlage dieses Datenmodells und dem formalen Datenmodell für XPath können wir nun eine abstrakte Interpretation von XPath-Ausdrücken in BPEL-Prozessen entwickeln.

## 4. Statische Analyse von XPath-Ausdrücken in WS-BPEL-Prozessen

Im vorhergehenden Kapitel haben wir ein Datenmodell für BPEL-Prozesse entwickelt. In diesem Datenmodell repräsentieren wir einen BPEL-Prozess als CSSA-Graphen. Die XPath-Ausdrücke im BPEL-Prozess repräsentieren wir in Form des Drei-Adress-Codes. In dem vorgestellten Datenmodell haben wir weiterhin ein formales Modell für XPath entwickelt, das eine Abstraktion von den konkreten Datenwerten in XPath ermöglicht. Auf Grundlage des vorgestellten Datenmodells beschreiben wir in diesem Kapitel die abstrakte Interpretation von XPath-Ausdrücken in BPEL-Prozessen. Mit Hilfe dieser abstrakten Interpretation berechnen wir die Wertebereiche von Variablen und Bedingungen in einem BPEL-Prozess. Da wir einen CSSA-Graphen für die Repräsentation eines BPEL-Prozesses verwenden, berechnen wir den Wertebereich einer Variablen zu einem bestimmten Punkt des Kontrollflusses im BPEL-Prozess.

Bei der statischen Datenanalyse für BPEL-Prozesse, die in diesem Kapitel vorgestellt wird, erläutern wir in Kapitel 4.1 zunächst einen Algorithmus, der über den CSSA-Graphen traversiert. Dieser Algorithmus überprüft für jeden Knoten im CSSA-Graphen, ob ein XPath-Ausdruck analysiert werden muss. Für die Analyse der XPath-Ausdrücke beschreiben wir dann in Kapitel 4.2 einen Algorithmus. Dieser Algorithmus berechnet für einen XPath-Ausdruck den Wertebereich, den der XPath-Ausdruck liefern kann.<sup>1</sup>

Die statische Analyse, die wir in diesem Kapitel vorstellen, ist eine sichere Analyse. Das bedeutet, dass die Analyse evtl. Ergebnisse liefert, die nicht zur Laufzeit entstehen. Allerdings ist jedes Ergebnis, das zur Laufzeit geliefert wird, in unserem Analyseergebnis enthalten. Somit bildet unser Analyseergebnis eine Obermenge für das Ergebnis zur Laufzeit. Die statische Analyse geht dabei von einem korrekten ausführbaren BPEL-Prozess aus.

Die Analyse in der vorliegenden Arbeit liefert eine Menge von Tupeln  $(x, W_x)$  als Ergebnis. Dabei repräsentiert  $x$  eine Variable bzw. Bedingung im BPEL-Prozess.  $W_x$  repräsentiert den berechneten Wertebereich für die Variable bzw. Bedingung. Der Wertebereich wird durch ein Element der vollständigen Verbände aus den Definitionen 1, 7 und 18 oder als Knotenmenge, wie in Definition 21 definiert, repräsentiert.

Um eine korrekte Berechnung des Wertebereichs zu garantieren, wird der Wertebereich  $W_x$  während der Analyse durch eine Menge von Tupeln repräsentiert. Das erste Element des Tupels repräsentiert den Wertebereich als Element der vollständigen Verbände

---

<sup>1</sup>Bei der Beschreibung der Algorithmen in Pseudo-Code bezeichnet ein Name, der mit einem Großbuchstaben beginnt, eine Menge. Ein Name, der mit einem Kleinbuchstaben beginnt, bezeichnet dagegen ein einzelnes Element.

aus den Definitionen 1, 7 und 18 oder als Knotenmenge, wie in Definition 21 definiert. Das zweite Element repräsentiert die Menge der Variablen, die für die Berechnung des Wertebereichs verwendet wurden. Am Ende der Analyse vereinigen wir die berechneten Wertebereiche in  $W_x$  und entfernen die Menge der verwendeten Variablen.

Im Folgenden stellen wir die Algorithmen vor, die die Datenanalyse für BPEL-Prozesse aus dieser Arbeit realisieren. Dabei geben wir für jeden Algorithmus eine ausführliche Beschreibung an. Weiterhin erläutern wir nach der ausführlichen Beschreibung die Korrektheit des jeweiligen Algorithmus.

#### 4.1. Analyse des CSSA-Graphen

In diesem Kapitel definieren wir einen Algorithmus `analyseCSSA()`, der über einen CSSA-Graphen traversiert und für die enthaltenen Variablen die Berechnung der Wertebereiche veranlasst. Der CSSA-Graph repräsentiert einen BPEL-Prozess, wie in Kapitel 3.3 vorgestellt. Ein Knoten im CSSA-Graph kann einen oder mehrere XPath-Ausdrücke enthalten, um die Werte von Variablen oder Bedingung zu beschreiben. Diese XPath-Ausdrücke sind in Form des Drei-Adress-Codes repräsentiert, wie wir in Kapitel 3.2 beschrieben haben.

##### 4.1.1. `analyseCSSA()`

Der Algorithmus `analyseCSSA()` ist in Algorithmus 1 definiert. Beim Aufruf werden dem Algorithmus fünf Parameter übergeben. Der erste Parameter enthält den CSSA-Graphen, über den traversiert werden soll. Der zweite Parameter enthält den Startknoten für die Analyse. Die letzten drei Parameter enthalten die benötigte Information, wenn der Algorithmus bei einer `flow`-Aktivität oder einer Schleife rekursiv aufgerufen wird. Der dritte Parameter enthält deshalb eine Liste mit bereits berechneten Wertebereichen. Der vierte Parameter enthält einen booleschen Wert, der angibt, ob wir zum aktuellen Zeitpunkt eine `flow`-Aktivität betrachten. Der fünfte Parameter enthält einen booleschen Wert, der angibt, ob wir zum aktuellen Zeitpunkt eine Schleife betrachten. Beim initialen Aufruf des Algorithmus `analyseCSSA()` wird der CSSA-Graph, der den vollständigen BPEL-Prozess repräsentiert, als erster Parameter übergeben. Der Startknoten des CSSA-Graphen wird als zweiter Parameter übergeben. Als dritter Parameter wird die leere Menge übergeben. Als vierter und fünfter Parameter wird der Wert `false` übergeben.

Nach dem Start des Algorithmus `analyseCSSA()` traversiert der Algorithmus in einer Variante der Breitensuche über den gegebenen CSSA-Graphen. Wir haben diese Form der Traversierung gewählt, um den tatsächlichen Kontrollfluss so gut wie möglich abzubilden. Außerdem können wir dadurch das mehrmalige Traversieren über den Graphen vermeiden. Allerdings müssen wir beachten, dass der CSSA-Graph kein Baum ist. Es gibt einige Knoten, die mehr als einen Vorgängerknoten haben. Beispielsweise wird der Kontrollfluss nach Verzweigungen wieder zusammengeführt. An diesen Stellen „wartet“ der Algorithmus ggf. solange, bis alle Vorgänger abgearbeitet wurden. Auch Schleifen müssen

gesondert betrachtet werden, da hier der Kontrollfluss vor der Schleife und der Kontrollfluss aus dem Schleifenkörper zusammengeführt wird. Bei Schleifen betrachten wir den Teilgraphen, der dem Schleifenkörper entspricht. Die mehrfache Iteration über den Schleifenkörper wird vom Algorithmus `analyseCSSA()` übernommen.

## Beschreibung

Im Algorithmus initialisieren wir die benötigten Variablen in den Zeilen 1 – 4. Die Mengen  $A$  und  $Kopie_A$  benötigen wir für die Realisierung der Breitensuche. Dabei enthält die Menge  $A$  alle Knoten, die in der aktuellen Hierarchieebene analysiert werden müssen. Die Menge  $Kopie_A$  wird verwendet, um nach der Analyse aller Knoten in der Menge  $A$ , die Knoten der nächsten Hierarchieebene zu ermitteln. Die Menge  $OUT$  repräsentiert die Ergebnismenge, in der die berechneten Wertebereiche gespeichert werden. Ein Element in der Menge  $OUT$  ist ein Tupel. Das erste Element im Tupel repräsentiert eine Variable. Das zweite Element im Tupel repräsentiert den Wertebereich der Variablen. Wie bereits erwähnt ist der Wertebereich während der Analyse erneut eine Menge von Tupeln.

Die Zeilen 5 – 9 realisieren die Breitensuche. Beim ersten Aufruf enthält die Menge  $Kopie_A$  den Startknoten. In den Zeilen 6 – 8 ermitteln wir dann die Nachfolger des Startknotens und speichern sie in der Menge  $A$ . Das bedeutet, dass der Startknoten nicht analysiert wird. Da er keine Dateninformation enthält, ist die Analyse dieses Knotens unnötig. In Zeile 9 speichern wir eine Kopie der Nachfolger in der Menge  $Kopie_A$ .

In den folgenden Zeilen des Algorithmus werden die Knoten in der Menge  $A$  nacheinander aus der Menge entfernt und analysiert. Ist die Menge  $A$  leer, werden die nächsten zu analysierenden Knoten mit Hilfe der Menge  $Kopie_A$  ermittelt. Für jeden Knoten in der Menge  $Kopie_A$  werden in den Zeilen 6 – 8 wie oben beschrieben die Nachfolger ermittelt und in der Menge  $A$  gespeichert. Die Bestimmung der Nachfolger wiederholt sich solange bis in der Menge  $Kopie_A$  nur noch der Endknoten des CSSA-Graphen enthalten ist. Ist nur noch der Endknoten enthalten, wurden alle Knoten im CSSA-Graphen besucht und der Algorithmus terminiert nach Abarbeitung von Zeile 84.

Wie bereits erwähnt, werden die Knoten aus der Menge  $A$  nacheinander entfernt und analysiert. Das Entfernen der Knoten wird in den Zeilen 10 – 12 realisiert. In Zeile 10 wird eine Schleife definiert, die eine Wiederholung für alle Knoten aus der Menge  $A$  ermöglicht. In den Zeilen 11 und 12 speichern wir einen Knoten aus der Menge  $A$  in der Variablen *aktuell* und entfernen ihn aus der Menge  $A$ .

Die Analyse des aktuellen Knotens beginnt in Zeile 13. Bei der Analyse unterscheiden wir zwischen den verschiedenen Knotentypen, da diese unterschiedliche Aktivitäten in BPEL repräsentieren. Die Analyse der verschiedenen Knotentypen beschreiben wir in den nächsten Abschnitten genauer. Die BPEL-Aktivitäten `reply`, `empty` und `wait` werden im Algorithmus vernachlässigt, da diese Aktivitäten keine Veränderung an den Daten vornehmen. Bei den Aktivitäten `sequence` und `scope` analysieren wir nur die Aktivitäten bzw. lokalen Variablen, die in der Sequenz bzw. dem Gültigkeitsbereich enthalten sind.

---

**Algorithmus 1:** analyseCSSA(CSSA-Graph, startknoten, Werteliste, flow, schleife)

```

1: KopieA ← {startknoten}
2: A ← ∅
3: OUT ← Werteliste, Besucht ← ∅, var ← ∅
4: inflow ← flow, Nochmal ← ∅
5: while not(KopieA = {x} ∧ nachfolger(x) = ∅) do
6:   for all x ∈ KopieA do
7:     A ← A ∪ nachfolger(x)
8:   end for
9:   KopieA ← A
10:  while A ≠ ∅ do
11:    aktuell ← hole einen Knoten aus A
12:    A ← A \ {aktuell}
13:    if aktuell.Typ = copy then
14:      W ← analyseXPath(aktuell.to, aktuell.from, OUT)
15:      var ← aktuell.to
16:    else if aktuell.Typ = receive ∨ aktuell.Typ = invoke ∨ aktuell.Typ = fromPart
17:      ∨ aktuell.Typ = onMessage then
18:        W ← {(aktuell.from, ∅)}
19:        var ← aktuell.to
20:    else if (aktuell.Typ = condition) then
21:      W ← analyseXPath(„condition“, aktuell.condition, OUT)
22:      var ← aktuell
23:    else if (aktuell.Typ = initialize) then
24:      W ← analyseXPath(„initialize“, aktuell.start, OUT)
25:      W ← W ∪ analyseXPath(„initialize“, aktuell.final, OUT)
26:      var ← aktuell.counter
27:    else if aktuell.Typ = begin flow then
28:      ((V,E), ende) ← teilgraphFlow(CSSA-Graph, aktuell)
29:      OUT ← analyseCSSA((V,E), aktuell, OUT, true, schleife)
30:      KopieA ← (KopieA \ {aktuell}) ∪ {ende}
31:      Besucht ← Besucht ∪ V
32:    else if aktuell.Typ = joinCondition ∨ aktuell.Typ = transitionCondition then
33:      if vorgänger(aktuell) ∈ Besucht then
34:        W ← analyseXPath(„condition“, aktuell.condition, OUT)
35:        if aktuell.Typ = joinCondition then
36:          var ← aktuell
37:        else
38:          var ← aktuell.Link
39:        end if
40:      else
41:        aktuell ← null

```

---

---

```

41:     KopieA ← KopieA \ {aktuell}
42:   end if
43: else if aktuell.Typ = π then
44:   OUT ← OUT ∪ analysePi(aktuell, OUT)
45: else if aktuell.Typ = end flow then
46:   if Nochmal ≠ ∅ then
47:     A ← {startknoten}
48:     KopieA ← {startknoten}
49:     Besucht ← ∅
50:     Nochmal ← ∅
51:   end if
52: else if aktuell.Typ = φ then
53:   (startSchleife, Schleife) ← startZyklus(CSSA-Graph, aktuell, aktuell, Besucht)
54:   Schleife ← Schleife \ Besucht
55:   if startSchleife then
56:     ((V,E), ende) ← teilgraphSchleife(CSSA-Graph, aktuell, Schleife)
57:     W1 ← analysePhi(aktuell, OUT)
58:     Tmp1 ← analyseCSSA((V,E), aktuell, W1, inflow, true)
59:     W2 ← analysePhi(aktuell, Tmp1)
60:     Tmp2 ← widening(W1, W2)
61:     while Tmp2 ≠ Tmp1 do
62:       Tmp1 ← Tmp2
63:       Tmp2 ← analyseCSSA((V,E), aktuell, Tmp1, inflow, true)
64:       Tmp2 ← analysePhi(aktuell, Tmp2)
65:     end while
66:     OUT ← Tmp2
67:     Besucht ← Besucht ∪ V
68:     KopieA ← (KopieA \ {aktuell}) ∪ {ende}
69:     A ← A ∪ {ende}
70:   else
71:     if vorgänger(aktuell) ∈ Besucht then
72:       OUT ← analysePhi(aktuell, OUT)
73:     else
74:       aktuell ← null
75:       KopieA ← KopieA \ {aktuell}
76:     end if
77:   end if
78: end if
79: (OUT, Nochmalaktuell) ← speichern(var, W, OUT, aktuell, inflow, schleife)
80: Nochmal ← Nochmal ∪ Nochmalaktuell
81: var ← ∅
82: Besucht ← Besucht ∪ {aktuell}
83: end while
84: end while

```

```
85: if not(flow)  $\wedge$  not(schleife) then  
86:   OUT  $\leftarrow$  zusammenfassen(OUT)  
87: end if  
88: return OUT
```

---

Eine Zuweisung wird in Zeile 13 erkannt. Ein copy-Knoten repräsentiert eine Zuweisung in der `assign`-Aktivität oder die Initialisierung einer Variablen. Der Wert, der zugewiesen wird, wird durch einen XPath-Ausdruck angegeben. Dieser XPath-Ausdruck ist im copy-Knoten unter „from“ gespeichert. Die Variable, der dieser Wert zugewiesen wird, ist im copy-Knoten unter „to“ gespeichert. Um den Wertebereich dieser Variablen zu berechnen, veranlassen wir die Analyse des XPath-Ausdrucks, der in „from“ gespeichert ist. Dazu rufen wir in Zeile 14 den Algorithmus `analyseXPath()` auf. Den ermittelten Wertebereich speichern wir in der Variablen  $W$ . In Zeile 15 speichern wir die Variable, der der Wert zugewiesen wird, in der Variablen  $var$ .

Die kommunikationsrelevanten Aktivitäten `receive`, `invoke` und `pick` werden in Zeile 16 erkannt. Wird in diesen Aktivitäten nur ein Teil einer Nachricht gespeichert, treten die `fromPart`-Knoten auf. Die zu speichernde Nachricht haben wir bei diesen Aktivitäten bereits als Knotenmenge, wie in Kapitel 3.1.4 definiert, repräsentiert. Deshalb können wir in Zeile 17 diese Knotenmenge in der Variablen  $W$  unverändert speichern. Die Variable, in der die Nachricht gespeichert wird, weisen wir in Zeile 18 der Variablen  $var$  zu.

Bedingungen werden in Zeile 19 erkannt. Eine Bedingung wird im BPEL-Prozess als XPath-Ausdruck angegeben. Um den Wertebereich der Bedingung zu ermitteln, rufen wir in Zeile 20 den Algorithmus `analyseXPath()` auf. Da das Resultat einer Bedingung keiner Variablen zugewiesen wird, speichern wir in Zeile 21 den aktuellen Knoten in der Variablen  $var$ .

Ein `initialize`-Knoten wird in Zeile 22 erkannt. Dieser Knoten enthält den Start- und Endwert der Laufvariablen einer `forEach`-Schleife. Sowohl der Startwert als auch der Endwert ist als XPath-Ausdruck angegeben. Um den Wertebereich des Startwerts zu ermitteln, rufen wir in Zeile 23 den Algorithmus `analyseXPath()` auf. Danach ermitteln wir in Zeile 24 den Wertebereich des Endwertes und vereinigen diesen mit dem Wertebereich des Startwertes. Wir vereinigen die beiden Wertebereiche, da die Laufvariable die Werte zwischen dem Startwert und dem Endwert annehmen kann. In Zeile 25 speichern wir den Namen der Laufvariable in der Variablen  $var$ .

Der Anfang einer `flow`-Aktivität wird in Zeile 26 erkannt. Eine Flow-Aktivität wird wegen seiner parallelen Ausführung speziell behandelt. In Zeile 27 ermitteln wir mit Hilfe des Algorithmus `teilgraphFlow()`<sup>2</sup> den Teilgraphen im CSSA-Graphen, der gerade die `flow`-Aktivität repräsentiert. Mit diesem Teilgraphen rufen wir in Zeile 28 den Algorithmus `analyseCSSA()` rekursiv auf. In einer `flow`-Aktivität verlaufen Datenzugriffe parallel. Deshalb müssen wir die Ergebnismenge sukzessiv aufbauen und am Ende einer `flow`-Aktivität ggf. erneut über den CSSA-Graphen traversieren. Da Flows geschach-

---

<sup>2</sup>Der Algorithmus `teilgraphFlow()` ist in Anhang B definiert.

telt werden können, muss am Ende einer `flow`-Aktivität der zugehörige Anfangsknoten ermittelt werden. Dies wird durch die Rekursion sicher gestellt.

Bei der Analyse des Flow-Teilgraphen werden vorher berechnete Daten benötigt. Deshalb übergeben wir beim rekursiven Aufruf von `analyseCSSA()` die Ergebnismenge `OUT`. Der rekursiv aufgerufene Algorithmus liefert eine Vereinigung der übergebenen Menge mit den im Flow berechneten Wertebereichen. Deshalb weisen wir in Zeile 28 das Ergebnis des rekursiven Aufrufs der Menge `OUT` zu. Schließlich ersetzen wir in Zeile 29 den Anfangsknoten des Flows mit dem Endknoten des Flows in der Menge `KopieA`. Somit werden die Knoten im Flow nicht mehr analysiert. In Zeile 30 markieren wir die Knoten im Flow als besucht, indem wir sie in die Menge `Besucht` aufnehmen.

In Zeile 31 werden die Knoten erkannt, die eine `joinCondition` oder `transitionCondition` innerhalb eines Flows repräsentieren. Eine `joinCondition` beschreibt die Bedingung, unter der die nachfolgende Aktivität ausgeführt wird. Eine `transitionCondition` ermittelt den Linkstatus des Links, der aus der vorhergehenden Aktivität herausführt. Da das Ergebnis dieser Bedingungen erst ermittelt werden kann, wenn die benötigten Daten berechnet sind, warten wir mit der Auswertung bis alle Vorgängerknoten analysiert wurden. Dies entspricht dem tatsächlichen Kontrollfluss.

In Zeile 32 überprüfen wir, ob alle Vorgängerknoten bereits besucht wurden. Ist dies der Fall, analysieren wir die Bedingung des Knotens in Zeile 33. Die Zeilen 34 – 38 bestimmen den Inhalt der Variablen `var`. Gibt es mindestens einen Vorgängerknoten, der noch nicht besucht wurde, werden die Zeilen 40 und 41 ausgeführt. Hier wird der Verweis auf den aktuellen Knoten gelöscht, damit er später nicht als besucht markiert wird. Außerdem löschen wir den aktuellen Knoten aus der Menge `KopieA`, damit seine Nachfolger noch nicht analysiert werden. Der aktuelle Knoten wird erneut besucht, wenn einer der nicht analysierten Vorgängerknoten analysiert wurde.

$\pi$ -Knoten im CSSA-Graphen werden in Zeile 43 erkannt.  $\pi$ -Knoten beschreiben parallele Datenzugriffe. Für die Analyse eines solchen Knotens haben wir den Algorithmus `analysePi()` definiert, der in Zeile 44 aufgerufen wird. Dieser Algorithmus betrachtet alle Variablen, die im  $\pi$ -Knoten definiert sind und ermittelt deren Wertebereich. Der Algorithmus liefert eine Vereinigung der übergebenen Wertebereichsmenge mit der Menge der Wertebereiche aus dem  $\pi$ -Knoten. Das Ergebnis des Algorithmus speichern wir als neue Menge `OUT`.

Das Ende eines Flows wird in Zeile 45 erkannt. Wie bereits erwähnt müssen wir die Ergebnismenge eines Flows sukzessiv aufbauen. Deshalb überprüfen wir am Ende eines Flows, ob der Flow erneut analysiert werden muss. Ein Flow wird solange analysiert, bis sich die Wertebereiche der Variablen innerhalb des Flows nicht mehr ändern. Ändert sich der Wertebereich einer Variablen innerhalb eines Knotens, wird dieser Knoten in der Menge `Nochmal` gespeichert. Deshalb wissen wir, dass der Flow erneut analysiert werden muss, wenn die Menge `Nochmal` nicht leer ist. Die Bedingung in Zeile 46 überprüft die Menge `Nochmal` auf enthaltene Elemente. Ist mindestens ein Element in der Menge `Nochmal` enthalten, stellen wir in den Zeilen 47 – 50 die Anfangsbedingungen für die

Analyse wieder her. Sobald der Endknoten des Flows besucht wird, ist die Menge  $A$  leer, da der Algorithmus eine Breitensuche realisiert und der Enknoten eines Flows genau einen Vorgängerknoten besitzt. Deshalb können wir die Anfangsbedingungen in den Zeilen 47 – 50 wieder herstellen, ohne dass wir weitere Knoten in der Menge  $A$  analysieren müssen. Ist die Menge *Nochmal* leer, muss der Flow nicht erneut analysiert werden. In diesem Fall terminiert der Algorithmus, da die Menge  $A$  leer ist und der Endknoten des Flows keinen Nachfolgeknoten besitzt.

Zeile 52 erkennt einen  $\phi$ -Knoten im CSSA-Graph. Ein  $\phi$ -Knoten repräsentiert im CSSA-Graphen das Zusammenführen des Kontrollflusses. Der Kontrollfluss wird entweder zu Beginn einer Schleife oder am Ende einer Verzweigung zusammengeführt. Um entscheiden zu können, ob der aktuelle  $\phi$ -Knoten den Beginn einer Schleife oder das Ende einer Verzweigung repräsentiert, rufen wir in Zeile 53 den Algorithmus `startZyklus()`<sup>3</sup> auf. Repräsentiert der  $\phi$ -Knoten den Beginn einer Schleife, liefert der Algorithmus `analyseZyklus()` den Wert *true* und eine Knotenmenge. Diese Knotenmenge enthält die bereits analysierten Knoten sowie die Knoten eines Pfades im Schleifenkörper. Repräsentiert der  $\phi$ -Knoten nicht den Beginn einer Schleife, liefert der Algorithmus den Wert *false* und die leere Menge.

In Zeile 55 überprüfen wir nun, ob der aktuelle  $\phi$ -Knoten den Beginn einer Schleife repräsentiert. Ist dies der Fall, ermitteln wir in Zeile 56 mit Hilfe des Algorithmus `teilgraphSchleife()`<sup>4</sup> den Teilgraphen, der den Schleifenkörper repräsentiert. Die Zeilen 57 und 58 analysieren nun die Schleife für eine Iteration. In Zeile 57 wird der aktuelle  $\phi$ -Knoten mit Hilfe des Algorithmus `analysePhi()` analysiert. Dieser Algorithmus betrachtet alle Variablen, die im  $\phi$ -Knoten definiert sind und ermittelt deren Wertebereich. Als Ergebnis liefert der Algorithmus `analysePhi()` die Vereinigung der übergebenen Wertebereichsmenge mit der Menge der Wertebereiche aus dem  $\phi$ -Knoten. Danach analysieren wir in Zeile 58 den Schleifenkörper, indem wir den Algorithmus `analyseCSSA()` rekursiv aufrufen. Da wir nur den Schleifenkörper analysieren, gerät die Analyse nicht in eine Endlosschleife. Schließlich analysieren wir in Zeile 59 den aktuellen  $\phi$ -Knoten erneut, um die Resultate aus der Analyse des Schleifenkörpers zu berücksichtigen.

Da wir nun eine Iteration über die Schleife analysiert haben, können wir entscheiden, ob ein *widening* vorgenommen werden muss. Beim *widening* abstrahieren wir vom konkreten Datenwert und verwenden stattdessen eine untere bzw. obere Grenze für den Datenwert. In Zeile 60 rufen wir dafür den Algorithmus `widening()` auf.<sup>5</sup> Dieser Algorithmus nimmt ggf. das *widening* vor, und liefert die resultierende Wertebereichsmenge zurück. Um die Auswirkungen des *widenings* im Schleifenkörper zu berücksichtigen, müssen wir die Schleife nach dem *widening* nochmals analysieren. In den Zeilen 61 – 65 realisieren wir die erneute Analyse der Schleife. In Zeile 61 wird sichergestellt, dass wir die Schleife solange analysieren, bis sich die Wertebereiche im Schleifenkörper nicht mehr ändern. Da wir das *widening* bereits vorgenommen haben, ist dieser Zeitpunkt irgendwann erreicht.

---

<sup>3</sup>Der Algorithmus `startZyklus()` ist in Anhang B definiert.

<sup>4</sup>Der Algorithmus `teilgraphSchleife()` ist in Anhang B definiert.

<sup>5</sup>Bei der Beschreibung des Algorithmus `widening()` erläutern wir die Technik des *widenings* genauer.

Nachdem sich die Wertebereiche im Schleifenkörper nicht mehr ändern, speichern wir in Zeile 66 die ermittelte Ergebnismenge in der Menge *OUT*. Da wir die Schleife vollständig analysiert haben, markieren wir in Zeile 67 die Knoten der Schleife als besucht. Außerdem ersetzen wir in Zeile 68 den aktuellen  $\phi$ -Knoten in der Menge *Kopie<sub>A</sub>* durch den Knoten, der nach der Schleife besucht wird. Da wir diesen Knoten noch analysieren müssen, fügen wir ihn in Zeile 69 in die Menge *A* ein.

Repräsentiert der aktuelle  $\phi$ -Knoten das Zusammenführen des Kontrollflusses am Ende einer Verzweigung, werden die Zeilen 71 – 76 ausgeführt. Beim Zusammenführen des Kontrollflusses am Ende einer Verzweigung warten wir solange, bis alle Daten zu Verfügung stehen, die in der Verzweigung berechnet werden. Dies entspricht dem tatsächlichen Kontrollfluss. Wurden alle Vorgänger des  $\phi$ -Knotens besucht, analysieren wir den  $\phi$ -Knoten in Zeile 72 mit Hilfe des Algorithmus *analysePhi()* und speichern die Ergebnismenge in Menge *OUT*. Gibt es mindestens einen Vorgängerknoten, der noch nicht analysiert wurde, entfernen wir in Zeile 74 den Verweis auf den  $\phi$ -Knoten, damit er später nicht als besucht markiert wird. Außerdem entfernen wir in Zeile 75 den  $\phi$ -Knoten aus der Menge *Kopie<sub>A</sub>*, damit seine Nachfolger noch nicht analysiert werden. Der Knoten wird erneut besucht, wenn einer der nicht analysierten Vorgängerknoten besucht wurde.

Nun haben wir jeden Knotentyp betrachtet, der für die Datenmanipulation interessant ist. Jeder andere Knotentyp wird ignoriert. Nach der Analyse eines Knotens, müssen wir das Ergebnis ggf. in der Ergebnismenge speichern. In diesem Fall ist die betrachtete Variable in *var* gespeichert. Der ermittelte Wertebereich ist in *W* gespeichert. Um das Analyseergebnis in der Ergebnismenge zu speichern, rufen wir in Zeile 79 den Algorithmus *speichern()* auf. Befinden wir uns in einem Flow oder einer Schleife, überprüft der Algorithmus zusätzlich, ob sich der Wertebereich der betrachteten Variable geändert hat. Deshalb liefert der Algorithmus ein Tupel. Das erste Element des Tupels enthält die neue Ergebnismenge. Das zweite Element enthält den aktuellen Knoten, falls wir uns in einem Flow befinden und der Wertebereich der betrachteten Variablen verändert wurde. Nachdem wir das Analyseergebnisse in der Ergebnismenge gespeichert haben, aktualisieren wir in Zeile 80 die Menge *Nochmal*. Die Menge *Nochmal* wird um den aktuellen Knoten erweitert, wenn wir uns in einem Flow befinden und der Wertebereich der aktuellen Variable verändert wurde.

Da wir die Analyse des aktuellen Knotens abgeschlossen haben, wird in Zeile 81 der Inhalt der Variablen *var* gelöscht. Dies ist nötig, damit das Ergebnis des nächsten Knotens korrekt gespeichert wird. Schließlich markieren wir in Zeile 82 den aktuellen Knoten als besucht, indem wir die Menge *Besucht* um den aktuellen Knoten erweitern. Damit ist die Analyse des aktuellen Knotens abgeschlossen. In Zeile 10 wird nun überprüft, ob weitere Knoten für die Analyse in der Menge *A* enthalten sind. Falls alle Knoten in der Menge *A* analysiert wurden, ermitteln wir in den Zeilen 6 – 9 die Nachfolger der analysierten Knoten und starten die Analyse der Nachfolger.

Ist die Analyse aller Knoten im CSSA-Graphen abgeschlossen, befindet sich in der Menge *Kopie<sub>A</sub>* nur noch der Endknoten des CSSA-Graphen. Da dieser Endknoten keinen Nach-

folger besitzt, ist die Bedingung in Zeile 5 nicht erfüllt. Die Schleife wird nicht mehr betreten und der Algorithmus terminiert nach Abarbeitung der Zeilen 85 – 88. In Zeile 85 überprüfen wir, ob der Algorithmus rekursiv aufgerufen wurde. Wurde der Algorithmus nicht rekursiv aufgerufen, rufen wir in Zeile 86 den Algorithmus `zusammenfassen()` mit dem Parameter *OUT* auf. Die Menge *OUT* enthält in diesem Fall die Ergebnismenge für den CSSA-Graphen, der einen vollständigen BPEL-Prozess repräsentiert. Da ein berechneter Wertebereich durch eine Menge gegeben ist, müssen wir die Elemente dieser Menge zusammenfassen. Der Algorithmus `zusammenfassen()` fasst die Wertebereichsmenge für jede Variablen zusammen und entfernt die Menge der verwendeten Variablen. Für jede Variable ermittelt der Algorithmus `zusammenfassen()` also ein Element der vollständigen Verbände aus den Definitionen 1, 7 und 18 oder eine Knotenmenge *K*, wie in Definition 21 definiert. Wurde der Algorithmus rekursiv aufgerufen, analysieren wir einen Flow oder eine Schleife. In diesem Fall verändern wir die Ergebnismenge *OUT* nicht. In Zeile 88 geben wir schließlich die ermittelte Ergebnismenge zurück und der Algorithmus terminiert.

### Terminierung und Korrektheit

Der Algorithmus 1 unterscheidet zwischen den verschiedenen Aktivitäten im BPEL-Prozess. Für jede Aktivität, die Daten im BPEL-Prozess manipuliert, analysiert der Algorithmus die entsprechenden Knoten im CSSA-Graphen. Das Ergebnis einer Knotenanalyse wird in der Ergebnismenge gespeichert, damit es für die Analyse der nächsten Knoten zur Verfügung steht.

Der Algorithmus terminiert, da wir eine Breitensuche realisieren. Bei der Analyse der speziellen Kontrollflussstrukturen Schleifen und Flows traversiert der Algorithmus mehrmals über den CSSA-Graphen. Hier terminiert der Algorithmus ebenfalls, da wir bei Schleifen eine widening-Operation anwenden und die Flowanalyse nur die tatsächlich möglichen parallelen Datenzugriffe berücksichtigt. Im BPEL-Prozess beschränkt der Kontrollfluss die möglichen Kombinationen des parallelen Datenzugriffs. Da wir diese Beschränkung in unserer Analyse beachten, terminiert der Algorithmus auch bei der Flowanalyse.

Der Algorithmus ist korrekt, da wir eine Breitensuche realisieren und damit garantiert ist, dass jeder Knoten im CSSA-Graphen besucht wird und die Knoten in der richtigen Reihenfolge analysiert werden. Das Analyseergebnis eines Flows ist ebenfalls korrekt, da alle tatsächlich möglichen parallelen Datenzugriffe berücksichtigt werden. Sobald alle möglichen Kombinationen des parallelen Datenzugriffs berücksichtigt wurden, ist die Analyse beendet. Ein Beispiel für die Analyse eines Flows ist in Anhang C enthalten.

Unsere Flowanalyse ist ein bekanntes Datenfluss-Problem, für dessen Lösung ein Algorithmus sowie ein Beweis für die Terminierung nach maximal fünf Iterationen in [ZC91] enthalten ist. Analog zur Flowanalyse in [MMG<sup>+</sup>07] können wir unseren Algorithmus zur Flowanalyse auf den Algorithmus in [ZC91] zurückführen und damit beweisen, dass die Flowanalyse korrekt ist und terminiert. Auf die Ausführung dieses Beweises verzichten wir in der vorliegenden Arbeit.

Im Folgenden beschreiben wir die Algorithmen, die vom Algorithmus `analyseCSSA()` verwendet werden. Die Hilfsalgorithmen `teilgraphFlow()`, `startZyklus()` und `teilgraphSchleife()` sind im Anhang B definiert.

#### 4.1.2. analysePi()

Der Algorithmus `analysePi()` analysiert einen  $\pi$ -Knoten, der in einem CSSA-Graph enthalten ist. Dabei betrachtet er alle Variablen, die im  $\pi$ -Knoten definiert sind. Für jede solche Variable ermittelt der Algorithmus den Wertebereich, indem die Wertebereiche der zugeordneten Variablen vereinigt werden. Der Algorithmus wird vom Algorithmus `analyseCSSA()` aufgerufen.

Der Algorithmus `analysePi()` ist in Algorithmus 2 definiert. Beim Aufruf werden dem Algorithmus zwei Parameter übergeben. Der erste Parameter enthält den  $\pi$ -Knoten, der analysiert werden soll. Der zweite Parameter enthält die Menge der Wertebereiche, die bereits im Algorithmus `analyseCSSA()` berechnet und gespeichert wurden.

Ein  $\pi$ -Knoten repräsentiert den parallelen Zugriff auf Daten. Hier müssen wir beachten, dass der Kontrollfluss trotz der parallelen Ausführung nur einmal an eine bestimmte Stelle im BPEL-Prozess gelangt. Bei der Berechnung des Wertebereich einer definierten Variable vereinigen wir deshalb nur die Wertebereiche, die nicht mit der definierten Variablen berechnet wurden.

---

#### Algorithmus 2 analysePi(knoten, Werteliste)

---

```

1: OUT  $\leftarrow$  Werteliste
2: for all Variable x, die im knoten definiert wird do
3:   W  $\leftarrow$   $\emptyset$ 
4:   for all Variable y, die der aktuellen Variable x zugeordnet ist do
5:     if (y, Wy)  $\in$  Werteliste then
6:       for all (z, Besuchtz)  $\in$  Wy do
7:         if x  $\notin$  Besuchtz then
8:           W  $\leftarrow$  W  $\cup$  {(z, Besuchtz  $\cup$  {y})}
9:         end if
10:      end for
11:    end if
12:  end for
13:  if (x, Walt)  $\in$  OUT then
14:    OUT  $\leftarrow$  (OUT  $\setminus$  {(x, Walt)})  $\cup$  {(x, W)}
15:  else
16:    OUT  $\leftarrow$  OUT  $\cup$  {(x, W)}
17:  end if
18: end for
19: return OUT

```

---

## Beschreibung

In Zeile 3 initialisieren wir für jede Variable  $x$ , die im  $\pi$ -Knoten definiert wird, den Wertebereich mit der leeren Menge. Danach überprüfen wir in Zeile 5 für jede Variable  $y$ , die der Variablen in  $x$  zugeordnet ist, ob der Wertebereich von  $y$  bereits berechnet wurde. Wurde der Wertebereich  $W_y$  der Variablen  $y$  bereits berechnet, betrachten wir in den Zeilen 6 – 10 jedes Element  $(z, \text{Besucht}_z)$  im Wertebereich  $W_y$ . Die Bedingung in Zeile 7 überprüft, ob das Element  $(z, \text{Besucht}_z)$  in den Wertebereich für die Variable  $x$  aufgenommen werden darf. Wie bereits erwähnt, darf das Element nur dann aufgenommen werden, wenn die Variable  $x$  bei der Berechnung von  $z$  noch nicht verwendet wurde. Gegebenfalls erweitern wir in Zeile 8 den Wertebereich der Variablen  $x$  um das Element  $(z, \text{Besucht}_z)$ . Da wir bei der Erweiterung die Variable  $y$  verwendet haben, erweitern wir auch  $\text{Besucht}_z$  um die Variable  $y$ .

Haben wir jede Variable  $y$ , die der Variablen  $x$  zugeordnet ist, betrachtet, speichern wir in den Zeilen 13 – 17 das ermittelte Ergebnis. Dabei überprüfen wir, ob bereits ein Wertebereich für die Variable  $x$  gespeichert ist. Ist dies der Fall, ersetzen wir in Zeile 14 den alten Wertebereich mit dem neuen berechneten Wertebereich. Sonst speichern wir den ermittelten Wertebereich in Zeile 16. Nun betrachten wir die nächste Variable  $x$ , die im  $\pi$ -Knoten definiert wird. Wurden alle Variablen  $x$  betrachtet, ist die Analyse des  $\pi$ -Knotens beendet. In Zeile 19 terminiert der Algorithmus, nachdem die aktualisierte Menge der Wertebereiche zurückgegeben wurde.

## Terminierung und Korrektheit

Der Algorithmus 2 analysiert jede im  $\pi$ -Knoten definierte Variable. Für eine definierte Variable ermittelt der Algorithmus den Wertebereich mit Hilfe der zugeordneten Variablen. Die Anzahl der im  $\pi$ -Knoten definierten Variablen ist endlich. Da auch die Anzahl der zugeordneten Variablen endlich ist, terminiert der Algorithmus.

Der Algorithmus ist korrekt, da beim parallelen Datenzugriff der tatsächliche Kontrollfluss beachtet wird. Der parallele Datenzugriff an einer bestimmten Stelle im BPEL-Prozess erlaubt keine Werte, bei dem der Kontrollfluss bereits bei der betrachteten Stelle im BPEL-Prozess war. In einem  $\pi$ -Knoten dürfen wir deshalb nur die Werte der anderen Pfade berücksichtigen, nicht aber den Wert des eigenen Pfades. Dies wird im Algorithmus dadurch realisiert, dass einer definierten Variable nur die Wertebereiche zugewiesen werden, bei deren Berechnung die definierte Variable noch nicht verwendet wurde.

### 4.1.3. analysePhi()

Der Algorithmus `analysePhi()` analysiert einen  $\phi$ -Knoten, der in einem CSSA-Graph enthalten ist. Dabei betrachtet er alle Variablen, die im  $\phi$ -Knoten definiert sind. Für jede solche Variable ermittelt der Algorithmus den Wertebereich, indem die Wertebereiche

der zugeordneten Variablen vereinigt werden. Der Algorithmus wird vom Algorithmus `analyseCSSA()` aufgerufen.

Der Algorithmus `analysePhi()` ist in Algorithmus 3 definiert. Beim Aufruf werden dem Algorithmus zwei Parameter übergeben. Der erste Parameter enthält den  $\phi$ -Knoten, der analysiert werden soll. Der zweite Parameter enthält die Menge der Wertebereiche, die bereits im Algorithmus `analyseCSSA()` berechnet und gespeichert wurden.

---

**Algorithmus 3** `analysePhi(knoten, Werteliste)`

---

```
1: OUT  $\leftarrow$  Werteliste
2: for all Variable x, die im knoten definiert wird do
3:   W  $\leftarrow$   $\emptyset$ 
4:   for all Variable y, die der aktuellen Variable x zugeordnet ist do
5:     if  $(y, W_y) \in$  Werteliste then
6:       for all  $(z, \text{Besucht}_z) \in W_y$  do
7:         W  $\leftarrow$   $W \cup \{(z, \text{Besucht}_z \cup \{y\})\}$ 
8:       end for
9:     end if
10:  end for
11:  if  $(x, W_{\text{alt}}) \in$  OUT then
12:    OUT  $\leftarrow$   $(\text{OUT} \setminus \{(x, W_{\text{alt}})\}) \cup \{(x, W)\}$ 
13:  else
14:    OUT  $\leftarrow$   $\text{OUT} \cup \{(x, W)\}$ 
15:  end if
16: end for
17: return OUT
```

---

**Beschreibung**

Der Algorithmus arbeitet genauso wie der bereits vorgestellte Algorithmus `analysePi()`. Es fehlt lediglich die Bedingung, die überprüft, ob das Element  $(z, \text{Besucht}_z)$  im Wertebereich auftreten darf. Da wir in einem  $\phi$ -Knoten keinen parallelen Kontrollfluss beachten müssen, erweitern wir den Wertebereich einer Variablen um jedes Element  $(z, \text{Besucht}_z)$  in  $W_y$ .

**Terminierung und Korrektheit**

Der Algorithmus 3 terminiert, wobei die Argumentation analog zum Algorithmus `analysePi()` ist. Der Algorithmus ist korrekt, da wir bei einem  $\phi$ -Knoten keine parallelen Datenzugriffe berücksichtigen müssen. Bei der Berechnung eines Wertebereichs für eine definierte Variable darf der Algorithmus demnach jeden zugeordneten Wertebereich verwenden.

#### 4.1.4. widening()

Der Algorithmus `widening()` übernimmt das `widening` für den Algorithmus `analyseCSSA()`. Beim `widening` abstrahieren wir von einem konkreten Datenwert und verwenden stattdessen eine untere bzw. obere Grenze für den Datenwert. Das `widening` ist bei der Analyse von Schleifen nötig. Bei der statischen Analyse von Schleifen kann die Anzahl der Iterationen unendlich sein. In diesem Fall endet die Analyse nie, weil der berechnete Wertebereich sich nie stabilisiert, d.h. nie gleich bleibt. Um unendlich viele Iterationen zu vermeiden, abstrahieren wir nach einer endlichen Anzahl von Iterationen von dem konkreten Datenwert und verwenden eine untere bzw. obere Grenze für den Datenwert.

In unserer Analyse wenden wir das `widening` nur für Zahlen an. Dabei ersetzen wir den konkreten Zahlenwert, der innerhalb der Schleife berechnet wird, durch  $-\infty$  bzw.  $\infty$ . Eine unendliche Anzahl von Iterationen entsteht, wenn wir bei der Berechnung eines Wertes innerhalb der Schleife einen Wert verwenden, der sich aus der vorherigen Iteration der Schleife ergibt. Deshalb überprüfen wir nach der ersten Iteration über eine Schleife, ob ein `widening` vorgenommen werden muss. Dazu überprüfen wir bei der Berechnung des Wertebereichs einer Variablen, ob diese Variable bereits in der Menge der verwendeten Variablen enthalten ist. Ist dies der Fall, wissen wir, dass wir eine unendliche Anzahl von Iterationen erzeugen würden. Um die unendliche Iteration zu vermeiden, nehmen wir in diesem Fall das `widening` vor. Danach ist der Wertebereich stabil und eine weitere Iteration ist für diese Variable unnötig. Allerdings müssen wir beachten, dass sich das `widening` auf andere Variablen auswirken kann. Deshalb iteriert der Algorithmus `analyseCSSA()` nach der Anwendung des `widenings` solange über die Schleife, bis sich kein Wertebereich mehr ändert.

Bei den Datentypen `boolean`, `string` und `node-set` ist das `widening` nicht nötig, da der Zustandsraum ihrer Wertebereiche endlich ist. Somit entsteht bei ihrer Analyse keine unendliche Anzahl von Iterationen. Beispielsweise lässt der Datentyp `boolean` maximal die Werte `{true}` und `{false}` zu. Deshalb können die Wertebereiche `{true}`, `{false}` und `{true, false}` in einer endlichen Anzahl von Iterationen berechnet werden. Auch die Mengen der verwendeten Variablen werden in einer endlichen Anzahl von Iterationen berechnet. Bei jeder Iteration über die Schleife, wird die gleiche Variable in die Menge der verwendeten Variablen aufgenommen. Somit stabilisiert sich sowohl der Wertebereich als auch die Menge der verwendeten Variablen für den Datentyp `boolean`.

Da wir bei Zeichenketten von der Reihenfolge und Anzahl der Zeichen in einer Zeichenkette abstrahieren, stabilisiert sich auch der Wertebereich für Zeichenketten nach einer endlichen Anzahl von Iterationen. So kann eine konkrete Zeichenkette immer länger werden, z.B. durch Verwendung der Funktion `concat()`, allerdings werden nach einer endlichen Anzahl von Iterationen keine neuen Zeichen mehr in den Wertebereich hinzugefügt. Wenn wir eine Zeichenkette, die sich aus der vorhergehenden Iteration ergibt, mit einer zweiten Zeichenkette konkatenieren, werden nur die Zeichen der zweiten Zeichenkette hinzugefügt. Diese zweite Zeichenkette ist entweder konstant oder sie ergibt sich durch Konvertierung bzw. aus der vorhergehenden Iteration. Wenn die zweite Zeichenkette

durch Konvertierung entsteht, ist der Wertebereich nach endlich vielen Iterationen stabil, da die Wertebereiche der Datentypen `boolean`, `numbers` und `node-set` nach endlich vielen Iterationen stabil sind. Wenn die Zeichenkette in der vorhergehenden Iteration berechnet wurde, können wir die Argumentation für die erste Zeichenkette erneut auf die zweite Zeichenkette anwenden. Dabei gelangen wir irgendwann zu einer Zeichenkette, die entweder konstant ist oder durch Konvertierung entsteht. Wie beim Datentyp `boolean` stabilisiert die Menge der verwendeten Variablen ebenfalls nach endlich vielen Schritten.

Der Wertebereich einer Variablen vom Typ Knotenmenge ist nach endlich vielen Iterationen stabil, da wir auf einer endlichen Anzahl von endlichen Bäumen operieren. Eine Knotenmenge wird durch einen Lokalisierungspfad ermittelt. Dieser ermittelt die Knotenmenge über Achsen und Knotentests (siehe Kapitel 2.3.5). Der Knotentest verringert lediglich die Knoten in der Knotenmenge. Da wir bei jeder Iteration die gleichen Lokalisierungspfade haben, betrachten wir immer die gleichen Achsen im Baum. Das bedeutet, dass wir nach endlich vielen Schritten an einem Wurzelknoten oder an einem Blattknoten ankommen, je nachdem welche Richtung die Achse angibt. Selbst wenn die Mengen über jede Iteration vereinigt werden, gelangen wir irgendwann an einen Punkt, an dem alle Knoten des Baums in der Menge enthalten sind. Dann ist die Knotenmenge stabil, da kein weiterer Knoten in der Menge aufgenommen werden kann. Wie beim Datentyp `boolean` stabilisiert die Menge der verwendeten Variablen ebenfalls nach endlich vielen Schritten.

Der Algorithmus `widening()` ist in Algorithmus 4 definiert. Beim Aufruf werden dem Algorithmus zwei Parameter übergeben. Der erste Parameter enthält die Wertebereiche, die nach der ersten Analyse des  $\phi$ -Knotens berechnet wurde. Der  $\phi$ -Knoten repräsentiert dabei das Zusammenführen des Kontrollflusses vor der Schleife und des Kontrollflusses aus dem Schleifenkörper. Somit enthält der erste Parameter die Wertebereiche, die vor der ersten Iteration über die Schleife berechnet wurden. Der zweite Parameter enthält die Wertebereiche nach der zweiten Analyse des  $\phi$ -Knotens. Der  $\phi$ -Knoten wurde nach der ersten Iteration über die Schleife ein zweites Mal analysiert.

## Beschreibung

Im Algorithmus vergleichen wir jedes Element, das im ersten Parameter enthalten ist, mit jedem Element, das im zweiten Parameter enthalten ist. Deshalb werden in den Zeilen 2 und 3 zwei Schleifen definiert, die diesen Vergleich ermöglichen. Danach stellen wir in Zeile 4 sicher, dass wir nur die Wertebereiche miteinander vergleichen, die der selben Variablen  $x$  bzw.  $y$  zugeordnet sind. Dadurch wird ebenfalls sicher gestellt, dass wir keine Variablen betrachten, die im Schleifenkörper definiert werden. Da der Wertebereich dieser Variablen erst während der ersten Iteration über die Schleife berechnet wird, sind diese Variablen nicht im ersten Parameter enthalten. Für die Variablen, die im Schleifenkörper definiert sind, nehmen wir kein explizites `widening` vor, da sich das `widening` für Variablen in  $\phi$ -Knoten auf diese Variablen bei der nächsten Iteration über die Schleife auswirken.

---

**Algorithmus 4** widening( $W_{\text{Phi1}}, W_{\text{Phi2}}$ )

---

```

1: OUT  $\leftarrow \emptyset$ 
2: for all  $(x, W_x) \in W_{\text{Phi1}}$  do
3:   for all  $(y, W_y) \in W_{\text{Phi2}}$  do
4:     if  $x = y$  then
5:        $W_{\text{dif}} \leftarrow W_y \setminus W_x$ 
6:        $W_{\text{neu}} \leftarrow \emptyset$ 
7:       if  $W_{\text{dif}} \neq \emptyset$  then
8:         for all  $(a, \text{Besucht}_a) \in W_x$  do
9:           if  $a \in \text{Intervalle}$  then
10:            for all  $(b, \text{Besucht}_b) \in W_{\text{dif}}$  do
11:              if  $x \in \text{Besucht}_b$  then
12:                 $[a_1, a_2] \leftarrow a$ 
13:                 $[b_1, b_2] \leftarrow b$ 
14:                if  $a_1 \leq b_1$  then
15:                   $c_1 \leftarrow a_1$ 
16:                else
17:                   $c_1 \leftarrow -\infty$ 
18:                end if
19:                if  $a_2 \geq b_2$  then
20:                   $c_1 \leftarrow a_2$ 
21:                else
22:                   $c_1 \leftarrow \infty$ 
23:                end if
24:                 $W_{\text{neu}} \leftarrow W_{\text{neu}} \cup \{([c_1, c_2], \text{Besucht}_b)\}$ 
25:              else
26:                 $W_{\text{neu}} \leftarrow W_{\text{neu}} \cup \{(b, \text{Besucht}_b)\}$ 
27:              end if
28:            end for
29:          else
30:             $W_{\text{neu}} \leftarrow W_{\text{neu}} \cup W_{\text{dif}} \cup \{(a, \text{Besucht}_a)\}$ 
31:          end if
32:        end for
33:         $\text{OUT} \leftarrow \text{OUT} \cup \{(x, W_{\text{neu}})\}$ 
34:      else
35:         $\text{OUT} \leftarrow \text{OUT} \cup \{(x, W_x)\}$ 
36:      end if
37:    end if
38:  end for
39: end for
40: return OUT

```

---

In Zeile 5 ermitteln wir nun die Elemente, die durch die erste Iteration über die Schleife entstanden sind und speichern diese Elemente in  $W_{dif}$ . Ist die Menge  $W_{dif}$  leer, betrachten wir eine Variable, deren Wertebereich bereits vor der Schleife berechnet wurde. Für diese Variablen ist ein widening unnötig. Betrachten wir eine Variable, deren Wertebereich von der Schleife abhängt, werden die Zeilen 7 – 33 ausgeführt. Hier operieren wir über jedes Element  $(a, Besucht_a)$ , das im Wertebereich des ersten Parameters enthalten ist. Da wir das widening nur für Zahlen anwenden, wird in Zeile 8 überprüft, ob  $a$  ein Intervall ist. Ist das der Fall, ermitteln wir in Zeile 10 jedes Element  $(b, Besucht_b)$ , das im Wertebereich  $W_{dif}$  enthalten ist. In Zeile 11 überprüfen wir nun, ob ein widening vorgenommen werden muss. Ist die Variable, dessen Wertebereich wir überprüfen möchten, bereits in der Menge der verwendeten Variablen enthalten, so nehmen wir in den Zeilen 14 – 23 das widening vor. Die Zeilen 14 – 18 vergleichen dabei die linken Grenzen der Intervalle  $a$  und  $b$ . Ist die linke Grenze vor der ersten Iteration kleiner oder gleich der linken Grenze nach der ersten Iteration, so bedeutet das, dass die linke Intervallgrenze in der Schleife nicht kleiner wird, als vor der Ausführung der Schleife. In diesem Fall speichern wir in Zeile 15 die linke Grenze des Intervalls  $a$  als neue linke Intervallgrenze. Ist die linke Grenze vor der ersten Iteration größer als die linke Grenze nach der ersten Iteration, so bedeutet das, dass die linke Grenze bei jeder Iteration kleiner wird. In diesem Fall speichern wir in Zeile 17 „ $-\infty$ “ als neue linke Intervallgrenze. Analog überprüfen wir in den Zeilen 19 – 23, ob die rechte Grenze durch die Iterationen über die Schleife immer größer wird. Ist dies der Fall, nehmen wir das widening vor und speichern in Zeile 22 „ $\infty$ “ als neue rechte Intervallgrenze. Schließlich speichern wir in Zeile 24 das neu berechnete Intervall in der Menge  $W_{neu}$ .

Müssen wir kein widening vornehmen, speichern wir in Zeile 26 das alte Intervall  $b$  in der Menge  $W_{neu}$ . Ist  $a$  kein Intervall, ist ein widening ebenfalls unnötig und wir speichern in Zeile 31 das Element  $(a, Besucht_a)$  sowie die Menge  $W_{dif}$  in der Menge  $W_{neu}$ . Haben wir den Vergleich für alle Elemente  $(a, Besucht_a)$  vorgenommen, die im Wertebereich des ersten Parameters enthalten sind, haben wir den Wertebereich der Variablen  $x$  bzw.  $y$  vollständig berechnet. In Zeile 33 speichern wir den berechneten Wertebereich  $W_{neu}$  für die Variable  $x$  in der Menge  $OUT$ .

Betrachten wir eine Variable, deren Wertebereich bereits vor der Schleife berechnet wurde, speichern wir in Zeile 35 diesen Wertebereich in der Menge  $OUT$ . Wurden auch alle Elemente des ersten Parameters betrachtet, ist die Menge  $OUT$  vollständig berechnet. Diese Menge enthält den Wertebereich nach Ausführung der ersten Iteration, wobei für diesen Wertebereich das widening vorgenommen wurde. Schließlich geben wir in Zeile 40 die Menge  $OUT$  zurück und der Algorithmus terminiert.

### Terminierung und Korrektheit

Der Algorithmus 4 betrachtet jede Variable und dessen Wertebereich, die in den Parametern übergeben werden. Der erste Parameter enthält eine Menge von Variablen und dessen Wertebereiche, die vor der ersten Iteration über eine Schleife berechnet wurden.

Der zweite Parameter enthält eine Menge von Variablen und dessen Wertebereiche, die nach der ersten Iteration über eine Schleife berechnet wurden. Der Algorithmus überprüft für jede Zahlenvariable, die sowohl im ersten als auch im zweiten Parameter enthalten ist, ob die widening-Operationen angewendet werden muss.

Die Mengen, die als Parameter übergeben werden, sind endlich. Da auch der Wertebereich für jede betrachtete Variable eine endliche Menge ist, terminiert der Algorithmus.

Der Algorithmus ist korrekt, da wir die widening-Operation nur dann anwenden, wenn eine unendliche Anzahl von Iterationen entstehen kann. Dies ist der Fall, wenn ein Variablenwert in einer Schleife von dem Variablenwert der vorherigen Schleifeniteration abhängig ist. Die Betrachtung dieses Falles genügt, da sich das widening in diesen Fall auf indirekte Variablenabhängigkeiten auswirkt, wenn danach erneut über die Schleife iteriert wird. Bei der Anwendung einer widening-Operation beachten wir, dass die verschiedenen Wertebereiche *einer* Variablen verglichen werden müssen. Deshalb vergleichen wir keine Wertebereiche unterschiedlicher Variablen. Da bei den Datentypen `boolean`, `string` und `node-set` kein widening notwendig ist, genügt die Betrachtung von Zahlen.

Die widening-Operation ist korrekt, da die Intervallgrenzen korrekt „erweitert“ werden. Bei Zahlenwerten, die in der Schleife kleiner werden, ändern wir die linke Intervallgrenze. Bei Zahlen, die in der Schleife größer werden, ändern wir die rechte Intervallgrenze. Da wir in diesen Fällen den Intervallgrenzen die Werte  $-\infty$  bzw.  $\infty$  zuweisen, ändert sich der Wertebereich in den folgenden Iterationen nicht mehr.

#### 4.1.5. speichern()

Der Algorithmus `speichern()` übernimmt das Speichern eines Wertebereichs, der im Algorithmus `analyseCSSA()` für eine bestimmte Variable berechnet wurde. Analysiert der Algorithmus `analyseCSSA()` einen Flow oder eine Schleife, überprüft der Algorithmus zusätzlich, ob sich der Wertebereich der betrachteten Variable geändert hat. Als Ergebnis liefert der Algorithmus `speichern()` ein Tupel. Das erste Element des Tupels enthält die aktualisierte Menge der Wertebereiche. Falls der Algorithmus `analyseCSSA()` einen Flow analysiert und der Wertebereich der betrachteten Variable verändert wurde, enthält das zweite Element den Knoten, der gerade vom Algorithmus `analyseCSSA()` analysiert wird. Sonst enthält das zweite Element die leere Menge.

Der Algorithmus `speichern()` ist in Algorithmus 5 definiert. Beim Aufruf werden dem Algorithmus sechs Parameter übergeben. Der erste Parameter enthält die Variable, deren Wertebereich gespeichert werden soll. Der zweite Parameter enthält den Wertebereich, der gespeichert werden soll. Der dritte Parameter enthält die Menge der Wertebereiche, die bereits vom Algorithmus `analyseCSSA()` berechnet und gespeichert wurden. Der vierte Parameter enthält den Knoten, der gerade im Algorithmus `analyseCSSA()` analysiert wird. Der fünfte Parameter enthält den Wahrheitswert, ob im Algorithmus `analyseCSSA()` gerade einen Flow analysiert wird. Der sechste Parameter enthält den Wahrheitswert, ob im Algorithmus `analyseCSSA()` gerade eine Schleife analysiert wird.

---

**Algorithmus 5** speichern( $\text{var}$ ,  $W$ ,  $\text{OUT}$ ,  $\text{aktuell}$ ,  $\text{schleife}$ )

---

```

1: Nochmal  $\leftarrow \emptyset$ 
2: if  $\text{var} \neq \emptyset$  then
3:   if  $\text{inflow} \vee \text{schleife}$  then
4:     if  $(\text{var}, W_{\text{var}}) \in \text{OUT}$  then
5:       if  $W_{\text{var}} \neq W$  then
6:          $\text{OUT} \leftarrow \text{OUT} \setminus \{(\text{var}, W_{\text{var}})\}$ 
7:         if  $\text{inflow}$  then
8:            $\text{Nochmal} \leftarrow \{\text{aktuell}\}$ 
9:         end if
10:        end if
11:       else
12:         if  $\text{inflow}$  then
13:            $\text{Nochmal} \leftarrow \{\text{aktuell}\}$ 
14:         end if
15:       end if
16:     end if
17:      $\text{OUT} \leftarrow \text{OUT} \cup \{(\text{var}, W)\}$ 
18:   end if
19: return ( $\text{OUT}$ ,  $\text{Nochmal}$ )

```

---

### Beschreibung

In Zeile 2 überprüfen wir zunächst, ob der Wertebereich  $W$  gespeichert werden muss. Wurde dem ersten Parameter ein Inhalt ungleich der leeren Menge zugewiesen, speichern wir den Wertebereich in den Zeilen 3 – 13. Zunächst überprüfen wir dabei in Zeile 3, ob der Algorithmus `analyseCSSA()` einen Flow oder eine Schleife analysiert. Ist dies der Fall, wird in den Zeilen 4 – 11 überprüft, ob sich der Wertebereich der betrachteten Variablen verändert hat. Dazu ermitteln wir in Zeile 4, ob der dritte Parameter bereits einen Eintrag für die betrachtete Variable enthält. Ist ein Eintrag im dritten Parameter enthalten und der gespeicherte Wertebereich unterscheidet sich vom neu berechneten Wertebereich, entfernen wir in Zeile 6 den Eintrag aus dem dritten Parameter. (Der Eintrag wird später in Zeile 13 ersetzt.) Analysiert der Algorithmus `analyseCSSA()` einen Flow, speichern wir in Zeile 8 außerdem den vierten Parameter in der Menge *Nochmal*. Ist im dritten Parameter noch kein Eintrag für die betrachtete Variable gespeichert, ändert sich der Wertebereich ebenfalls. Deshalb speichern wir auch in Zeile 13 den vierten Parameter in der Menge *Nochmal*, wenn der Algorithmus `analyseCSSA()` einen Flow analysiert.

In Zeile 17 wird nun der Wertebereich, der im zweiten Argument übergeben wurde, für die betrachtete Variable gespeichert. Schließlich geben wir in Zeile 19 die aktualisierte Menge der Wertebereiche sowie die Menge *Nochmal* zurück. Danach terminiert der Algorithmus.

## Terminierung und Korrektheit

Der Algorithmus 5 speichert den berechneten Wertebereich einer Variablen. Wurde der Wertebereich in einer Schleife oder einem Flow berechnet, überprüft der Algorithmus zusätzlich, ob bereits ein Wertebereich für diese Variable gespeichert ist. Ist dies der Fall, wird der alte Wertebereich durch den neuen berechneten Wertebereich ersetzt. Zusätzlich wird in diesem Fall der vierte Parameter zurückgegeben, wenn der Wertebereich in einem Flow berechnet wurde.

Der Algorithmus terminiert, da er sequentiell ist. Weiterhin ist der Algorithmus korrekt, da er die Variable und den berechneten Wertebereich in der korrekten Form speichert. Ist bereits ein Wertebereich für die Variable gespeichert, wird der Wertebereich durch den neu berechneten Wertebereich ersetzt. Gegebenfalls wird auch der vierte Parameter zurückgegeben.

### 4.1.6. zusammenfassen()

Der Algorithmus `zusammenfassen()` fasst den Wertebereich für jede Variable einer übergebenen Menge zusammen. Der Algorithmus wird vom Algorithmus `analyseCSSA()` aufgerufen, wenn der BPEL-Prozess vollständig analysiert wurde. Der Algorithmus `analyseCSSA()` speichert den Wertebereich einer Variablen als Menge von Tupeln  $(y, M)$ . Dabei repräsentiert  $y$  einen berechneten Teil des Wertebereichs und  $M$  eine Menge von Variablen, die bei der Berechnung von  $y$  verwendet wurden. Der Algorithmus `zusammenfassen()` fasst die berechneten Teile des Wertebereichs zusammen und entfernt die Menge der verwendeten Variablen. Das Ergebnis ist für jede Variable ein Element der vollständigen Verbände aus den Definitionen 1, 7 und 18 oder eine Knotenmenge  $K$ , wie in Definition 21 definiert.

Der Algorithmus `zusammenfassen()` ist in Algorithmus 6 definiert. Beim Aufruf wird dem Algorithmus ein Parameter übergeben. Dieser Parameter enthält die Wertebereiche, die vom Algorithmus `analyseCSSA()` berechnet wurden.

Der Algorithmus `analyseCSSA()` übergibt dem Algorithmus `zusammenfassen()` beim Aufruf eine Menge von Tupeln  $(x, W_x)$  als Parameter. Ein solches Tupel repräsentiert eine Variable  $x$  im CSSA-Graph und deren Wertebereich  $W_x$ . Wie bereits erwähnt, ist der Wertebereich  $W_x$  eine Menge von Tupeln  $(y, M)$ . Möchten wir die Teile  $y$  des Wertebereichs einer Variablen  $x$  zusammenfassen, vereinigen wir alle Elemente  $y$ , die in  $W_x$  enthalten sind. Der Vereinigungsoperator ist dabei vom Datentyp abhängig. Werden Intervalle vereinigt, wird der in Kapitel 3.1.2 definierte Operator  $\sqcup_{Int}$  verwendet. Analog werden bei Zeichenkettenvereinigungen bzw. Vereinigungen von booleschen Wertebereichen die Operatoren  $\sqcup_Z$  bzw.  $\sqcup_{bool}$  verwendet. Bei der Vereinigung von Knotenmengen verwenden wir die Vereinigungsoperation für Mengen.

---

**Algorithmus 6** zusammenfassen(Werteliste)

---

```

1: OUT  $\leftarrow \emptyset$ 
2: for all  $(x, W_x) \in \text{Werteliste}$  do
3:   Tmp  $\leftarrow \emptyset$ 
4:   for all  $(y, M) \in W_x$  do
5:     Tmp  $\leftarrow \text{Tmp} \cup y$ 
6:   end for
7:   OUT  $\leftarrow \text{OUT} \cup \{(x, \text{Tmp})\}$ 
8: end for
9: return OUT

```

---

**Beschreibung**

In den Zeilen 2 – 8 betrachten wir jede Variable  $x$ , für die der Algorithmus `analyseCS-SA()` den Wertebereich  $W_x$  berechnet hat. In den Zeilen 4 – 6 ermitteln wir nun jedes Tupel  $(y, M)$ , das der Variablen  $x$  zugeordnet ist. In Zeile 5 vereinigen wir die Teile des berechneten Wertebereichs und speichern das Ergebnis in der Menge `Tmp`. Haben wir alle Elemente  $(y, M)$  für eine Variable  $x$  betrachtet, ist der Wertebereich für die Variable  $x$  zusammengefasst. In Zeile 7 speichern wir den zusammengefassten Wertebereich für die Variable  $x$  in der Ergebnismenge `OUT`. Haben wir jede Variable betrachtet, die im Parameter enthalten ist, ist die Ergebnismenge vollständig berechnet und wir geben sie in Zeile 9 zurück.

**Terminierung und Korrektheit**

Der Algorithmus 5 fasst den Wertebereich für jede Variable in der übergebenen Menge zusammen. Der Wertebereich einer Variablen ist als Menge angegeben. Diese Menge wird zusammengefasst.

Der Algorithmus terminiert, da die übergebene Menge sowie die Mengen der Wertebereiche endlich sind. Der Algorithmus ist korrekt, da jede Variable in der übergebenen Menge betrachtet wird. Weiterhin wird für eine Variable jedes Element im Wertebereich berücksichtigt.

**4.2. Analyse der XPath-Ausdrücke**

In diesem Kapitel beschreiben wir eine statische Analyse für XPath-Ausdrücke. Dazu definieren wir den Algorithmus `analyseXPath()`. Der Algorithmus beschreibt die abstrakte Interpretation eines XPath-Ausdrucks und berechnet auf Grundlage des Datenmodells aus Kapitel 3.1 den Wertebereich, den der XPath-Ausdruck liefern kann. Da wir eine sichere Analyse vorstellen, ist jedes Ergebnis, das bei der Auswertung des XPath-

Ausdrucks zur Laufzeit entsteht, im Ergebnis der Analyse enthalten. Allerdings kann die Analyse auch Ergebnisse liefern, die nicht bei der Auswertung zur Laufzeit entstehen.

Ein XPath-Ausdruck ist in Form des Drei-Adress-Codes repräsentiert, wie wir in Kapitel 3.2 beschrieben haben. Der Analyse steht ein XPath-Ausdruck somit in Form einer Menge von Quadrupeln zur Verfügung. Ein Quadrupel repräsentiert eine Anweisung im XPath-Ausdruck, die maximal einen Operator und maximal zwei Operanden enthält. Das erste Element des Quadrupels repräsentiert die temporäre Variable, in der das Ergebnis der Anweisung gespeichert wird. Das zweite Element des Quadrupels repräsentiert die Operation. Das dritte und vierte Element des Quadrupels repräsentieren den ersten und zweiten Operanden.

In unserer Analyse werden konkrete Zahlen durch Intervalle repräsentiert. Das kleinste Element, das bei der Analyse eines XPath-Ausdrucks entsteht, ist das leere Intervall vereinigt mit dem speziellen Wert *NaN*. Das leere Intervall (ohne den speziellen Wert *NaN*) wird bei der Analyse eines XPath-Ausdrucks nie erzeugt. Deshalb berücksichtigen wir das leere Intervall bei der Analyse von XPath-Ausdrücken nicht. Somit dient das leere Intervall lediglich der Definition des vollständigen Verbandes für Intervalle in Definition 7. Dort benötigen wir ein Null-Element, das durch das leere Intervall repräsentiert wird.

#### 4.2.1. analyseXPath()

Der Algorithmus `analyseXPath()` berechnet den Wertebereich eines XPath-Ausdrucks. Werden innerhalb des XPath-Ausdrucks Variablen verwendet, verwenden wir die berechneten Wertebereiche der Variablen. Der Wertebereich einer Variablen wird durch eine Menge von Tupeln repräsentiert. Deshalb berechnet der Algorithmus das Kreuzprodukt der Wertebereiche, für die jeweiligen Variablen. Der Algorithmus `analyseXPath()` wird vom Algorithmus `analyseCSSA()` aufgerufen, um Wertebereiche von BPEL-Variablen und Bedingungen zu ermitteln.

Der Algorithmus `analyseXPath()` ist in Algorithmus 7 definiert. Beim Aufruf werden dem Algorithmus zwei Parameter übergeben. Der erste Parameter enthält den XPath-Ausdruck, der analysiert werden soll. Der XPath-Ausdruck ist in Form des Drei-Adress-Codes repräsentiert. Der zweite Parameter enthält die Menge der Wertebereiche, die bereits durch den Algorithmus `analyseCSSA()` berechnet wurden.

#### Beschreibung

In den ersten beiden Zeilen initialisieren wir die Variablen, die im Algorithmus benötigt werden. Um den XPath-Ausdruck vollständig analysieren zu können, speichern wir zunächst eine Kopie des XPath-Ausdrucks in der Variablen *MeineListe*. Damit wir die Elemente in *MeineListe* in der richtigen Reihenfolge bearbeiten können, definieren wir zusätzlich eine Variable *zähler*. Die Menge *Wertebereich* initialisieren wir mit der leeren

---

**Algorithmus 7:** analyseXPath(Ausdruck, Werteliste)

---

```

1: MeineListe  $\leftarrow$  Ausdruck, zähler  $\leftarrow$  1, Wertebereich  $\leftarrow$   $\emptyset$ 
2: TmpListe  $\leftarrow$   $\emptyset$ , B  $\leftarrow$  Werteliste
3: while MeineListe  $\neq$   $\emptyset$  do
4:   (tmp_zähler, operation, operand1, operand2)  $\leftarrow$  hole Element aus MeineListe
5:   MeineListe  $\leftarrow$  MeineListe  $\setminus$  {(tmp_zähler, operation, operand1, operand2)}
6:   if operation  $\neq$  param then
7:     if operand1 ist Verweis then
8:       hole (operand1, W_operand1) aus Werteliste oder TmpListe
9:       for all (W_op1, Besucht1)  $\in$  W_operand1 do
10:        op1  $\leftarrow$  W_op1
11:        if operand2 ist Verweis then
12:          hole (operand2, W_operand2) aus Werteliste oder TmpListe
13:          for all (W_op2, Besucht2)  $\in$  W_operand2 do
14:            op2  $\leftarrow$  W_op2
15:            W_tmp  $\leftarrow$  analyseOperation(operation, op1, op2, Ausdruck, B)
16:            Besucht_tmp  $\leftarrow$  Besucht1  $\cup$  Besucht2  $\cup$  {operand1, operand2}
17:            Wertebereich  $\leftarrow$  Wertebereich  $\cup$  {(W_tmp, Besucht_tmp)}
18:          end for
19:        else
20:          W_tmp  $\leftarrow$  analyseOperation(operation, op1, operand2, Ausdruck, B)
21:          Besucht_tmp  $\leftarrow$  Besucht1  $\cup$  {operand1}
22:          Wertebereich  $\leftarrow$  Wertebereich  $\cup$  {(W_tmp, Besucht_tmp)}
23:        end if
24:      end for
25:    else
26:      if operand2 ist Verweis then
27:        hole (operand2, W_operand2) aus Werteliste oder TmpListe
28:        for all (W_op2, Besucht2)  $\in$  W_operand2 do
29:          op2  $\leftarrow$  W_op2
30:          W_tmp  $\leftarrow$  analyseOperation(operation, operand1, op2, Ausdruck, B)
31:          Besucht_tmp  $\leftarrow$  Besucht2  $\cup$  {operand2}
32:          Wertebereich  $\leftarrow$  Wertebereich  $\cup$  {(W_tmp, Besucht_tmp)}
33:        end for
34:      else
35:        if operation  $\neq$  call then
36:          W_tmp  $\leftarrow$  analyseOperation(operation, operand1, operand2, Ausdruck, B)
37:          Wertebereich  $\leftarrow$  Wertebereich  $\cup$  {(W_tmp,  $\emptyset$ )}
38:        else
39:          Wertebereich  $\leftarrow$  analyseOperation(operation, operand1, operand2,
40:                                             Ausdruck, B)

```

---

```
41:     end if
42: end if
43:   TmpListe  $\leftarrow$  TmpListe  $\cup$  {(tmpzähler, Wertebereich)}
44:   B  $\leftarrow$  B  $\cup$  {(tmpzähler, Wertebereich)}
45:   Wertebereich  $\leftarrow$   $\emptyset$ 
46: end if
47:   zähler  $\leftarrow$  zähler + 1
48: end while
49: hole (tmpzähler-1, OUT) aus TmpListe
50: return OUT
```

---

Menge. In dieser Menge speichern wir den Wertebereich, der aus der Analyse eines Elements aus *MeineListe* resultiert. Die Menge *TmpListe* initialisieren wir ebenfalls mit der leeren Menge. In dieser Menge speichern wir die Wertebereiche der temporären Variablen aus dem Drei-Adress-Codes. Die Menge *B* enthält sowohl die Wertebereiche, die im Algorithmus *analyseCSSA()* ermittelt wurden, als auch die Wertebereiche der temporären Variablen des Drei-Adress-Codes. Vor der Analyse initialisieren wir die Menge *B* daher mit den Wertebereichen, die im Algorithmus *analyseCSSA()* ermittelt wurden. Um jedes Element in *MeineListe* zu bearbeiten, definieren wir in Zeile 3 eine Schleife. In den Zeilen 4 und 5 ermitteln mit Hilfe des Zählers das nächste Element, das analysiert werden muss, und entfernen es aus der Menge *MeineListe*. Die relevanten Informationen für die Analyse sind nun in den Variablen *operation*, *operand1* und *operand2* gespeichert.

Wenn das aktuelle Element keinen Parameter einer Funktion repräsentiert, berechnen wir in den Zeilen 7 – 40 das Ergebnis der aktuellen Operation. Zunächst überprüfen wir in Zeile 7, ob der erste Operand durch einen Verweis gegeben ist. Beispielsweise kann der Operand durch einen Verweis auf eine Variable gegeben sein. Ist der erste Operand durch einen Verweis auf eine temporäre Variable des Drei-Adress-Codes gegeben, ist der Wertebereich des Operanden in *TmpListe* enthalten. Ist der erste Operand durch einen Verweis auf eine BPEL-Variable gegeben, ist der Wertebereich des Operanden bereits durch den Algorithmus *analyseCSSA()* berechnet. In diesem Fall ist der Wertebereich des Operanden in *Werteliste*. In Zeile 8 ermitteln wir den Wertebereich des ersten Operanden aus den Mengen *TmpListe* oder *Werteliste*.

Da der Wertebereich des ersten Operanden durch eine Menge von Tupeln ( $W_{op1}$ , *Besucht1*) gegeben ist, definieren wir in Zeile 9 eine Schleife, die jedes Tupel im Wertebereich des ersten Operanden betrachtet. In Zeile 10 speichern wir den für die Berechnung relevanten Teil  $W_{op1}$  in der Variable *op1*. Nun überprüfen wir in der Zeile 11, ob der zweite Operand ebenfalls durch einen Verweis gegeben ist. Ist dies der Fall, verfahren wir wie beim ersten Operanden und speichern schließlich in Zeile 14 den für die Berechnung relevanten Teil  $W_{op2}$  in der Variable *op2*.

Nun können wir die Operation mit den Operanden *op1* und *op2* ausführen. Dafür rufen wir in Zeile 15 den Algorithmus `analyseOperation()` auf und speichern das Ergebnis in der Variablen *W<sub>tmp</sub>*. Weiterhin speichern wir in Zeile 16 die Menge der Variablen, die für die Berechnung des Ergebnisses verwendet wurden, in der Variablen *Besucht<sub>tmp</sub>*. Da wir bei der Berechnung des Ergebnisses die Variablen *operand1* und *operand2* verwendet haben, vereinigen wir die Mengen *Besucht1* und *Besucht2* und erweitern diese Vereinigung um die Variablen *operand1* und *operand2*. Für die Ausführung der Operation mit den Operanden *op1* und *op2* speichern wir in Zeile 17 den ermittelten Wertebereich in der Menge *Wertebereich*. Schließlich wiederholen wir die Berechnung für alle Wertebereiche, die dem zweiten Operanden zugeordnet sind. Die Wiederholung ist in Zeile 18 beendet.

Ist der zweite Operand nicht durch einen Verweis gegeben, werden die Zeilen 20 – 22 ausgeführt. Hier können wir den zweiten Operanden direkt für die Ausführung der Operation nutzen. Dementsprechend rufen wir in Zeile 20 der Algorithmus `analyseOperation()` mit den Operanden *op1* und *operand2* auf. Das Ergebnis wird in der Variablen *W<sub>tmp</sub>* gespeichert. In Zeile 21 speichern wir die Menge der Variablen, die für die Berechnung des Ergebnisses verwendet wurden, in der Variablen *Besucht<sub>tmp</sub>*. Da wir bei der Berechnung des Ergebnisses nur die Variable *operand1* verwendet haben, ergibt sich die Menge der verwendeten Variablen durch die Erweiterung von *Besucht1* um die Variable *operand1*. Für die Ausführung der Operation mit den Operanden *op1* und *operand2* speichern wir in Zeile 22 den ermittelten Wertebereich in der Menge *Wertebereich*. Schließlich wiederholen wir die Berechnung für alle Wertebereiche, die dem ersten Operanden zugeordnet sind. Ist die Wiederholung in Zeile 24 beendet, wurde die Menge *Wertebereich* für das aktuelle Element aus *MeineListe* vollständig berechnet.

Ist der erste Operand nicht durch einen Verweis gegeben, werden die Zeilen 26 – 37 ausgeführt. Dabei überprüfen wir zunächst in Zeile 26, ob der zweite Operand ein Verweis ist. Ist dies der Fall, werden in den Zeilen 27 – 33 wieder alle Wertebereiche betrachtet, die dem zweiten Operanden zugeordnet sind. Dies geschieht analog zu den Zeilen 12 – 18, die wir bereits beschrieben haben.

Ist sowohl der erste als auch der zweite Operand nicht durch einen Verweis gegeben, werden die Zeilen 35 – 40 ausgeführt. In diesem Fall müssen wir beachten, ob ein Funktionsaufruf ausgeführt werden soll. Soll kein Funktionsaufruf ausgeführt werden, berechnen wir in Zeile 36 das Ergebnis der Operation durch den Aufruf des Algorithmus `analyseOperation()` mit den Operanden *operand1* und *operand2*. Das Ergebnis speichern wir in der Variablen *W<sub>tmp</sub>*. Da wir bei der Berechnung des Ergebnisses keine Variablen verwendet haben, ist die Menge der verwendeten Variablen leer. Für die Ausführung der Operation mit den Operanden *operand1* und *operand2* speichern wir in Zeile 37 den berechneten Wertebereich in der Menge *Wertebereich*. Soll ein Funktionsaufruf ausgeführt werden, rufen wir in Zeile 39 den Algorithmus `analyseOperation()` auf. In diesem Fall liefert der Algorithmus sowohl den Wertebereich als auch die Menge der verwendeten Variablen. Deshalb speichern wir für die Ausführung eines Funktionsaufrufs das Ergebnis des Algorithmus `analyseOperation()` in *Wertebereich*.

Ist die Menge *Wertebereich* vollständig berechnet, können wir das Analyseergebnis für das aktuelle Element aus *MeineListe* in *TmpListe* speichern. Dazu erweitern wir in Zeile 43 die Menge *TmpListe* um ein Element, das der betrachteten temporären Variablen den berechneten Wertebereich zuordnet. In Zeile 44 erweitern wir zusätzlich die Menge *B* um dieses Element. Da das aktuelle Element aus *MeineListe* vollständig analysiert wurde, weisen wir in Zeile 45 der Variablen *Wertebereich* die leere Menge zu. Schließlich inkrementieren wir in Zeile 47 den Zähler, damit das nächste Element aus *MeineListe* analysiert werden kann. Ist die Analyse aller Elemente in *MeineListe* abgeschlossen, haben wir den Wertebereich des gesamten XPath-Ausdrucks ermittelt. Dieser Wertebereich wurde mit dem letzten betrachteten Element aus *MeineListe* berechnet. Deshalb ermitteln wir in Zeile 49 den Wertebereich dieses Elements aus der Menge *TmpListe*. Schließlich geben wir in Zeile 56 diesen Wertebereich zurück und der Algorithmus terminiert.

### Terminierung und Korrektheit

Der Algorithmus 7 berechnet den Wertebereich eines XPath-Ausdrucks. Werden innerhalb des XPath-Ausdrucks Variablen verwendet, werden die Wertebereiche dieser Variablen für die Berechnung verwendet. Die Menge, die den XPath-Ausdruck repräsentiert, ist endlich. Da auch die Wertebereiche der Variablen endlich sind, terminiert der Algorithmus immer.

Der Algorithmus ist korrekt, da der XPath-Ausdruck vollständig und in der richtigen Reihenfolge analysiert wird. Die Parameter einer Funktion werden nicht analysiert, damit sie einem bestimmten Funktionsaufruf zugeordnet werden können. Die Werte der Funktionsparameter werden später bei der Ausführung des Funktionsaufrufs ermittelt.

#### 4.2.2. analyseOperation()

Der Algorithmus `analyseOperation()` berechnet das Ergebnis, das bei der Ausführung einer Operation entsteht. Diese Operation hat maximal zwei Operanden. Der Algorithmus wird vom Algorithmus `analyseXPath()` aufgerufen. Deshalb beachtet der Algorithmus `analyseOperation()` die Ausführungssemantik der Operationen, wie sie in XPath oder BPEL definiert ist.

Der Algorithmus `analyseOperation` ist in Algorithmus 8 definiert. Beim Aufruf werden dem Algorithmus fünf Parameter übergeben. Der erste Parameter enthält die Operation, die ausgeführt werden soll. Der zweite und dritte Parameter enthält den ersten und zweiten Operanden. Der vierte Parameter enthält den XPath-Ausdruck, der gerade vom Algorithmus `analyseXPath()` analysiert wird. Der fünfte Parameter enthält die Wertebereiche, die bereits in `analyseXPath()` oder `analyseCSSA()` berechnet wurde. Die letzten beiden Parameter werden bei der Ausführung eines Funktionsaufrufs benötigt.

---

**Algorithmus 8** analyseOperation(operation, operand1, operand2, Ausdruck, Werte)

---

```

1: if operation = „id“ then
2:   if operand1  $\in L_1 \cup S_1 \cup B_1 \cup K$  then
3:     return operand1
4:   else if operand1.Typ = numbers then
5:     return  $\beta_{\text{Zahlen}}$ (operand1)
6:   else if operand1.Typ = string then
7:     return  $\beta_{\text{Zeichenketten}}$ (operand1)
8:   else if operand1.Typ = boolean then
9:     return  $\beta_{\text{Boolean}}$ (operand1)
10:  end if
11: else if ((operation = „+“)  $\vee$  (operation = „-“)  $\vee$  (operation = „*“)  $\vee$ 
           (operation = „div“)  $\vee$  (operation = „mod“)) then
12:  return analyseNumerisch(operation, operand1, operand2)
13: else if operation = „or“ then
14:  return boolean(operand1) or boolean(operand2)
15: else if operation = „and“ then
16:  return boolean(operand1) and boolean(operand2)
17: else if ((operation = „<“)  $\vee$  (operation = „>“)  $\vee$  (operation = „=“)  $\vee$ 
           (operation = „!=“)  $\vee$  (operation = „<=“)  $\vee$  (operation = „>=“)) then
18:  return analyseKonvertVergleich(operation, operand1, operand2)
19: else if operation = „$“ then
20:  return operand1
21: else if operation = „/“ then
22:  OUT  $\leftarrow \emptyset$ 
23:  for all k  $\in$  operand1 do
24:    OUT  $\leftarrow$  OUT  $\cup$  analyseAchse(k, operand2)
25:  end for
26:  return OUT
27: else if operation = „::“ then
28:  return analyseKnotentest(operand1, operand2)
29: else if operation = „[ ]“ then
30:  return operand1
31: else if operation = „|“ then
32:  return operand1  $\cup$  operand2
33: else if operation = „call“ then
34:  return analyseFkt(operand1, zähler, Ausdruck, Werte)
35: end if

```

---

## Beschreibung

Der Algorithmus prüft zunächst, welche Operation übergeben wurde, und führt sie danach aus. In der ersten Zeile wird die Operation „*id*“ erkannt. Diese Operation ist eine unäre Operation und wurde bei der Erstellung des Drei-Adress-Codes eingefügt. Die Operation „*id*“ identifiziert ein Datum. Dieses Datum ist als (erster) Operand angegeben und ist entweder eine Variable oder ein konstanter Wert. In Zeile 2 überprüfen wir, ob das Datum eine Variable ist. In diesem Fall, liegt das Datum bereits als Element der vollständigen Verbände aus den Definitionen 1, 7 und 18 oder als Knotenmenge, wie in Definition 21 definiert, vor. Demnach können wir in Zeile 3 das Datum unverändert zurück geben. Ist das Datum eine Konstante, werden die Zeilen 4 – 9 ausgeführt. Hier überprüfen wir, ob die Konstante eine Zahl, eine Zeichenkette oder ein boolescher Wert ist. Eine Knotenmenge kann in BPEL-Prozessen nicht als Konstante angegeben werden. Deshalb können wir hier eine Knotenmenge ausschließen. Je nach Typ der Konstanten überführen wir die Konstante mit Hilfe der entsprechenden Repräsentationsfunktion in ein Element der vollständigen Verbände aus den Definitionen 1, 7 und 18 und geben das Ergebnis zurück.

Die numerischen Operationen „+“, „-“, „\*“, „*div*“ und „*mod*“ aus XPath erkennen wir in Zeile 11. Für die Ausführung dieser Operationen rufen wir in Zeile 12 den Algorithmus `analyseNumerisch()` auf und geben das Ergebnis zurück. Die logischen Operationen „*or*“ und „*and*“ aus XPath erkennen wir in den Zeilen 13 und 15. Für die Ausführung dieser Operationen verwenden wir die Definitionen 2 und 3 aus dem formalen Datenmodell. Hier werden Funktionen definiert, die eine Disjunktion bzw. Konjunktion zweier boolescher Werte auswerten. Den Funktionswert geben wir als Ergebnis zurück.

Die Vergleichsoperationen „<“, „>“, „=“, „! =“, „<=“ und „>=“ aus XPath erkennen wir in Zeile 17. Bei der Ausführung dieser Operationen muss das Konvertierungsverhalten beachtet werden. In Zeile 18 rufen wir deshalb den Algorithmus `analyseKonvertVergleich()` auf und geben das Ergebnis zurück.

Die Operation „\$“ beschreibt einen Variablenzugriff in einem BPEL-Prozess. Ein solcher Variablenzugriff wird in Zeile 19 erkannt. Die Operation „\$“ ist eine unärer Operation. Da der Algorithmus `analyseOperation()` vom Algorithmus `analyseXPath()` aufgerufen wird, wurde ein Verweis auf eine Variable bereits aufgelöst. Deshalb können wir in Zeile 20 den übergebenen Operanden unverändert zurück geben.

Die Operation „/“ beschreibt den Anfang eines Lokalisierungsschrittes in XPath und wird in Zeile 21 erkannt. Der erste Operand beschreibt die Menge der Kontextknoten für den Lokalisierungsschritt. Der zweite Operand enthält die Achse des Lokalisierungsschrittes. Da wir jeden Knoten aus dem ersten Operanden als Kontextknoten betrachten müssen, initialisieren wir in Zeile 22 die Menge *OUT* mit der leeren Menge. In den Zeilen 23 – 25 betrachten wir nun jeden einzelnen Knoten aus dem ersten Operanden als Kontextknoten und wenden auf diesen Knoten die Achse aus dem zweiten Operanden an. Dafür rufen wir in Zeile 24 den Algorithmus `analyseAchse()` auf und speichern das Ergebnis in der Menge *OUT*. Wurde jeder Knoten aus dem ersten Operanden als Kon-

textknoten betrachtet, ist die Menge *OUT* vollständig berechnet und wir geben sie in Zeile 26 zurück.

Die Operation „:“ beschreibt den mittleren Teil eines Lokalisierungsschrittes in XPath und wird in Zeile 27 erkannt. Der erste Operand gibt eine Knotenmenge an, die mit Hilfe einer Achse ermittelt wurde. Der zweite Operand gibt den Knotennamen bzw. -typ für den Knotentest an. In Zeile 28 rufen wir die den Algorithmus `analyseKnotentest()` auf und geben das Ergebnis zurück.

Die Operation „[“ beschreibt das Ende eines Lokalisierungsschrittes und wird in Zeile 29 erkannt. Der erste Operand gibt die Knotenmenge an, die durch den Knotentest ermittelt wurde. Der zweite Operand gibt das Prädikat an. Das Prädikat filtert die Knotenmenge aus dem ersten Operand. Ein Prädikat ist ein XPath-Ausdruck, den wir mit Hilfe des Algorithmus `analyseXPath()` analysieren können. Für die Filterung der Knotenmenge müssen wir das Ergebnis der Analyse allerdings interpretieren. Beispielsweise kann das Prädikat dem Ausdruck `position() = 5` entsprechen. Da wir die Position eines Knotens statisch nicht bestimmen können, können wir hier nur interpretieren, dass die Knotenmenge nach der Filterung maximal ein Element enthält. Für den Ausdruck `position() > 5` können wir keine sichere Interpretation mehr liefern. Da wir nicht in jedem Fall eine Interpretation der Filterung liefern können, vernachlässigen wir in unserem Algorithmus die Filterung einer Knotenmenge durch Prädikate und liefern die Knotenmenge in Zeile 30 ohne Filterung zurück.

Die Operation „|“ beschreibt die Vereinigung von Knotenmengen und wird in Zeile 31 erkannt. Die beiden Operanden geben die Knotenmengen an, die vereinigt werden sollen. In Zeile 32 vereinigen wir die Knotenmengen und geben das Ergebnis zurück.

Die Operation „call“ beschreibt den Aufruf einer Funktion und wird in Zeile 33 erkannt. Der erste Operand gibt die Funktion an. Der zweite Operand gibt die Anzahl der Parameter an. Für den Aufruf der Funktion rufen wir in Zeile 34 den Algorithmus `analyseFkt()` auf und geben das Ergebnis zurück.

### Terminierung und Korrektheit

Der Algorithmus 8 berechnet das Ergebnis, das bei der Ausführung einer Operation entsteht. Eine solche Operation ist eine XPath-Operation, die Zugriffsoperation auf eine BPEL-Variable oder eine Operation, die bei der Erstellung des Drei-Adress-Codes eingefügt wurde.

Der Algorithmus terminiert immer, da er sequentiell ist. Der Algorithmus ist korrekt, da alle Operationen, die vorkommen dürfen, berücksichtigt werden.

#### 4.2.3. `analyseNumerisch()`

Der Algorithmus `analyseNumerisch()` führt eine numerische Operation aus XPath aus. Dabei beachten wir, dass die Operanden zunächst in Zahlen konvertiert werden, wenn

eine numerische Operation in XPath ausgeführt wird. Der Algorithmus wird vom Algorithmus `analyseOperation()` aufgerufen.

Der Algorithmus `analyseNumerisch()` ist in Algorithmus 9 definiert. Beim Aufruf werden dem Algorithmus drei Parameter übergeben. Der erste Parameter enthält die Operation, die ausgeführt werden soll. Der zweite und dritte Parameter enthält den ersten und zweiten Operanden.

---

**Algorithmus 9** `analyseNumerisch(operation, operand1, operand2)`

---

```
1: if operation = „+“ then
2:   return number(operand1)  $\oplus$  number(operand2)
3: else if operation = „-“ then
4:   return number(operand1)  $\ominus$  number(operand2)
5: else if operation = „*“ then
6:   return number(operand1)  $\otimes$  number(operand2)
7: else if operation = „div“ then
8:   return number(operand1)  $\oslash$  number(operand2)
9: else
10:  return number(operand1)  $\circ_{mod}$  number(operand2)
11: end if
```

---

## Beschreibung

Der Algorithmus führt eine numerische Operation mit Hilfe der Funktionen aus den Definitionen 8, 9, 10, 11 und 12 aus. Je nach Operation, die ausgeführt werden soll, wählt der Algorithmus eine dieser Funktionen aus. Der Funktionswert wird als Ergebnis zurückgegeben.

## Terminierung und Korrektheit

Der Algorithmus 9 führt eine der numerischen Operationen aus den Definitionen 8, 9, 10, 11 und 12 aus und ist somit korrekt. Er terminiert immer, da er sequentiell ist.

### 4.2.4. `analyseKonvertVergleich()`

Der Algorithmus `analyseKonvertVergleich()` übernimmt die Konvertierung bei der Ausführung einer Vergleichsoperation in XPath, wenn Knotenmengen am Vergleich beteiligt sind. Der Algorithmus konvertiert die Knotenmengen gemäß den Vorgaben von XPath. Sind beide Operanden keine Knotenmenge, ist eine Konvertierung unnötig und die Vergleichsoperation wird sofort ausgeführt. Der Algorithmus wird vom Algorithmus `analyseOperation()` aufgerufen.

Der Algorithmus `analyseKonvertVergleich()` ist in Algorithmus 10 definiert. Beim Aufruf werden dem Algorithmus drei Parameter übergeben. Der erste Parameter enthält die Vergleichsoperation, die ausgeführt werden soll. Der zweite und dritte Parameter enthält den ersten und zweiten Operanden.

### Beschreibung

Sollen zwei Knotenmengen miteinander verglichen werden, werden die Zeilen 3 – 13 ausgeführt. Wie bereits in Kapitel 2.3.4 erwähnt, werden zwei Knotenmenge verglichen, indem jeder Knoten der einen Knotenmenge mit jedem Element der anderen Knotenmenge verglichen wird. Dafür definieren wir in den Zeilen 3 und 4 zwei Schleifen. Der Vergleich zwischen zwei Knoten wird dann in Zeile 5 durch den Aufruf des Algorithmus `analyseVergleich()` ausgeführt. Zwei Knoten werden miteinander verglichen, indem deren Zeichenkettenwerte verglichen werden. Deshalb werden dem Algorithmus `analyseVergleich()` die beiden Operanden nach Konvertierung mit Hilfe der Funktion `string()` übergeben. Das Ergebnis des Algorithmus `analyseVergleich()` wird in der Variablen `Tmp` gespeichert. In Zeile 6 überprüfen wir nun, ob der Vergleich der beiden Knoten eindeutig den Wert `true` ergibt. Ist dies der Fall, geben wir in Zeile 7 den Wert `true` zurück. Denn ein Vergleich, bei dem mindestens eine Knotenmenge beteiligt ist, ist wahr, sobald der Vergleich für einen Knoten wahr ist. Ergibt der Vergleich der beiden Knoten potentiell den Wert `false`, speichern wir in Zeile 9 das Ergebnis des Vergleichs in der Menge `W`. Danach vergleichen die nächsten beiden Knoten. Wurden alle Knoten miteinander verglichen und kein Vergleich lieferte eindeutig den Wert `true`, erreicht der Algorithmus Zeile 13. Hier wird die vollständig berechnete Menge `W` zurückgegeben und der Algorithmus terminiert.

Ist nur einer der beiden Operanden eine Knotenmenge, werden die Zeilen 15 – 41 ausgeführt. In den Zeilen 15 – 21 werden die Variablen `op1` und `op2` initialisiert. In `op1` wird die Knotenmenge gespeichert. In `op2` wird der Operand gespeichert, der keine Knotenmenge ist. Um jedes Element in der Knotenmenge mit `op2` zu vergleichen, definieren wir in Zeile 22 eine Schleife. Bevor wir einen Knoten aus der Knotenmenge mit `op2` vergleichen können, müssen wir den Knoten konvertieren. Wie bereits in Kapitel 2.3.4 beschrieben, ist dabei die Konvertierung des Knotens vom Typ des Operanden in `op2` abhängig. Die Zeilen 23 – 29 beschreiben die Konvertierung, wie sie in XPath vorgenommen wird. Ist der Operand in `op2` eine Zahl, wird in Zeile 24 der Zeichenkettenwert des Knotens in eine Zahl konvertiert. Ist der Operand in `op2` eine Zeichenkette, wird in Zeile 26 der Zeichenkettenwert des Knotens ermittelt. Sonst wird in Zeile 28 der Zeichenkettenwert des Knotens in einen booleschen Wert konvertiert. Das Ergebnis der Konvertierung wird in der Variablen `xneu` gespeichert.

Der Vergleich zwischen dem konvertierten Knoten und dem Operanden in `op2` wird in den Zeilen 30 – 34 ausgeführt. Ist der erste Operand ein Knoten, wird der Algorithmus `analyseVergleich()` in Zeile 31 aufgerufen. Ist der zweite Operand ein Knoten, wird der Algorithmus `analyseVergleich` in Zeile 33 aufgerufen. Das Ergebnis des Algorithmus ana-

---

**Algorithmus 10** analyseKonvertVergleich(operation, operand1, operand2)

---

```
1:  $W \leftarrow \emptyset$ 
2: if operand1.Typ = Knotenmenge  $\wedge$  operand2.Typ = Knotenmenge then
3:   for all  $x \in$  operand1 do
4:     for all  $y \in$  operand2 do
5:       Tmp  $\leftarrow$  analyseVergleich(operation, string(x), string(y))
6:       if Tmp = {true} then
7:         return Tmp
8:       else
9:          $W \leftarrow W \cup$  Tmp
10:      end if
11:    end for
12:  end for
13:  return W
14: else if operand1.Typ = Knotenmenge  $\vee$  operand2.Typ = Knotenmenge then
15:   if operand1.Typ = Knotenmenge then
16:     op1  $\leftarrow$  operand1
17:     op2  $\leftarrow$  operand2
18:   else
19:     op1  $\leftarrow$  operand2
20:     op2  $\leftarrow$  operand1
21:   end if
22:   for all  $x \in$  op1 do
23:     if op2.Typ = numbers then
24:        $x_{\text{neu}} \leftarrow$  number(string(x))
25:     else if op2.Typ = string then
26:        $x_{\text{neu}} \leftarrow$  string(x)
27:     else {op2.Typ = boolean}
28:        $x_{\text{neu}} \leftarrow$  boolean(string(x))
29:     end if
30:     if operand1.Typ = Knotenmenge then
31:       Tmp  $\leftarrow$  analyseVergleich(operation,  $x_{\text{neu}}$ , operand2)
32:     else
33:       Tmp  $\leftarrow$  analyseVergleich(operation, operand1,  $x_{\text{neu}}$ )
34:     end if
35:     if Tmp = {true} then
36:       return Tmp
37:     else
38:        $W \leftarrow W \cup$  Tmp
39:     end if
40:   end for
41:   return W
42: else {am Vergleich ist keine Knotenmenge beteiligt}
43:   return analyseVergleich(operation, operand1, operand2)
44: end if
```

---

lyseVergleich() wird in der Variablen Tmp gespeichert. In den Zeilen 35 – 39 überprüfen wir wieder, ob der Vergleich eindeutig den Wert *true* ergibt. Ist dies der Fall, geben wir in Zeile 36 den Wert *true* zurück. Sonst speichern wir in Zeile 38 das Ergebnis des Vergleichs in der Menge *W* und betrachten den nächsten Knoten. Wurden alle Knoten in der Knotenmenge betrachtet und kein Vergleich lieferte eindeutig den Wert *true*, erreicht der Algorithmus Zeile 41. Hier wird die vollständig berechnete Menge *W* zurückgegeben und der Algorithmus terminiert.

Ist keine Knotenmenge am Vergleich beteiligt, werden die Operanden nicht konvertiert. In diesem Fall rufen wir in Zeile 43 den Algorithmus analyseVergleich() mit den beiden (unveränderten) Operanden auf. Das Ergebnis des Vergleichs wird zurückgegeben und der Algorithmus terminiert.

### Terminierung und Korrektheit

Der Algorithmus 10 konvertiert zwei Operanden für die Ausführung einer Vergleichsoperation in XPath. Die Konvertierung der Operanden ist nur nötig, wenn eine Knotenmenge am Vergleich beteiligt ist. In diesem Fall wird jeder Knoten in der Knotenmenge konvertiert und verglichen. Sind beide Operanden keine Knotenmenge, wird die Vergleichsoperation ohne Konvertierung ausgeführt.

Der Algorithmus terminiert, weil eine Knotenmenge immer endlich ist. Der Algorithmus ist korrekt, da das Konvertierungsverhalten bei Vergleichen mit Knotenmengen in XPath korrekt ausgeführt wird. Das Konvertierungsverhalten bei Vergleichen in XPath haben wir bereits in Kapitel 2.3.4 beschrieben.

#### 4.2.5. analyseVergleich()

Der Algorithmus analyseVergleich() führt eine Vergleichsoperation aus XPath aus. Dabei beachten wir, dass die Operanden zunächst in einen gemeinsamen Typ konvertiert werden, wenn eine Vergleichsoperation in XPath ausgeführt wird. Der Algorithmus wird vom Algorithmus analyseKonvertVergleich() aufgerufen. Deshalb können Knotenmengen als Operanden ausgeschlossen werden.

Der Algorithmus analyseVergleich() ist in Algorithmus 11 definiert. Beim Aufruf werden dem Algorithmus drei Parameter übergeben. Der erste Parameter enthält die Vergleichsoperation, die ausgeführt werden soll. Der zweite und dritte Parameter enthält den ersten und zweiten Operanden.

**Algorithmus 11** analyseVergleich(operation, operand1, operand2)

---

```
1: if operation = „<“ then
2:   return number(operand1) <Int number(operand2)
3: else if operation = „>“ then
4:   return number(operand1) >Int number(operand2)
5: else if operation = „<=“ then
6:   kleiner ← number(operand1) <Int number(operand2)
7:   gleich ← number(operand1) =Int number(operand2)
8:   return kleiner or gleich
9: else if operation = „>=“ then
10:  groesser ← number(operand1) >Int number(operand2)
11:  gleich ← number(operand1) =Int number(operand2)
12:  return groesser or gleich
13: else if operation = „=“ then
14:   if operand1.Typ = boolean ∨ operand2.Typ = boolean then
15:     boolean(operand1) =Bool boolean(operand2)
16:   else if operand1.Typ = numbers ∨ operand2.Typ = numbers then
17:     number(operand1) =Int number(operand2)
18:   else
19:     string(operand1) =Z string(operand2)
20:   end if
21: else {operation = „!=“}
22:   if operand1.Typ = boolean ∨ operand2.Typ = boolean then
23:     boolean(operand1) !=Bool boolean(operand2)
24:   else if operand1.Typ = numbers ∨ operand2.Typ = numbers then
25:     number(operand1) !=Int number(operand2)
26:   else
27:     string(operand1) !=Z string(operand2)
28:   end if
29: end if
```

---

**Beschreibung**

Der Algorithmus führt die Vergleichsoperationen „<“, „>“, „<=“ und „>=“ werden in den Zeilen 1 – 12 ausgeführt. Diese Vergleichsoperationen sind nur für Zahlen definiert. Deshalb werden die Operanden in Zahlen konvertiert und danach mit Hilfe der Funktionen aus den Definitionen 2, 13, 14, 15 und 16 verglichen.

Die Vergleichsoperationen „=“ und „!=“ sind sowohl für Zahlen als auch für boolesche Werte und Zeichenketten definiert. Vor ihrer Ausführung müssen die Operanden in einen gemeinsamen Typ konvertiert werden. Die Konvertierung ist dabei von den Typen der beteiligten Operanden abhängig. Ist einer der Operanden eine Zahl, werden beide Operanden in eine Zahl konvertiert. Sonst wird überprüft, ob einer der Operanden einen

booleschen Wert repräsentiert. Ist dies der Fall, werden beide Operanden in boolesche Werte konvertiert. In jedem anderen Fall werden die Operanden in Zeichenketten konvertiert. Der Vergleich mit der Operation „=“ wird dann in den Zeilen 14 – 20 mit Hilfe der Funktionen aus den Definitionen 4, 15 und 19 ausgeführt. Der Vergleich mit der Operation „!=“ wird in den Zeilen 22 – 28 mit Hilfe der Funktionen aus den Definitionen 5, 16 und 20 ausgeführt.

### Terminierung und Korrektheit

Der Algorithmus 11 führt eine Vergleichsoperation aus XPath aus. Dabei werden nur Zahlenwerte, boolesche Werte oder Zeichenketten miteinander verglichen. Der Algorithmus terminiert immer, da er sequentiell ist. Er ist korrekt, da das Konvertierungsverhalten bei Vergleichsoperationen in XPath beachtet wird und die verwendeten Definitionen zur Ausführung der Vergleichsoperationen aus dem formalen Modell korrekt sind. Die Operationen „<“, „<=“, „>“ und „>=“ konvertieren ihre Operanden immer in Zahlen. Bei den Operationen „<=“ und „>=“ werden die Operanden in boolesche Werte, Zahlen oder Zeichenketten konvertiert.

#### 4.2.6. analyseAchse()

Der Algorithmus `analyseAchse()` ermittelt eine Menge von Knoten, die über eine Achse erreicht werden können. Diese Achse ist Teil eines Lokalisierungsschrittes aus XPath. Ausgehend von einem gegebenen Kontextknoten werden die Knoten ermittelt, die über die Achse erreicht werden können. Der Algorithmus wird vom Algorithmus `analyseOperation()` aufgerufen.

Der Algorithmus `analyseAchse()` ist in Algorithmus 12 definiert. Beim Aufruf werden dem Algorithmus zwei Parameter übergeben. Der erste Parameter enthält den Kontextknoten. Der zweite Parameter enthält die Achse, anhand derer eine Knotenmenge ermittelt werden soll.

Der Kontextknoten ist ein Tupel  $(x,y)$ , wie in Kapitel 3.1.4 beschrieben.  $x$  beschreibt einen ausgezeichneten Knoten im Baum  $y$ .  $x$  repräsentiert dabei ein XML-Element bzw. Attribut.  $y$  repräsentiert ein XML-Schema. Um eine Achse auf den Kontextknoten anzuwenden, suchen wir alle Knoten  $z$  im Baum  $y$ , die von  $x$  aus erreichbar sind. Das Ergebnis des Algorithmus ist eine Menge von Tupeln  $(z,y)$ .

### Beschreibung

Die Achsen `ancestor` und `ancestor-or-self` werden in der ersten Zeile erkannt. Um die Vorfahren des Kontextknotens zu ermitteln, rufen wir in Zeile 2 den Algorithmus `analyseAncestor()` auf. In den Zeilen 3 – 5 erweitern wir die Menge der Vorfahren um den Kontextknoten, wenn wir die Achse `ancestor-or-self` betrachten.

---

**Algorithmus 12** analyseAchse(knoten, achse)

---

```

1: if achse = ancestor  $\vee$  achse = ancestor-or-self then
2:   OUT  $\leftarrow$  analyseAncestor(knoten)
3:   if achse = ancestor-or-self then
4:     OUT  $\leftarrow$  OUT  $\cup$  {knoten}
5:   end if
6: else if achse = child then
7:   OUT  $\leftarrow$  analyseChild(knoten)
8: else if achse = descendant  $\vee$  achse = descendant-or-self then
9:   OUT  $\leftarrow$  analyseDescendant(knoten)
10:  if achse = descendant-or-self then
11:    OUT  $\leftarrow$  OUT  $\cup$  {knoten}
12:  end if
13: else if achse = following then
14:   OUT  $\leftarrow$  analyseFollowing(knoten)
15: else if achse = preceding then
16:   OUT  $\leftarrow$  analysePreceding(knoten)
17: else if achse = preceding-sibling  $\vee$  achse = following-sibling then
18:   (x, baum)  $\leftarrow$  knoten
19:   hole (x,y,z) aus  $Typ_{baum}$ 
20:   if y = A  $\vee$  {x.vorgänger} =  $\emptyset$  then
21:     OUT  $\leftarrow$   $\emptyset$ 
22:   else
23:     OUT  $\leftarrow$  analyseChild((x.vorgänger, baum))
24:     OUT  $\leftarrow$  OUT  $\setminus$  {knoten}
25:   end if
26: else if achse = parent then
27:   (x, baum)  $\leftarrow$  knoten
28:   if {x.vorgänger} =  $\emptyset$  then
29:     OUT  $\leftarrow$   $\emptyset$ 
30:   else
31:     OUT  $\leftarrow$  {(x.vorgänger, baum)}
32:   end if
33: else if achse = self then
34:   OUT  $\leftarrow$  {knoten}
35: else if achse = attribute then
36:   OUT  $\leftarrow$  analyseAttributNamespace(knoten, 'A')
37: else if achse = namespace then
38:   OUT  $\leftarrow$  analyseAttributNamespace(knoten, 'N')
39: end if
40: return OUT

```

---

Die Achse `child` wird in Zeile 6 erkannt. Um die Kinder des Kontextknotens zu ermitteln, rufen wir in Zeile 7 den Algorithmus `analyseChild()` auf.

Die Achsen `descendant` und `descendant-or-self` werden in Zeile 8 erkannt. Um die Nachkommen des Kontextknotens zu ermitteln, rufen wir in Zeile 9 den Algorithmus `analyseDescendant()` auf. In den Zeilen 10 – 12 erweitern wir die Menge der Nachkommen um den Kontextknoten, wenn wir die Achse `descendant-or-self` betrachten.

Die Achse `following` wird in Zeile 13 erkannt. Die Achse ermittelt alle Knoten, die im Instanzdokument nach dem Kontextknoten auftreten. Da wir das konkrete Instanzdokument nicht kennen, ermitteln wir eine Menge von Knoten, die potentiell nach dem Kontextknoten im Instanzdokument auftreten können. Damit sind in der ermittelten Menge eventuell Knoten enthalten, die nicht in der Ergebnismenge zur Laufzeit enthalten sind. Allerdings ist jeder Knoten, der zur Laufzeit in der Ergebnismenge enthalten ist, in unserer ermittelten Menge enthalten. Um die Menge der Knoten zu ermitteln, die potentiell nach dem Kontextknoten im Instanzdokument auftreten können, rufen wir in Zeile 14 den Algorithmus `analyseFollowing()` auf.

Die Achse `preceding` wird in Zeile 15 erkannt. Die Achse ermittelt alle Knoten, die im Instanzdokument vor dem Kontextknoten auftreten. Analog zur `following`-Achse ermitteln wir die Menge der Knoten, die potentiell vor dem Kontextknoten im Instanzdokument auftreten. Um diese Menge zu ermitteln, rufen wir in Zeile 16 den Algorithmus `analysePreceding()` auf.

Die Achsen `preceding-sibling` und `following-sibling` werden in Zeile 17 erkannt. Diese Achsen ermitteln die Geschwister des Kontextknotens, die vor bzw. nach dem Kontextknoten im Instanzdokument auftreten. Da wir das Instanzdokument nicht kennen, ermitteln wir die Geschwister, die potentiell vor bzw. nach dem Kontextknoten im Instanzdokument auftreten. Da wir nicht entscheiden können, ob Geschwisterknoten vor oder nach dem Kontextknoten auftreten, ist die Ergebnismenge dieser beiden Achsen gleich. Sie enthält jeweils die Menge aller Geschwister. Zunächst überprüfen wir in Zeile 20, ob der Kontextknoten ein Attribut repräsentiert oder ob der Kontextknoten ein Wurzelknoten ist. Ist dies der Fall, hat der Kontextknoten keine Geschwister und wir speichern in Zeile 21 die leere Menge als Ergebnismenge. Sonst ermitteln wir in Zeile 23 den Elterknoten des Kontextknotens und dann die Kinder des Elterknotens. Die so ermittelte Menge enthält alle Geschwister des Kontextknotens und den Kontextknoten selbst. Deshalb entfernen wir in Zeile 24 den Kontextknoten aus der ermittelten Menge.

Die Achse `parent` wird in Zeile 26 erkannt. Diese Achse ermittelt den Elterknoten des Kontextknotens. In Zeile 28 überprüfen wir zunächst, ob der Kontextknoten ein Wurzelknoten ist. In diesem Fall hat der Kontextknoten keinen Elterknoten und wir speichern die leere Menge als Ergebnismenge. Sonst speichern wir in Zeile 31 die Menge als Ergebnismenge, die nur den Elterknoten des Kontextknotens enthält.

Die Achse `self` wird in Zeile 33 erkannt. Diese Achse ermittelt eine Menge, die nur den Kontextknoten selbst enthält. Deshalb speichern wir in Zeile 34 die Menge als Ergebnismenge, die nur den Kontextknoten enthält.

Die speziellen Achsen `attribute` und `namespace` werden in den Zeilen 35 und 37 erkannt. Um die Attribut- bzw. Namensraumknoten des Kontextknotens zu ermitteln, rufen wir in Zeile 36 bzw. 38 den Algorithmus `analyseAttributNamespace()` auf. Als zweiten Parameter übergeben wir dem Algorithmus `analyseAttributNamespace()` ein 'A' wenn die Attributknoten ermittelt werden sollen. Sollen die Namensraumknoten ermittelt werden, übergeben wir dem Algorithmus `analyseAttributNamespace()` als zweiten Parameter ein 'N'.

### Terminierung und Korrektheit

Der Algorithmus 12 ermittelt eine Knotenmenge, die ausgehend von einem Kontextknoten über die Achse `attribute` oder `namespace` erreicht werden kann. Der Algorithmus terminiert immer, da er sequentiell ist. Er ist korrekt, da nur die Knoten in die Ergebnismenge aufgenommen werden, die über die Achsen erreichbar sind. Die Semantik der Achsen wurden bereits in Kapitel 2.3.5 vorgestellt.

#### 4.2.7. `analyseAncestor()`

Der Algorithmus `analyseAncestor()` ermittelt die Vorfahren eines Knotens. Die Vorfahren eines Knotens bestehen aus dem Elterknoten, dessen Elterknoten usw.. Der Algorithmus wird vom Algorithmus `analyseAchse()` aufgerufen.

Der Algorithmus ist in Algorithmus 13 definiert. Beim Aufruf wird dem Algorithmus ein Parameter übergeben. Dieser Parameter enthält den Kontextknoten, dessen Vorfahren ermittelt werden soll. Der Knoten ist als Tupel (x,y) gegeben, wie in Kapitel 3.1.4 beschrieben.

---

**Algorithmus 13** `analyseAncestor((knoten, baum))`

---

```
1: OUT ← ∅
2: w ← knoten
3: while {w.vorgänger} ≠ ∅ do
4:   OUT ← OUT ∪ {(w.vorgänger, baum)}
5:   w ← w.vorgänger
6: end while
7: return OUT
```

---

## Beschreibung

In Zeile 2 speichern wir den Kontextknoten als aktuellen Knoten. In den Zeilen 3 – 6 definieren wir eine Schleife. In dieser Schleife ermitteln wir den Elterknoten des aktuellen Knotens und speichern diesen in der Ergebnismenge. Danach wird der ermittelte Elterknoten als aktueller Knoten gespeichert und die Schleife wiederholt sich. Wir wiederholen die Schleife solange, bis der aktuelle Knoten einen Wurzelknoten repräsentiert. Danach ist die Menge der Vorfahren vollständig berechnet und wir geben sie in Zeile 7 zurück.

## Terminierung und Korrektheit

Der Algorithmus 13 ermittelt die Vorfahren eines Knotens. Der Algorithmus terminiert, da der betrachtete Baum endlich viele Knoten hat und die Ermittlung der Voränger mit Erreichen des Wurzelknotens endet. Der Algorithmus ist korrekt, da alle Elterknoten sowie deren Elterknoten, usw. ermittelt werden und Attribut- und Namensraumknoten aus der ermittelten Menge ausgeschlossen werden.

### 4.2.8. analyseChild()

Der Algorithmus analyseChild() ermittelt die Kinder eines Knotens. Die ermittelten Kinder repräsentieren niemals Attribut- oder Namensraumknoten. Der Algorithmus wird vom Algorithmus analyseAchse() aufgerufen.

Der Algorithmus ist in Algorithmus 14 definiert. Beim Aufruf wird dem Algorithmus ein Parameter übergeben. Dieser Parameter enthält den Kontextknoten, dessen Kinder ermittelt werden sollen. Der Knoten ist als Tupel (x,y) gegeben, wie in Kapitel 3.1.4 beschrieben.

---

**Algorithmus 14** analyseChild((knoten, baum))

---

```
1: OUT  $\leftarrow$   $\emptyset$ 
2: Nachfolger  $\leftarrow$  nachfolger(knoten)
3: for all x  $\in$  Nachfolger do
4:   OUT  $\leftarrow$  OUT  $\cup$  {(x, baum)}
5: end for
6: return analyseKnotentyp(OUT)
```

---

## Beschreibung

Wir ermitteln die Kinder des Kontextknotens, indem wir in Zeile 2 zunächst die Menge aller Nachfolger des Kontextknotens bestimmen. Die Schleife in den Zeilen 3 – 5 fügt diese Nachfolger in die Ergebnismenge ein. Um die Attribut- und Namensraumknoten in der

Ergebnismenge zu entfernen, rufen wir in Zeile 6 den Algorithmus `analyseKnotentyp()` auf, bevor wir das Ergebnis zurück geben.

### Terminierung und Korrektheit

Der Algorithmus 14 ermittelt die Kinder eines Knotens. Der Algorithmus terminiert, da ein Knoten nur endlich viele Kinder hat. Der Algorithmus ist korrekt, da alle Kinder betrachtet werden und Attribut- und Namensraumknoten aus der ermittelten Menge ausgeschlossen werden.

#### 4.2.9. `analyseDescendant()`

Der Algorithmus `analyseDescendant()` ermittelt die Nachkommen eines Knotens. Die Nachkommen eines Knotens bestehen aus den Kindern, dessen Kindern usw.. Die ermittelten Nachkommen repräsentieren niemals Attribut- oder Namensraumknoten. Der Algorithmus wird vom Algorithmus `analyseAchse()` aufgerufen.

Der Algorithmus ist in Algorithmus 15 definiert. Beim Aufruf wird dem Algorithmus ein Parameter übergeben. Dieser Parameter enthält den Kontextknoten, dessen Nachkommen ermittelt werden soll. Der Knoten ist als Tupel  $(x,y)$  gegeben, wie in Kapitel 3.1.4 beschrieben.

---

#### Algorithmus 15 `analyseDescendant((knoten, baum))`

---

```
1: OUT  $\leftarrow \emptyset$ 
2: Besuchen = {knoten}
3: while Besuchen  $\neq \emptyset$  do
4:   w  $\leftarrow$  hole Knoten aus Besuchen
5:   Besuchen  $\leftarrow$  Besuchen  $\setminus$  {w}
6:   Nachfolger  $\leftarrow$  nachfolger(w)
7:   for all x  $\in$  Nachfolger do
8:     OUT  $\leftarrow$  OUT  $\cup$  {(x, baum)}
9:   end for
10:  Besuchen  $\leftarrow$  Besuchen  $\cup$  Nachfolger
11: end while
12: return analyseKnotentyp(OUT)
```

---

### Beschreibung

Zu Anfang müssen wir die Kinder des Kontextknotens ermitteln. In Zeile 2 fügen wir deshalb den Kontextknoten in die Menge *Besuchen* ein. Diese Menge enthält alle Knoten, dessen Kinder noch ermittelt werden müssen. Die Schleife in Zeile 3 ermöglicht eine Wiederholung bis alle Kinder und Kindesinder ermittelt wurden. In den Zeilen 4 und 5

entfernen wir einen Knoten aus der Menge *Besuchen* und speichern diesen als aktuellen Knoten. In Zeile 6 ermitteln wir nun die Kinder des aktuellen Knotens. Die Schleife in den Zeilen 7 – 9 fügt die ermittelten Kinder in die Ergebnismenge ein. Da wir nun noch die Kinder der Kinder ermitteln müssen, erweitern wir in Zeile 10 die Menge *Besuchen* um die Kinder des aktuellen Knotens. Danach wiederholen wir die Schleife aus Zeile 3 solange bis alle Kinder und Kindeskindern betrachtet wurden. Um die Attribut- und Namensraumknoten in der Ergebnismenge zu entfernen, rufen wir in Zeile 12 den Algorithmus `analyseKnotentyp()` auf, bevor wir das Ergebnis zurück geben.

### Terminierung und Korrektheit

Der Algorithmus 15 ermittelt die Nachkommen eines Knotens. Der Algorithmus terminiert, da der betrachtete Baum endlich ist und somit die Anzahl der Kinder eines Knotens, deren Kinder usw. endlich ist. Der Algorithmus ist korrekt, da alle Kinder, deren Kinder, usw. betrachtet werden und Attribut- und Namensraumknoten aus der ermittelten Menge ausgeschlossen werden.

#### 4.2.10. `analyseFollowing()`

Der Algorithmus `analyseFollowing()` ermittelt die Knoten, die im Instanzdokument nach einem bestimmten Kontextknoten auftreten können. Die ermittelte Knotenmenge enthält niemals die Kinder des Kontextknotens und keine Attribut- und Namensraumknoten. Der Algorithmus wird vom Algorithmus `analyseAchse()` aufgerufen.

Der Algorithmus ist in Algorithmus 16 definiert. Beim Aufruf wird dem Algorithmus ein Parameter übergeben. Dieser Parameter enthält den Kontextknoten. Der Kontextknoten ist als Tupel  $(x,y)$  gegeben, wie in Kapitel 3.1.4 beschrieben.

Da wir das konkrete Instanzdokument nicht kennen, ermitteln wir eine Menge von Knoten, die potentiell nach dem Kontextknoten im Instanzdokument auftreten können. Deshalb sind in der ermittelten Menge eventuell Knoten enthalten, die sich nicht in der Ergebnismenge zur Laufzeit befinden. Allerdings ist jeder Knoten, der zur Laufzeit in der Ergebnismenge befindet, in unserer ermittelten Menge enthalten. Somit ist die Menge, die zur Laufzeit erzeugt wird, eine Teilmenge unserer ermittelten Menge.

Die Geschwister des Kontextknotens gehören zu den Knoten, die potentiell nach dem Kontextknoten auftreten können. Weiterhin gehören die Nachfolger der Geschwister zu den Knoten, die potentiell nach dem Kontextknoten auftreten können.

### Beschreibung

Der Algorithmus arbeitet analog zum Algorithmus `analyseDescendant()`. In Zeile 2 speichern wir zunächst den Elterknoten des Kontextknotens in der Menge *Besuchen*. In der Schleife aus Zeile 3 ermitteln wir danach die Geschwister des Kontextknotens. In den

**Algorithmus 16** analyseFollowing((knoten, baum))

---

```
1: OUT  $\leftarrow \emptyset$ 
2: Besuchen = {knoten.vorgänger}
3: while Besuchen  $\neq \emptyset$  do
4:   hole Knoten w aus Besuchen
5:   Besuchen  $\leftarrow$  Besuchen  $\setminus$  {w}
6:   Nachfolger  $\leftarrow$  nachfolger(w)
7:   for all x  $\in$  Nachfolger do
8:     if x  $\neq$  knoten then
9:       OUT  $\leftarrow$  OUT  $\cup$  {(x, baum)}
10:    else
11:      Nachfolger  $\leftarrow$  Nachfolger  $\setminus$  {x}
12:    end if
13:  end for
14:  Besuchen  $\leftarrow$  Besuchen  $\cup$  Nachfolger
15: end while
16: return analyseKnotentyp(OUT)
```

---

Zeilen 7 – 13 nehmen wir die Geschwister des Kontextknotens in die Ergebnismenge auf. Der Kontextknoten wird nicht in die Ergebnismenge aufgenommen. Zusätzlich entfernen wir den Kontextknoten in Zeile 11 aus der Menge *Nachfolger*. Damit werden in den folgenden Schritten auch nicht die Kinder und Kindeskinde des Kontextknotens in der Ergebnismenge aufgenommen. Schließlich ermitteln wir die Nachfolger der Geschwister des Kontextknotens analog zum Algorithmus analyseDescendant().

**Terminierung und Korrektheit**

Der Algorithmus 16 ermittelt die Knoten im Instanzdokument, die nach einem bestimmten Kontextknoten auftreten. Der Algorithmus terminiert, da der betrachtete Baum endlich ist und somit die Anzahl der Knoten, die nach dem Kontextknoten auftreten können, endlich ist. Der Algorithmus ist korrekt, da alle Geschwister sowie deren Nachfolger betrachtet werden und die Nachfolger des Kontextknotens sowie Attribut- und Namensraumknoten aus der ermittelten Menge ausgeschlossen werden.

**4.2.11. analysePreceding()**

Der Algorithmus analysePreceding() ermittelt die Knoten, die im Instanzdokument vor einem bestimmten Kontextknoten auftreten können. Die ermittelte Menge von Knoten enthält niemals die Vorfahren des Kontextknotens und keine Attribut- und Namensraumknoten. Der Algorithmus wird vom Algorithmus analyseAchse() aufgerufen.

Der Algorithmus ist in Algorithmus 17 definiert. Beim Aufruf wird dem Algorithmus ein Parameter übergeben. Dieser Parameter enthält den Kontextknoten. Der Kontextknoten ist als Tupel  $(x,y)$  gegeben, wie in Kapitel 3.1.4 beschrieben.

Da wir das konkrete Instanzdokument nicht kennen, ermitteln wir eine Menge von Knoten, die potentiell vor dem Kontextknoten im Instanzdokument auftreten können. Wie beim Algorithmus `analyseFollowing()` ist die Menge, die zur Laufzeit erzeugt wird, eine Teilmenge unserer ermittelten Menge. Die ermittelte Menge enthält alle Geschwister des Kontextknotens. Weiterhin enthält die ermittelte Menge die Geschwister der Vorfahren des Kontextknotens und die Nachfolger aller ermittelter Geschwister.

---

**Algorithmus 17** `analysePreceding((knoten, baum))`

---

```
1: OUT  $\leftarrow \emptyset$ 
2: aktuell  $\leftarrow$  knoten
3: while {aktuell.vorgänger}  $\neq \emptyset$  do
4:   OUT  $\leftarrow$  OUT  $\cup$  analyseFollowing((aktuell, baum))
5:   aktuell  $\leftarrow$  aktuell.vorgänger
6: end while
7: return OUT
```

---

### Beschreibung

Zunächst speichern wir in Zeile 2 den Kontextknoten als aktuellen Knoten. Danach führen wir die Schleife in Zeile 3 aus. In Zeile 4 ermitteln wir die Geschwister des aktuellen Knotens sowie die Nachfolger der Geschwister und speichern sie in der Ergebnismenge. Danach ermitteln wir in Zeile 5 den Elterknoten des aktuellen Knotens und speichern ihn als aktuellen Knoten. Nun wiederholen wir die Schleife aus Zeile 3 bis der Elterknoten des aktuellen Knoten den Wurzelknoten repräsentiert. Damit ermitteln wir auch die Geschwister der Elterknoten und speichern diese in der Ergebnismenge. Ist die Ergebnismenge vollständig berechnet, entfernen wir in Zeile 7 die Attribut- und Namensraumknoten mit Hilfe des Algorithmus `analyseKnotentyp()` und geben das Ergebnis zurück.

### Terminierung und Korrektheit

Der Algorithmus 17 ermittelt die Knoten im Instanzdokument, die vor einem bestimmten Kontextknoten auftreten. Der Algorithmus terminiert, da der betrachtete Baum endlich ist und somit die Anzahl der Knoten, die nach dem Kontextknoten auftreten können, endlich ist. Der Algorithmus ist korrekt, da alle Geschwister sowie deren Nachfolger betrachtet werden und die Vorfahren des Kontextknotens sowie Attribut- und Namensraumknoten aber aus der ermittelten Menge ausgeschlossen werden. Diese Knoten werden aus der Menge ausgeschlossen, da der Algorithmus `analyseFollowing()` mit den Vorfahren als Parameter aufgerufen wird. Der Algorithmus `analyseFollowing()` liefert

gerade eine Menge, die den übergebenen Parameter sowie Attribut- und Namensraumknoten nicht enthält.

#### 4.2.12. analyseAttributNamespace()

Der Algorithmus analyseAttributNamespace() ermittelt die Attribut- oder Namensraumknoten eines Knotens. Der Algorithmus wird vom Algorithmus analyseAchse() aufgerufen.

Der Algorithmus ist in Algorithmus 18 definiert. Beim Aufruf werden dem Algorithmus zwei Parameter übergeben. Der erste Parameter enthält den Kontextknoten, dessen Attribut- oder Namensraumknoten ermittelt werden sollen. Der Kontextknoten ist als Tupel (x,y) gegeben, wie in Kapitel 3.1.4 beschrieben. Der zweite Parameter enthält den Knotentyp, der ermittelt werden soll. Enthält der zweite Parameter ein A, werden Attributknoten ermittelt. Enthält der zweite Parameter ein N, werden Namensraumknoten ermittelt.

---

**Algorithmus 18** analyseAttributNamespace((knoten, baum), Typ)

---

```
1: OUT  $\leftarrow \emptyset$ 
2: Nachfolger  $\leftarrow$  nachfolger(knoten)
3: for all x  $\in$  Nachfolger do
4:   if (x, Typ, y)  $\in$  Typbaum then
5:     OUT  $\leftarrow$  OUT  $\cup$  {(x, baum)}
6:   end if
7: end for
8: return OUT
```

---

#### Beschreibung

In Zeile 2 ermitteln wir zunächst alle Nachfolger des Kontextknotens. In den Zeilen 3 – 7 betrachten wir nun jeden Knoten, der in der Menge der Nachfolger enthalten ist. Ist ein solcher Knoten vom geforderten Typ, speichern wir ihn in Zeile 5 in der Ergebnismenge. Wurden alle Knoten in der Menge *Nachfolger* betrachtet, ist die Ergebnismenge vollständig berechnet, da Attribut- und Namensraumknoten keine Kinder haben. In Zeile 8 geben wir die Ergebnismenge zurück.

#### Terminierung und Korrektheit

Der Algorithmus 18 berechnet die Attribute bzw. Namensraumknoten eines Knotens. Der Algorithmus terminiert, da die Anzahl der Kinder eines Knotens immer endlich ist. Der Algorithmus ist korrekt, da alle Kinder des Knotens betrachtet und auf den geforderten Typ überprüft werden.

#### 4.2.13. analyseKnotentyp()

Der Algorithmus `analyseKnotentyp()` entfernt Attribut- und Namensraumknoten aus einer Knotenmenge. Der Algorithmus wird von den Algorithmen `analyseChild()`, `analyseDescendant()` und `analyseFollowing()` aufgerufen.

Der Algorithmus `analyseKnotentyp()` ist in Algorithmus 19 definiert. Beim Aufruf wird dem Algorithmus ein Parameter übergeben. Dieser Parameter enthält die Knotenmenge, aus der Attribut- und Namensraumknoten entfernt werden sollen.

---

**Algorithmus 19** `analyseKnotentyp(Knotenmenge)`

---

```
1: OUT ← Knotenmenge
2: for all (x, baum) ∈ Knotenmenge do
3:   hole (x, typ, z) aus  $Typ_{baum}$ 
4:   if typ = 'A' ∨ typ = 'N' then
5:     OUT ← OUT \ {(x, baum)}
6:   end if
7: end for
8: return OUT
```

---

#### Beschreibung

In der ersten Zeile speichern wir die Knotenmenge, die im Parameter übergeben wurde, als Ergebnismenge. In den Zeilen 2 – 7 betrachten wir nun jeden Knoten in dieser Knotenmenge. In Zeile 4 überprüfen wir, ob ein betrachteter Knoten ein Attribut- oder Namensraumknoten ist. Ist dies der Fall, wird er in Zeile 5 aus der Ergebnismenge entfernt. Die Ergebnismenge ist vollständig berechnet, wenn jeder Knoten aus der im Parameter übergebenen Knotenmenge betrachtet wurde. In Zeile 8 geben wir die Ergebnismenge zurück.

#### Terminierung und Korrektheit

Der Algorithmus 19 entfernt Attribut- und Namensraumknoten aus einer Knotenmenge. Der Algorithmus terminiert, da die übergebene Menge immer endlich ist. Er ist korrekt, da die Knoten in der Knotenmenge auf ihren Typ überprüft werden können und so korrekt entfernt werden.

#### 4.2.14. analyseKnotentest()

Der Algorithmus `analyseKnotentest()` führt einen Knotentest innerhalb eines Lokalisierungsschrittes aus. Bei einem Knotentest werden Knoten mit einem bestimmten Knotennamen oder -typ aus einer Knotenmenge ausgewählt. Ein Knotentest betrachtet dabei

den XPath-Knotentyp (Element-, Kommentar-, Textknoten, usw.) und nicht den Datentyp, den wir im formalen Modell annotieren. Der Algorithmus wird vom Algorithmus `analyseOperation()` aufgerufen.

Der Algorithmus `analyseKnotentest()` ist in Algorithmus 20 definiert. Beim Aufruf werden dem Algorithmus zwei Parameter übergeben. Der erste Parameter enthält die Knotenmenge, aus der die Knoten ausgewählt werden sollen. Der zweite Parameter enthält den Knotennamen oder den Knotentyp, auf dessen Grundlage die Knoten ausgewählt werden.

Beim Knotentest können wir einige Knotentypen nicht betrachten, da diese Knoten im XML-Schema nicht explizit beschrieben wird. So werden Kommentar-, Text- und processing-instruction-Knoten im XML-Schema nicht beschrieben. Diese Knoten können im Instanzdokument beliebig deklariert werden. Deshalb können wir statisch keine Voraussage über ihr Auftreten machen. Aus diesem Grund werden Kommentar-, Text- und processing-instruction-Knoten in unserer Analyse nicht berücksichtigt.

---

**Algorithmus 20** `analyseKnotentest(Knotenmenge, operand2)`

---

```
1: OUT  $\leftarrow$   $\emptyset$ 
2: if operand2 = node()  $\vee$  operand2 = „*“ then
3:   OUT  $\leftarrow$  Knotenmenge
4: else if operand2 = „namensraum:*“ then
5:   for all (x, baum)  $\in$  Knotenmenge do
6:     hole (x, (name, nr)) aus  $Name_{baum}$ 
7:     if nr = namensraum then
8:       OUT  $\leftarrow$  OUT  $\cup$  {(x, baum)}
9:     end if
10:  end for
11: else {Test auf den Namen der Knoten}
12:  for all (x, baum)  $\in$  Knotenmenge do
13:    hole (x, (name, namensraum)) aus  $Name_{baum}$ 
14:    if namensraum =  $\emptyset$  then
15:      namex  $\leftarrow$  name
16:    else
17:      namex  $\leftarrow$  „namensraum:name“
18:    end if
19:    if namex = operand2 then
20:      OUT  $\leftarrow$  OUT  $\cup$  {(x, baum)}
21:    end if
22:  end for
23: end if
24: return OUT
```

---

## Beschreibung

In der ersten Zeile erkennen wir einen Knotentest, der alle Knoten eines beliebigen Knotentyps auswählt oder einen Knotentest, der alle Knoten mit einem beliebigen Namen auswählt. Da diese Knotentest die Knotenmenge nicht verändern, speichern wir in Zeile 3 die Knotenmenge als Ergebnismenge, die im Parameter übergeben wurde.

In Zeile 4 erkennen wir einen Knotentest, der alle Knoten in einem bestimmten Namensraum mit einem beliebigen Namen auswählt. In den Zeilen 5 – 10 testen wir nun jeden Knoten aus der übergebenen Knotenmenge. Jeder Knoten, der in dem bestimmten Namensraum enthalten ist, wird in Zeile 8 in der Ergebnismenge gespeichert.

Die Zeilen 12 – 22 werden ausgeführt, wenn der Knotentest alle Knoten mit einem bestimmten Namen auswählt, die ggf. in einem bestimmten Namensraum enthalten sind. Dabei testen wir wieder jeden Knoten in der übergebenen Knotenmenge. Zunächst bestimmen wir in den Zeilen 14 – 18 für jeden Knoten den qualifizierten Namen und speichern diesen in der Variablen *name<sub>x</sub>*. Ist ein Knoten keinem Namensraum zugeordnet, entspricht der qualifizierte Name dem Namen des Knotens. Ist ein Knoten einem Namensraum zugeordnet, entspricht der qualifizierte Name dem Namensraum gefolgt von einem „:“ und dem Namen des Knotens. In den Zeilen 19 – 21 speichern wir nun jeden Knoten in der Ergebnismenge, dessen qualifizierte Name dem bestimmten Namensraum gefolgt von einem „:“ und dem bestimmten Namen entspricht. In Zeile 24 geben wir die Ergebnismenge zurück.

## Terminierung und Korrektheit

Der Algorithmus 20 führt einen Knotentest innerhalb eines Lokalisierungsschrittes aus. Bei einem Knotentest werden Knoten mit einem bestimmten Knotennamen oder -typ aus einer Knotenmenge ausgewählt. Ein Knotentest betrachtet dabei den XPath-Knotentyp (Element-, Kommentar-, Textknoten, usw.). Im Algorithmus werden Kommentar-, Text- und processing-instruction-Knoten nicht berücksichtigt, da sie nicht im XML-Schema beschrieben werden.

Der Algorithmus terminiert, da die übergebene Knotenmenge endlich ist. Der Algorithmus ist korrekt da, alle Knoten in der Knotenmenge berücksichtigt werden und der Name eines Knotens sowie die betrachteten Knotentypen korrekt überprüft werden können. Diese Informationen sind in den Bäumen annotiert, in denen die Knoten enthalten sind. Da wir für jeden Knoten den richtigen Baum betrachten, können wir den Knotentest korrekt analysieren.

### 4.2.15. analyseFkt()

Der Algorithmus `analyseFkt()` führt einen Funktionsaufruf innerhalb eines XPath-Ausdrucks auf. Die erlaubten Funktionen haben wir in Kapitel 3.1.5 definiert. Der Algorithmus wird vom Algorithmus `analyseOperation()` aufgerufen.

Der Algorithmus `analyseFkt()` ist in Algorithmus 21 definiert. Beim Aufruf werden dem Algorithmus vier Parameter übergeben. Der erste Parameter enthält die Funktion, die aufgerufen werden soll. Der zweite Parameter enthält den Zähler der temporären Variablen, unter der das Ergebnis des Funktionsaufruf gespeichert wird. Der dritte Parameter enthält den XPath-Ausdruck, der aktuell ausgewertet wird. Der XPath-Ausdruck ist in Form des Drei-Adress-Codes repräsentiert, wie wir in Kapitel 3.2 beschrieben haben. Der vierte Parameter enthält die bereits berechneten Wertebereiche.

## Beschreibung

In den Zeilen 3 – 11 ermitteln wir die Parameter für den Funktionsaufruf mit Hilfe des Drei-Adress-Codes. Im Drei-Adress-Code werden Parameter durch Quadrupel repräsentiert. Repräsentiert ein Quadrupel einen Parameter für den betrachteten Funktionsaufruf, enthält das zweite Element des Quadrupels die Operation „*param*“. Das vierte Element des Quadrupels verweist auf die temporäre Variable, in der das Ergebnis des Funktionsaufrufs gespeichert wird. In der Zeile 6 überprüfen wir den Drei-Adress-Code, der im dritten Parameter übergeben wurde, nach solchen Quadrupeln. Für jedes gefundene Quadrupel speichern wir in Zeile 7 den Parameter für den Funktionsaufruf in einer Variablen  $p_i$ . Der Funktionsparameter ist im Quadrupel als drittes Element enthalten. Haben wir alle Funktionsparameter gefunden, speichern wir in Zeile 12 die Anzahl der gefundenen Parameter in der Variablen *anzahl*.

In den Zeilen 13 – 21 lösen wir nun Verweise auf Variablen im Parameter auf. Verweist ein Parameter auf eine Variable, ermitteln wir in Zeile 15 den berechneten Wertebereich der Variablen. Dieser Wertebereich wird in der Menge  $X_i$  gespeichert. Wie bereits erwähnt, ist ein berechneter Wertebereich eine Menge von Tupeln. Das erste Element des Tupels repräsentiert den Wertebereich als Element der vollständigen Verbände aus den Definitionen 1, 7 und 18 oder als Knotenmenge, wie in Definition 21 definiert. Das zweite Element des Tupels enthält die Menge der Variablen, die bei der Berechnung des Wertebereichs verwendet wurden. Für einen Parameter, der nicht auf eine Variable verweist, speichern wir die Menge  $X_i$  in Zeile 18. Die Menge  $X_i$  enthält in diesem Fall den Parameter selbst als Wertebereich und die Menge der verwendeten Variablen ist leer.

Bei einem Funktionsaufruf müssen wir jede Kombination der Elemente aus den Wertebereichen der Funktionsparameter betrachten. Deshalb bilden wir in Zeile 22 das Kreuzprodukt über alle Mengen  $X_i$ . In den Zeilen 23 – 35 führen wir nun die Funktionsaufrufe aus und berechnen den Wertebereich. In den Zeilen 24 und 25 ermitteln wir zunächst ein Element aus dem Kreuzprodukt und löschen dieses Element aus dem Kreuzprodukt. In den Zeilen 28 – 32 ermitteln wir nun die konkreten Parameter  $x_1$  bis  $x_{anzahl}$  für einen Funktionsaufruf. In Zeile 30 ermitteln wir zusätzlich die Menge der Variablen, die bei der Berechnung des Funktionswertes verwendet werden.

Nun führen wir in Zeile 33 einen Funktionsaufruf mit den Parametern  $x_1, \dots, x_{anzahl}$  aus. Die aufgerufene Funktion ist in Kapitel 3.1.5 definiert. Das Ergebnis des Funktionsaufrufs speichern wir in der Variablen *Tmp*. Schließlich speichern wir in Zeile 34 das Ergebnis

**Algorithmus 21** analyseFkt(funktion, zähler, Ausdruck, Werte)

---

```

1: OUT  $\leftarrow \emptyset$ , MeineListe  $\leftarrow$  Ausdruck
2:  $i \leftarrow 1$ ,  $j \leftarrow 1$ ,
3: while MeineListe  $\neq \emptyset$  do
4:   hole (tmpj, operation, fktparam, fkt) aus MeineListe
5:   MeineListe  $\leftarrow$  MeineListe  $\setminus$  {(tmpj, operation, fktparam, fkt)}
6:   if operation = „param“  $\wedge$  fkt = tmpzähler then
7:     pi  $\leftarrow$  fktparam
8:      $i \leftarrow i + 1$ 
9:   end if
10:   $j \leftarrow j + 1$ 
11: end while
12: anzahl  $\leftarrow i$ 
13: while  $i > 0$  do
14:   if pi ist Verweis then
15:     hole (pi, W) aus Werte
16:     Xi  $\leftarrow$  W
17:   else
18:     Xi  $\leftarrow$  {(pi,  $\emptyset$ )}
19:   end if
20:    $i \leftarrow i - 1$ 
21: end while
22: Kreuzprodukt  $\leftarrow$  X1  $\times$  X2  $\times$  ...  $\times$  Xanzahl
23: while Kreuzprodukt  $\neq \emptyset$  do
24:   hole (z1, ..., zanzahl) aus Kreuzprodukt
25:   Kreuzprodukt  $\leftarrow$  Kreuzprodukt  $\setminus$  {(z1, ..., zanzahl)}
26: Besuchtaktuell  $\leftarrow \emptyset$ 
27:  $i \leftarrow 1$ 
28: while  $i \leq$  anzahl do
29:   (xi, Besuchtxi)  $\leftarrow$  zi
30:   Besuchtaktuell  $\leftarrow$  Besuchtaktuell  $\cup$  Besuchtxi  $\cup$  {pi}
31:    $i \leftarrow i + 1$ 
32: end while
33: Tmp  $\leftarrow$  Rückgabewert der Funktion funktion mit den Parametern
      X1, ..., Xanzahl
34: OUT  $\leftarrow$  OUT  $\cup$  {(Tmp, Besuchtaktuell)}
35: end while
36: return OUT

```

---

des Funktionsaufrufs sowie die Menge der verwendeten Variablen in der Ergebnismenge *OUT*. Wurden alle Elemente aus dem Kreuzprodukt betrachtet, ist die Ergebnismenge vollständig berechnet und wir geben sie in Zeile 36 zurück.

### **Terminierung und Korrektheit**

Der Algorithmus 21 führt einen Funktionsaufruf innerhalb eines XPath-Ausdrucks auf. Die erlaubten Funktionen haben wir im formalen Modell aus in Kapitel 3.1.5 definiert. Da die Wertebereiche der Funktionsparameter durch Mengen angegeben werden, bilden wir das Kreuzprodukt dieser Wertebereiche und führen für jedes Element des Kreuzprodukts den Funktionsaufruf aus. Den Aufruf von Funktionen ohne Parameter haben wir bereits im Drei-Adress-Code ausgeführt.

Der Algorithmus terminiert, da für einen Funktionsaufruf nur endlich viele Parameter angegeben werden. Deshalb ist auch das Kreuzprodukt der Wertebereiche der Parameter endlich. Der Algorithmus ist korrekt, da für die Ausführung eines Funktionsaufrufs die Funktionen aus dem formalen Modell verwendet werden. Da wir das Kreuzprodukt der Wertebereiche für Funktionsparameter betrachten, ermitteln wir alle möglichen Ergebnisse des Funktionsaufrufs.

### **Zusammenfassung**

In diesem Kapitel haben wir eine statische Datenanalyse für BPEL-Prozesse vorgestellt. Bei der Datenanalyse haben wir die Wertebereiche für Variablen und Bedingungen in BPEL-Prozessen berechnet. Die Grundlage für die vorgestellte Analyse bildet das Datenmodell aus Kapitel 3. Da wir in diesem Datenmodell einen CSSA-Graphen für die Repräsentation des BPEL-Prozesses verwenden, berechnet die Analyse sogar den Wertebereich einer Variablen zu einem bestimmten Zeitpunkt im BPEL-Prozess.

Die Wertebereiche berechnen wir mit Hilfe der abstrakten Interpretation von im BPEL-Prozess enthaltenen XPath-Ausdrücke. Das formale Datenmodell aus Kapitel 3.1 bietet uns dafür eine Grundlage. In diesem formalen Modell haben wir eine Abstraktion für die Datenwerte in XPath-Ausdrücken entwickelt. Diese Abstraktionen haben wir in unserer Analyse verwendet, um das Ergebnis eines XPath-Ausdrucks zu berechnen.

## 5. Zusammenfassung und Ausblick

In der vorliegenden Arbeit haben wir ein Verfahren zur abstrakten Interpretation von XPath-Ausdrücken in BPEL-Prozessen entwickelt. Dafür haben wir zunächst ein Datenmodell für BPEL-Prozesse vorgestellt. Auf Grundlage dieses Datenmodells haben wir dann eine statische Analyse beschrieben, welche die Wertebereiche von Variablen und Bedingungen in BPEL-Prozessen berechnet. In diesem Kapitel fassen wir die Ergebnisse der vorliegenden Arbeit zusammen und geben einen Ausblick für weiterführende Arbeiten.

### 5.1. Zusammenfassung der Ergebnisse

In der vorliegenden Arbeit haben wir eine statische Datenanalyse für BPEL-Prozesse entwickelt. Mit dieser Datenanalyse können wir die Wertebereiche von Variablen und Bedingungen in BPEL-Prozessen berechnen. Die statische Datenanalyse, die in dieser Arbeit vorgestellt wurde, ist eine sichere Analyse. Das bedeutet, dass die Ergebnismenge der Analyse eine Obermenge des Ergebnisses ist, das die Auswertung des XPath-Ausdruck zur Laufzeit liefert.

Da für BPEL-Prozesse noch keine umfassende Datenanalyse existiert, haben wir zunächst ein *Datenmodell* für BPEL-Prozesse erstellt. In diesem Datenmodell wird der Kontrollfluss und die Datenabhängigkeiten eines BPEL-Prozesses durch einen CSSA-Graphen repräsentiert. Der CSSA-Graph bietet eine kompakte Darstellung und ermöglicht uns somit die gemeinsame Repräsentation des Kontrollflusses und der Datenabhängigkeiten. Weiterhin kann durch die Verwendung eines CSSA-Graphen der Wert einer Variablen zu einem bestimmten Zeitpunkt im BPEL-Prozess berechnet werden. Bei der Erstellung des CSSA-Graphen haben wir die bestehende CSSA-Form für BPEL-Prozesse aus [MMG<sup>+</sup>07] als Grundlage verwendet und um einige Aspekte erweitert, die in der Datenanalyse benötigt werden.

So haben wir eine Nachricht, die von der `receive`-, `invoke`- bzw. `pick`-Aktivität empfangen wird, präziser repräsentiert, da sie ein Datum im BPEL-Prozess darstellt. Weiterhin haben wir die Repräsentation von Wertzuweisungen im CSSA-Graphen vervollständigt. Dabei haben wir die `assign`-Aktivität verändert und den CSSA-Graphen um eine Repräsentation von Variableninitialisierungen im BPEL-Prozess erweitert. Außerdem haben wir den CSSA-Graphen um die Repräsentation der `repeatUntil`- und der `scope`-Aktivität erweitert. Die grundlegendste Änderung des CSSA-Graphen aus [MMG<sup>+</sup>07] betrifft jedoch die Repräsentation der XPath-Ausdrücke in den einzelnen Aktivitäten. Bisher wurden die XPath-Ausdrücke lediglich als annotierter Text im CSSA-Graphen

repräsentiert. In der vorliegenden Arbeit haben wir den CSSA-Graphen um eine analysierbare Repräsentation der XPath-Ausdrücke erweitert.

Für die Repräsentation der XPath-Ausdrücke in BPEL-Prozessen haben wir den Drei-Adress-Code vorgestellt. Der Drei-Adress-Code ist eine sequentielle Darstellung eines abstrakten Syntaxbaums, die eine schnelle Traversierung über den Syntaxbaum ermöglicht. Sowohl das Erstellen eines abstrakten Syntaxbaums als auch die Überführung in den Drei-Adress-Code sind Standardtechniken aus dem Compilerbau. In der vorliegenden Arbeit konnten wir deshalb auf diese Techniken zurückgreifen. Schließlich haben wir den Drei-Adress-Code in den CSSA-Graphen eingebettet und haben damit eine gemeinsame Repräsentation des BPEL-Prozesses und der enthaltenen XPath-Ausdrücke geschaffen. Auf Grundlage dieser Repräsentation können die Daten eines BPEL-Prozesses analysiert werden.

Zusätzlich haben wir im Datenmodell ein *formales Modell* für XPath 1.0 entwickelt. Dieses formale Modell ermöglicht eine Abstraktion von den konkreten Datenwerten in XPath-Ausdrücken. Da in dieser Arbeit eine Datenanalyse bzgl. der Wertebereiche vorgestellt wird, haben wir ein formales Modell entwickelt, mit dem die Wertebereiche möglichst genau eingegrenzt werden können. Im formalen Modell haben wir zunächst eine abstrakte Semantik für die Datentypen in XPath beschrieben. Dabei haben wir Galois Korrespondenzen verwendet, um eine Abstraktion ohne Informationsverlust zu gewährleisten. Auf Grundlage der abstrakten Semantik haben wir im formalen Modell zusätzlich die Operationen und Funktionen aus XPath definiert. Das formale Modell für XPath ermöglicht uns die abstrakte Interpretation von XPath-Ausdrücken.

Im formalen Modell für XPath 1.0 haben wir auch den Datentyp `node-set` berücksichtigt. Dieser Datentyp repräsentiert Knotenmengen, deren Knoten jeweils ein XML-Element oder ein Attribut ist. Der Typ eines XML-Elements wird in BPEL-Prozessen durch ein XML-Schema beschrieben. Da XML-Elemente hierarchisch strukturiert sind, haben wir für die Abstraktion eines XML-Elements bzw. Attributs einen Knoten in einem Baum gewählt. Der Baum repräsentiert dabei das XML-Schema, welches das XML-Element bzw. Attribut typisiert. Damit ist die Abstraktion für eine Knotenmenge in XPath eine Menge von Knoten in Bäumen.

Bei der Abstraktion einer Knotenmenge im formalen Modell haben wir keine Galois Korrespondenz verwendet. Da wir den konkreten BPEL-Prozess nicht kennen, haben wir auch keine Informationen über das XML-Schema sowie den XPath-Lokalisierungspfad, der eine Knotenmenge liefert. Folglich kennen wir den Zustandsraum für Knotenmengen in XPath nicht. Möchten wir für die Abstraktion von Knotenmengen eine Galois Korrespondenz definieren, müssen wir deshalb alle möglichen XML-Schemata und jeden möglichen Lokalisierungspfad berücksichtigen. Darauf haben wir in dieser Arbeit verzichtet. Stattdessen haben wir für die Abstraktion die erwähnte hierarchische Struktur definiert, die uns eine Analyse ermöglicht. In dieser hierarchischen Struktur haben wir das korrespondierende XML-Schema durch Annotationen repräsentiert. Weiterhin haben wir ein Verfahren angegeben, wie ein gegebenes XML-Schema in der hierarchischen Struktur annotiert werden kann.

Auf Grundlage unseres Datenmodells haben wir im zweiten Teil der Arbeit eine statische Datenanalyse für BPEL-Prozesse vorgestellt. Die Korrektheit und die Terminierung dieser Analyse haben wir argumentativ gezeigt. Die Analyse berechnet die Wertebereiche von Variablen und Bedingungen in BPEL-Prozessen. Für die Analyse haben wir einen Algorithmus `analyseCSSA()` angegeben, der über den CSSA-Graphen traversiert. Bei der Traversierung betrachtet der Algorithmus jeden Knoten des CSSA-Graphen und überprüft die Dateninformationen. Gegebenfalls ruft der Algorithmus einen weiteren Algorithmus `analyseXPath()` auf, der die abstrakte Interpretation eines XPath-Ausdrucks übernimmt. Ein XPath-Ausdruck wird in BPEL-Prozessen standardmäßig bei Wertzuweisungen oder Bedingungen angegeben. Das Ergebnis der abstrakten Interpretation speichern wir deshalb als Wertebereich für eine Variable oder eine Bedingung im BPEL-Prozess. Wurden alle Knoten im CSSA-Graphen analysiert, wird die Ergebnismenge zurückgegeben.

Die abstrakte Interpretation wird auf Grundlage des formalen Modells für XPath ausgeführt. Mit der *abstrakten Interpretation* berechnen wir den Wertebereich, den ein XPath-Ausdruck liefern kann. Dazu werden die Operationen, Funktionen und Lokalisierungspfade im XPath-Ausdruck auf Grundlage der abstrakten Datenwerte ausgeführt. Die Operationen sowie die Funktion aus der Grundbibliothek von XPath haben wir im formalen Modell definiert. Bei Lokalisierungspfaden traversieren wir entsprechend der angegebenen Achsen über den gegebenen Baum und wenden auf die so ermittelten Knoten den Knotentest an. Die Filterung mit Prädikaten berücksichtigen wir in dieser Arbeit nicht. Wir können zwar den im Prädikat enthaltenen XPath-Ausdruck analysieren, allerdings muss das Ergebnis für die Filterung einer Knotenmenge statisch interpretiert werden. Diese Interpretation überlassen wir weiterführenden Arbeiten.

Zusätzlich zu den Funktionen aus der Grundbibliothek von XPath stellt BPEL zwei weitere Funktionen für die Datenmanipulation zur Verfügung. In der vorliegenden Arbeit unterstützen wir auch diese beiden Funktionen. Allerdings können wir das Ergebnis der Funktion `doXslTransform()` nicht analysieren. Bei der Berechnung des Wertebereichs dieser Funktion müsste eine statische Analyse der XSLTransformation angewendet werden. Eine solche Analyse existiert noch nicht und übersteigt den Rahmen dieser Arbeit. Da die Funktion nur in Zuweisungen verwendet wird, abstrahieren wir von dem Ergebnis der Funktion und verwenden den bekannten Wertebereich der Variablen, der das Ergebnis der Funktion zugewiesen wird.

Bei der Berechnung des Wertebereichs einer BPEL-Variablen haben wir den Wertebereich zu einem bestimmten Zeitpunkt im BPEL-Prozess ermittelt. Dies ergibt sich aus der Verwendung des CSSA-Graphen, der jeder Variablen einen Index zuordnet. Unterschiedliche Indizes für eine Variable repräsentieren unterschiedliche Zeitpunkte im BPEL-Prozess. Bei der Interpretation des berechneten Wertebereichs einer Variablen, die in verschiedenen Namensräumen definiert ist, muss der Index gesondert beachtet werden. So dürfen zwei nicht überlappende Namensräume Variablen mit dem gleichen Namen enthalten. Im CSSA-Graphen werden diese Variablen durch unterschiedliche Indizes repräsentiert. Betrachten wir beispielsweise eine Variable  $x$ , die in einem Namens-

raum die Indizes 1 – 5 und in einem anderen Namensraum die Indizes 10 – 15 erhält. Dann repräsentieren die Variablen  $x_1, \dots, x_5$  die Variable  $x$  im ersten Namensraum. Die Variablen  $x_{10}, \dots, x_{15}$  repräsentieren die andere Variable  $x$ , die im zweiten Namensraum enthalten ist. Die beiden Variablen  $x$  in den verschiedenen Namensräumen können also nicht mit Hilfe der Namen, sondern nur mit Hilfe der Indizes unterschieden werden. Dies repräsentiert aber gerade das Verhalten von lokalen Variablen in Namensräumen.

In BPEL-Prozessen können andere Sprachen als XPath, wie z.B. Java, für die Beschreibung von Datenmanipulationen verwendet werden. Diese Sprachen sind nicht Gegenstand dieser Arbeit und werden deshalb in der vorgestellten Datenanalyse nicht berücksichtigt. Allerdings kann das Datenmodell zur Unterstützung solcher Sprachen leicht angepasst werden. So kann der CSSA-Graph nach kleinen Änderungen an den kommunikationsrelevanten Aktivitäten weiterhin für die Repräsentation des Kontrollflusses und der Datenabhängigkeiten im BPEL-Prozess verwendet werden. Zur Repräsentation der Datenmanipulationen in anderen Sprachen kann das Datenmodell analog zu unserem vorgestellten Modell für XPath beschrieben werden.

## 5.2. Ausblick

Die Ergebnisse dieser Arbeit können vielfältig in weiterführenden Arbeiten verwendet werden. Einerseits können die Analyseergebnisse zur Optimierung der BPEL-Prozesse oder zur Verbesserung von Verifikationstechniken für BPEL-Prozesse verwendet werden. Andererseits gibt es viele denkbare Ansätze, die auf dieser Arbeit aufbauen können. Beispielsweise ermöglicht eine Implementierung der Analyse die Validierung der Ergebnisse. Weiterhin gibt es denkbare Ansätze zur Verbesserung der Analyse. Aber auch anderer Analysen können aufbauend auf dem vorgestellten Datenmodell beschrieben werden.

Betrachten wir zunächst die Verwendung der Analyseergebnisse für die *Optimierung* von BPEL-Prozessen. Analog zu den bekannten Optimierungstechniken im Compilerbau können beispielsweise tote Pfade im BPEL-Prozess erkannt und entfernt werden. Das Entfernen toter Pfade ermöglicht eine schnellere Ausführungszeit von BPEL-Prozessen, da die Aktivitäten der toten Pfade nicht mehr von der Engine geladen werden müssen und die Dead-Path-Elimination in BPEL weniger Zeit in Anspruch nimmt. Die Dead-Path-Elimination in BPEL ist obligatorisch. Sie entfernt die toten Pfade rückwirkend, nachdem sie von der Engine geladen wurden. Des Weiteren können durch Konstantenpropagierung evtl. einige XPath-Ausdrücke in BPEL-Prozessen bereits vor der Ausführung ausgewertet werden.

Die Optimierung mit Hilfe der Analyseergebnisse ist besonders interessant, wenn nicht nur ein einzelner BPEL-Prozess betrachtet wird, sondern vielmehr eine *Choreographie* von BPEL-Prozessen. Sendet ein BPEL-Prozess eine Nachricht an einen Partnerprozess, ist für den Partnerprozess i.A. nur der Datentyp der Nachricht bekannt. Wenn der sendende BPEL-Prozess durch die vorgestellte Analyse den Wertebereich seiner Nachricht einschränken kann, kann in einer Choreographie ggf. diese Information für den Partnerprozess zur Verfügung gestellt werden. Damit hat der Partnerprozess eine genauere

Beschreibung der Nachrichten, die er empfangen wird. Bislang muss ein Partnerprozess jeden Wert im Datentyp einer Nachricht betrachten. Können wir aber den Wertebereich für die Nachricht einschränken, kann die vorgestellte Analyse für den Partnerservice ggf. genauere Wertebereiche berechnen. Damit kann der Partnerservice dann u.U. effizienter optimiert werden, weil beispielsweise mehr tote Pfade erkannt werden können, als wenn der Partnerprozess für sich allein betrachtet wird.

Neben der Optimierung können die Analyseergebnisse auch für die Verbesserung von bestehenden *Verifikationstechniken* für BPEL-Prozesse verwendet werden. Solche Verifikationstechniken werden beispielsweise in [Sch05] und [MM06] vorgestellt. Bisher vernachlässigen diese Techniken Dateninformationen. Werden die Ergebnisse unserer Datenanalyse aber in die Betrachtungen einbezogen, können die Verifikationstechniken aussagekräftiger werden.

Um das Analyseergebnis dieser Arbeit zu verbessern, können folgende vier Aspekte betrachtet werden: Handler in BPEL, XSLTransformation, Schleifen und Prädikate in XPath-Lokalisierungspfade. *Handler* in BPEL-Prozessen werden in dieser Arbeit vernachlässigt, da sie nicht im verwendeten CSSA-Graphen repräsentiert werden. Allerdings können Handler analysiert werden, sobald sie im CSSA-Graphen repräsentiert werden. Die abstrakte Interpretation der XPath-Ausdrücke muss dazu nicht verändert werden.

Die statische Analyse von *XSLTransformationen* wird in dieser Arbeit vernachlässigt, da sie noch nicht existiert und den Rahmen der Arbeit übersteigt. Statt den Wertebereich des Ergebnisses einer XSLTransformation zu analysieren, verwenden wir in dieser Arbeit den Wertebereich, der für den Datentyp des Ergebnisses angegeben ist. Der Informationsverlust ist dementsprechend groß. Wird eine statische Analyse bzgl. der Wertebereiche von XSLTransformationen entwickelt, kann der Informationsverlust verhindert werden. Eine Analyse von XSLTransformationen kann dazu leicht in die Analyse aus dieser Arbeit eingebunden werden.

Das Analyseergebnis für *Schleifen* kann verbessert werden, da wir ggf. eine widening-Operation anwenden müssen. Durch die Anwendung einer widening-Operation wird eine endlose Analyse vermieden, indem von dem tatsächlichen Wert eines Datums abstrahiert und stattdessen der kleinste bzw. größte Wert für das betrachtete Datum verwendet wird. Der Informationsverlust ist dementsprechend groß. Wenn aber die Bedingung für den Schleifeneintritt analysiert wird, können die Werte ermittelt werden, die in der Schleife auftreten dürfen. Mit Hilfe dieser Information kann dann der Wertebereich nach der widening-Operation ggf. wieder verkleinert werden.

Da wir in der vorgestellten Analyse das Filtern von Knotenmengen durch *Prädikate* in Lokalisierungspfaden vernachlässigen, kann eine umfassende Betrachtung der Prädikate das Analyseergebnis verbessern. Der XPath-Ausdruck in einem betrachteten Prädikat kann mit der vorgestellten Analyse ausgewertet werden. Da ein Prädikat die Eigenschaften der Knoten in der Ergebnismenge spezifiziert, muss das Prädikat aber für die Filterung einer Knotenmenge interpretiert werden. Das Ergebnis dieser Interpretation kann beispielsweise in der Knotenmenge annotiert werden.

Schließlich ist noch zu erwähnen, dass weitere Datenanalysen für BPEL auf dieser Arbeit aufbauen können. Möchte man noch andere Eigenschaften der Daten als den Wertebereich betrachten, kann das Datenmodell als Grundlage für solche Analysen verwendet werden. Betrachtet man dabei weiterhin XPath als Beschreibungssprache für Datenmanipulationen, kann das vorgestellte formale Modell für XPath verwendet und ggf. erweitert werden. Werden auch andere Beschreibungssprachen für Datenmanipulationen betrachtet, kann der CSSA-Graph nach kleinen Änderungen an den kommunikationsrelevanten Aktivitäten weiterhin verwendet werden. Das Datenmodell für andere Beschreibungssprachen kann analog zu unserem vorgestellten Modell für XPath entwickelt werden.

## A. Mathematische Grundlagen

Zunächst beschreiben wir einige grundlegende mathematische Definitionen, die in dieser Arbeit benötigt werden. Danach definieren wir die Galois Korrespondenz.

### Definition 22 (Halbordnung)

Eine Relation  $R \subseteq A \times A$  heißt Halbordnung auf  $A$ , genau dann wenn gilt:

- $\forall x \in A : x R x$ ;
- $\forall x, y \in A : x R y \wedge y R x \Rightarrow x = y$ ;
- $\forall x, y, z \in A : x R y \wedge y R z \Rightarrow x R z$ .

$(A, R)$  heißt halbgeordnete Menge. ┘

### Definition 23 (aufrunden)

Sei  $x \in \mathbb{R}$ . Der Wert  $y$  entsteht durch aufrunden von  $x$ :

$$y = \lceil x \rceil =_{\text{def}} y \in \mathbb{N} \wedge y \geq x \wedge \forall z \in \mathbb{N} \text{ mit } z \geq x : z \geq y$$

Die Werte  $\infty$  und  $-\infty$  verändern sich beim aufrunden nicht. Das bedeutet:

$$\begin{aligned} \lceil \infty \rceil &= \infty \text{ bzw.} \\ \lceil -\infty \rceil &= -\infty \end{aligned}$$

Entsprechendes gilt für  $-0$  und  $0$ . ┘

### Definition 24 (abrunden)

Sei  $x \in \mathbb{R}$ . Der Wert  $y$  entsteht durch abrunden von  $x$ :

$$y = \lfloor x \rfloor =_{\text{def}} y \in \mathbb{N} \wedge y \leq x \wedge \forall z \in \mathbb{N} \text{ mit } z \leq x : z \leq y$$

Die Werte  $\infty$  und  $-\infty$  verändern sich beim abrunden nicht. Das bedeutet:

$$\begin{aligned} \lfloor \infty \rfloor &= \infty \text{ bzw.} \\ \lfloor -\infty \rfloor &= -\infty \end{aligned}$$

Entsprechendes gilt für  $-0$  und  $0$ . ┘

### A.1. Die Galois Korrespondenz

#### Definition 25 (vollständiger Verband)

$(L, \sqsubseteq, \sqcup, \sqcap, \perp, \top)$  ist ein vollständiger Verband, wenn  $(L, \sqsubseteq)$  eine halbgeordnete Menge ist, so dass alle Teilmengen von  $L$  eine kleinste obere Schranke  $\sqcup$  und eine größte untere Schranke  $\sqcap$  besitzen.  $\perp$  repräsentiert das Nullelement und  $\top$  das Einselement, mit  $\perp = \sqcup \emptyset = \sqcap L$  und  $\top = \sqcap \emptyset = \sqcup L$ .  $\lrcorner$

#### Definition 26 (Galois Korrespondenz)

Seien  $L$  und  $M$  vollständige Verbände. Sei weiterhin  $\alpha : L \rightarrow M$  und  $\gamma : M \rightarrow L$  Funktionen.  $(L, \alpha, \gamma, M)$  ist eine Galois Korrespondenz zwischen  $L$  und  $M$ , gdw.  $\alpha$  und  $\gamma$  monotone Funktionen sind, für die gilt:

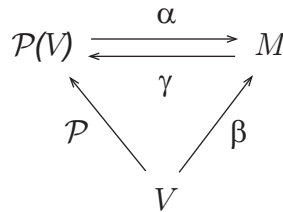
$$\gamma(\alpha(l)) \sqsupseteq l \quad \forall l \subseteq L \tag{A.1}$$

$$\alpha(\gamma(m)) \sqsubseteq m \quad \forall m \subseteq M \tag{A.2}$$

$\lrcorner$

Die Eigenschaften A.1 und A.2 sichern zu, dass bei der Abstraktion die Informationen nicht verloren gehen, es wird nur unpräziser.

Betrachten wir nun eine Funktion  $\beta : V \rightarrow M$ . Diese ordnet einem konkreten Wert aus der Menge  $V$  das entsprechende Element in der Menge  $M$  zu, das den konkreten Wert „am Besten“ beschreibt (z.B.  $\beta(-5) = \{-\}$ ;  $\beta(5) = \{+\}$ ). Die Funktion  $\beta$  wird als Repräsentationsfunktion bezeichnet. Mit Hilfe einer Repräsentationsfunktion können die Funktionen  $\alpha$  und  $\gamma$  definiert werden, so dass  $(\mathcal{P}(V), \alpha, \gamma, M)$  eine Galois Korrespondenz ist.



**Satz A.1** Sei  $\beta : V \rightarrow M$  eine Repräsentationsfunktion. Dann kann die Abstraktionsfunktion  $\alpha$  und die Konkretisierungsfunktion  $\gamma$  mit Hilfe der Funktion  $\beta$  definiert werden, so dass  $(\mathcal{P}(V), \alpha, \gamma, M)$  eine Galois-Korrespondenz bildet. Dabei werden  $\alpha$  und  $\gamma$  wie folgt definiert:

$$\begin{aligned}
 \alpha(V') &= \sqcup \{ \beta(v) \mid v \in V' \} \quad \text{für } V' \subseteq V \\
 \gamma(m) &= \{ v \in V \mid \beta(v) \sqsubseteq m \} \quad \text{für } m \in M
 \end{aligned}$$

Der Beweis dieses Satzes ist in [NNH05] auf Seite 237 enthalten.

## B. Hilfsalgorithmen

### B.1. Hilfsalgorithmen für den Algorithmus `analyseCSSA()`

#### B.1.1. `teilgraphFlow()`

Der Algorithmus `teilgraphFlow()` ermittelt einen Teilgraphen, der einem Flow entspricht. Der Teilgraph wird aus einem CSSA-Graphen ermittelt. Der ermittelte Teilgraph enthält den Knoten, der den Anfang des Flows repräsentiert, die Knoten, die die Aktivitäten innerhalb des Flows repräsentieren und den Knoten, der das Ende des Flows repräsentiert. Der Algorithmus wird vom Algorithmus `analyseCSSA()` aufgerufen.

Der Algorithmus `teilgraphFlow()` ist in Algorithmus 22 definiert. Beim Aufruf werden dem Algorithmus zwei Parameter übergeben. Der erste Parameter enthält den CSSA-Graphen, aus dem der Teilgraph ermittelt werden soll. Der zweite Parameter verweist auf den Startknoten des Flows, dessen Teilgraph ermittelt werden soll.

#### Beschreibung

Wie bereits der Algorithmus `analyseCSSA()` traversiert der Algorithmus `teilgraphFlow()` in einer Variante der Breitensuche über den CSSA-Graphen. Dies ist sehr wichtig, da innerhalb eines Flows weitere Flows enthalten sein können. Um das Ende des Flows, dessen Teilgraph ermittelt werden soll, zu erkennen, müssen wir die enthaltenen Flows zählen. Erst wenn alle enthaltenen Flows beendet sind, können wir das Ende des Flows, dessen Teilgraph ermittelt werden soll, erreichen.

In den Zeilen 1 – 3 werden die Startwerte des Algorithmus initialisiert. Die Variable *flows* enthält die Anzahl der betretenen Flows. Da der Parameter *startknoten* einen Knoten enthält, der den Anfang eines Flows repräsentiert, initialisieren wir in Zeile 1 die Variable *flows* mit dem Wert 1. Der Wahrheitswert in Variable *weiter* wird im Algorithmus als Abbruchbedingung verwendet. Ist der Wert der Variablen *true*, haben wir noch nicht alle Knoten im Flow besucht und die Traversierung über den CSSA-Graph wird fortgesetzt. Deshalb initialisieren wir in Zeile 1 die Variable *weiter* mit dem Wert *true*. Sobald das Ende des Flows erkannt wurde, enthält die Variable *weiter* den Wert *false*. In Zeile 2 werden die Variablen *A* und *Kopie<sub>A</sub>* initialisiert, die wie im Algorithmus `analyseCSSA()` für die Realisierung der Breitensuche verwendet werden.

In Zeile 3 werden die Variablen initialisiert, die den Teilgraph repräsentieren. Die Variable *V* repräsentiert die Knoten des Teilgraphs, der ermittelt werden soll. Da der übergebene Startknoten im Teilgraph enthalten ist, initialisieren wir in Zeile 3 die Variable *V* mit

---

**Algorithmus 22:** teilgraphFlow(CSSA-Graph, startknoten)

```

1: flows  $\leftarrow$  1, weiter  $\leftarrow$  true
2: A  $\leftarrow$   $\emptyset$ , KopieA  $\leftarrow$  {startknoten}
3: V  $\leftarrow$  {startknoten}, E  $\leftarrow$   $\emptyset$ 
4: for all k  $\in$  nachfolger(startknoten) do
5:   V  $\leftarrow$  V  $\cup$  {k}
6:   E  $\leftarrow$  E  $\cup$  {(startknoten, k)}
7: end for
8: Besucht  $\leftarrow$  {startknoten}
9: while weiter do
10:  for all x  $\in$  KopieA do
11:    A  $\leftarrow$  A  $\cup$  nachfolger(x)
12:  end for
13:  KopieA  $\leftarrow$  A
14:  while A  $\neq$   $\emptyset$  do
15:    aktuell  $\leftarrow$  hole Knoten aus A
16:    A  $\leftarrow$  A  $\setminus$  {aktuell}
17:    if aktuell.Typ =  $\phi$  then
18:      (startSchleife, Schleife)  $\leftarrow$  startZyklus(CSSA-Graph, aktuell, aktuell, Besucht)
19:      Schleife  $\leftarrow$  Schleife  $\setminus$  Besucht
20:    else
21:      startSchleife  $\leftarrow$  false
22:    end if
23:    if vorgänger(aktuell)  $\subseteq$  Besucht  $\vee$  startSchleife then
24:      if startSchleife  $\wedge$  vorgänger(aktuell)  $\subseteq$  Besucht then
25:        Besucht  $\leftarrow$  Besucht  $\cup$  {aktuell}
26:        A  $\leftarrow$  A  $\cup$  (nachfolger(aktuell)  $\setminus$  Besucht)
27:        KopieA  $\leftarrow$  KopieA  $\setminus$  {aktuell}  $\cup$  A
28:      else
29:        if aktuell.Typ = begin flow then
30:          flows  $\leftarrow$  flows + 1
31:        else if aktuell.Typ = end flow then
32:          flows  $\leftarrow$  flows - 1
33:          if flows = 0 then
34:            weiter  $\leftarrow$  false
35:          end if
36:        end if
37:        if weiter then
38:          Nachfolger  $\leftarrow$  nachfolger(aktuell)
39:          for all k  $\in$  Nachfolger do
40:            V  $\leftarrow$  V  $\cup$  {k}
41:            E  $\leftarrow$  E  $\cup$  {(aktuell, k)}

```

```
42:         end for
43:     end if
44:     if not(startSchleife) then
45:         Besuchte ← Besuchte ∪ {aktuell}
46:     end if
47: end if
48: else
49:     KopieA ← KopieA \ {aktuell}
50: end if
51: end while
52: end while
53: return ((V,E), aktuell)
```

---

einer Menge, die nur den Startknoten enthält. Die Variable  $E$  repräsentiert die Kanten des Teilgraphs. Die Variable  $E$  wird in Zeile 3 mit der leeren Menge initialisiert.

In den folgenden Zeilen 4 – 7 speichern wir die Nachfolgeknoten des Startknoten in der Menge  $V$  und die Kanten vom Startknoten zu dessen Nachfolgeknoten in der Menge  $E$ . Da wir nun den Startknoten abgearbeitet haben, initialisieren wir in Zeile 8 die Menge  $Besuchte$  mit der Menge, die nur den Startknoten enthält. Die Variable  $Besuchte$  enthält alle Knoten, die bereits besucht worden sind.

Analog zum Algorithmus analyseCSSA() realisieren die Zeilen 9 – 13 die Breitensuche. Die Zeile 9 überprüft zunächst die Abbruchbedingung für die Ausführung des Algorithmus. Die Zeilen 10 – 12 ermitteln die Nachfolger der Knoten, die in  $Kopie_A$  enthalten sind und speichern diese Nachfolger in der Menge  $A$ . Zeile 13 speichert eine Kopie der ermittelten Menge  $A$  in  $Kopie_A$ .

Wurde die Menge  $A$  ermittelt, werden die Knoten aus der Menge  $A$  nacheinander besucht. Dazu entfernen wir in den Zeilen 14 – 16 einen Knoten aus der Menge  $A$  und speichern ihn in der Variablen *aktuell*. Die Schleife, die in Zeile 14 definiert wird, ermöglicht eine Wiederholung bis die Menge  $A$  leer ist.

Um zu ermitteln, ob der aktuelle Knoten den Beginn einer Schleife repräsentiert, rufen wir in Zeile 18 den Algorithmus startZyklus() auf. Repräsentiert der aktuelle Knoten den Beginn einer Schleife liefert der Algorithmus analyseZyklus() den Wert *true* und die Menge  $Besuchte$  vereinigt mit den Knoten, die auf einem Pfad im Schleifenkörper enthalten sind. In Zeile 19 löschen wir aus dieser Menge nun die Knoten aus der Menge  $Besuchte$ . Repräsentiert der aktuelle Knoten nicht den Beginn einer Schleife, liefert der Algorithmus den Wert *false* und die leere Menge. Da der Beginn einer Schleife immer ein  $\phi$ -Knoten ist, können wir für jeden anderen Knotentyp in Zeile 21 den Wert *false* voraussetzen.

Die Bedingung in Zeile 23 überprüft, ob alle Vorgänger des aktuellen Knoten besucht wurden oder ob wir uns am Beginn einer Schleife befinden. Da wir eine Breitensuche realisieren, muss eine dieser beiden Bedingungen erfüllt sein, um den aktuellen Knoten

abzuarbeiten. Ist die Bedingung in Zeile 23 erfüllt, überprüfen wir in Zeile 24, ob wir uns am Beginn einer Schleife befinden und die Schleife bereits abgearbeitet wurde. Ist dies der Fall, wurde der aktuelle Knoten vollständig abgearbeitet und wir können in Zeile 25 den aktuellen Knoten als besucht markieren. Weiterhin ermitteln wir in Zeile 26 den Knoten, der nach der Schleife besucht werden muss und speichern ihn in der Menge  $A$ . Der Knoten, der nach der Schleife besucht werden muss, ist ein Nachfolger des aktuellen Knotens und wurde noch nicht besucht. In Zeile 27 ersetzen wir in der Menge  $Kopie_A$  den aktuellen Knoten durch den Knoten, der nach der Schleife besucht werden muss.

Haben wir den aktuellen Knoten noch nicht besucht, werden die Zeilen 29 – 46 ausgeführt. In Zeile 30 inkrementieren wir den Zähler für Flows, wenn der aktuelle Knoten den Anfangsknoten eines Flows repräsentiert. Repräsentiert der aktuelle Knoten stattdessen das Ende eines Flows, dekrementieren wir den Zähler für Flows in Zeile 32. Außerdem überprüfen wir in Zeile 33, ob der aktuelle Knoten das Ende des Flows repräsentiert, dessen Teilgraph ermittelt werden soll. Ist das Ende dieses Flows erreicht, weisen wir in Zeile 34 der Variablen weiter den Wert *false* zu. Dies hat zu Folge, dass der Algorithmus terminiert.

Haben wir das Ende des Flows, dessen Teilgraph ermittelt werden soll, noch nicht erreicht, wird in Zeile 38 die Menge der Nachfolger des aktuellen Knotens ermittelt. In den Zeilen 38 – 42 wird dann die Menge  $V$  um die Nachfolger und die Menge  $E$  um die Kanten zwischen dem aktuellen Knoten und dessen Nachfolger erweitert. Schließlich markieren wir in Zeile 45 den aktuellen Knoten als besucht, wenn wir uns nicht am Beginn einer Schleife befinden. Der Beginn einer Schleife wird erst dann als besucht markiert, wenn die Schleife vollständig abgearbeitet wurde.

Ist die Bedingung in Zeile 23 nicht erfüllt, befinden wir uns nicht am Beginn einer Schleife und es existiert mindestens ein Vorgänger des aktuellen Knotens, der noch nicht besucht wurde. In diesem Fall warten wir mit der Bearbeitung des Knotens, bis alle Vorgänger besucht worden sind. Deshalb löschen wir in Zeile 49 den aktuellen Knoten aus der Menge  $Kopie_A$ . Der aktuelle Knoten wird erneut besucht, wenn einer der nicht analysierten Vorgängerknoten besucht wurde.

Haben wir das Ende des Flows erreicht, dessen Teilgraph ermittelt werden soll, wird die Zeile 53 ausgeführt. Der Algorithmus terminiert hier, nachdem der ermittelte Teilgraph sowie der Verweis auf den aktuellen Knoten zurückgegeben wurde. Der aktuelle Knoten verweist zu diesem Zeitpunkt auf den Knoten, der das Ende des Flows repräsentiert.

## Terminierung und Korrektheit

Der Algorithmus 22 berechnet den Teilgraphen, der einen Flow im CSSA-Graphen repräsentiert. Der Teilgraph sowie der Knoten, der das Ende des Flows repräsentiert, wird zurückgegeben.

Der Algorithmus terminiert immer, da wir eine Breitensuche realisieren und über Schleifen nur einmal traversieren. Der Algorithmus ist korrekt, da wir jeden Knoten im Flow mit der Breitensuche erreichen. Durch die Breitensuche erreichen wir am Ende des Algorithmus den Knoten, der das Ende eines Flows repräsentiert.

### B.1.2. startZyklus()

Der Algorithmus startZyklus() überprüft, ob ein Knoten im CSSA-Graph den Beginn einer Schleife repräsentiert. Wenn der Knoten den Beginn einer Schleife repräsentiert, liefert er den Wahrheitswert *true* und eine Knotenmenge. Diese Menge enthält alle Knoten, die im CSSA-Graphen vor der Schleife analysiert wurden sowie die Knoten eines Pfades im Schleifenkörper. Repräsentiert der Knoten keinen Beginn einer Schleife, liefert der Algorithmus den Wahrheitswert *false* und die leere Menge. Der Algorithmus wird vom Algorithmus analyseCSSA() sowie vom Algorithmus teilgraphFlow() aufgerufen.

Der Algorithmus startZyklus() ist in Algorithmus 23 definiert. Dem Algorithmus werden beim Aufruf vier Parameter übergeben. Der erste Parameter enthält den CSSA-Graphen, in dem die Schleife potentiell enthalten ist. Der zweite enthält den Knoten im CSSA-Graph, der darauf überprüft werden soll, ob er den Beginn einer Schleife repräsentiert. Der dritte Parameter enthält den Knoten im CSSA-Graph, der aktuell vom Algorithmus besucht wird. Dieser Parameter wird benötigt, da der Algorithmus rekursiv ist. Der vierte Parameter enthält die Knoten, die auf dem Weg zum aktuellen Knoten (dritter Parameter) besucht wurden. Beim initialen Aufruf des Algorithmus wird der zu untersuchende CSSA-Graph als erster Parameter übergeben. Als zweiter Parameter wird der Knoten übergeben, der darauf überprüft werden soll, ob er den Beginn einer Schleife repräsentiert. Als dritten Parameter übergeben wir ebenfalls diesen Knoten, da er als erstes besucht werden muss. Als vierten Parameter übergeben wir die Knoten, die von den Algorithmen analyseCSSA() bzw. teilgraphFlow() bereits besucht wurden.

### Beschreibung

Der Algorithmus traversiert in einer Tiefensuche über den CSSA-Graphen. In der ersten Zeile werden zunächst die Nachfolger des aktuellen Knotens ermittelt. Wenn der aktuelle Knoten keinen Nachfolger besitzt, befinden wir uns am Ende einer Sequenz. Damit können wir in Zeile 3 den Wahrheitswert *false* und die leere Menge zurückliefern. Ansonsten überprüfen wir in Zeile 4, ob der Knoten, der überprüft werden soll, in der Menge der Nachfolger enthalten ist. Ist dies der Fall haben wir einen Zyklus erkannt. Da wir den Anfang eines Zyklus erkennen wollen, müssen wir in Zeile 5 zusätzlich überprüfen, ob alle Vorgänger des Schleifenanfangs besucht wurden. Der Knoten vor dem Schleifenanfang wurde bereits vom Algorithmus analyseCSSA() bzw. teilgraphFlow() besucht. Der ausgezeichnete Knoten am Ende eines Schleifenkörpers haben wir in diesem Algorithmus besucht. Die Menge *Besucht* erweitert um den aktuellen Knoten enthält nun die Knoten, die auf dem Pfad im Schleifenkörper besucht wurden. In Zeile 6 geben wir deshalb den

**Algorithmus 23** startZyklus(CSSA-Graph, start, aktuell, Besucht)

---

```
1: Nachfolger ← nachfolger(aktuell)
2: if Nachfolger =  $\emptyset$  then
3:   return (false,  $\emptyset$ )
4: else if start  $\in$  Nachfolger then
5:   if vorgänger(start)  $\subseteq$  Besucht  $\cup$  {aktuell} then
6:     return (true, Besucht  $\cup$  {aktuell})
7:   else
8:     return (false,  $\emptyset$ )
9:   end if
10: else
11:   Besucht ← Besucht  $\cup$  {aktuell}
12:   for all x  $\in$  Nachfolger do
13:     if x  $\notin$  Besucht then
14:       (ergebnis, Schleife) ← findeZyklus(CSSA-Graph, start, x, Besucht)
15:       if ergebnis = true then
16:         return (true, Schleife)
17:       end if
18:     end if
19:   end for
20:   return (false,  $\emptyset$ )
21: end if
```

---

Wahrheitswert *true* und die Menge *Besucht* zurück. Haben wir einen Zyklus erkannt, befinden uns aber nicht am Beginn einer Schleife, geben wir in Zeile 8 den Wahrheitswert *false* und die leere Menge zurück.

Haben wir nicht das Ende einer Sequenz oder einen Zyklus erkannt, markieren wir in Zeile 11 den aktuellen Knoten als besucht. Danach rufen wir in Zeile 14 für jeden Knoten, der Nachfolger des aktuellen Knotens ist, den Algorithmus `startZyklus()` rekursiv auf. Dabei übergeben wir den betrachteten Nachfolger als dritten Parameter und die aktualisierte Liste *Besucht* als vierten Parameter. Ergibt einer dieser Aufrufe, dass der zweite Parameter den Beginn einer Schleife repräsentiert, wird das Ergebnis dieses Aufrufs zurückgegeben und der Algorithmus terminiert in Zeile 16. Ergeben die Aufrufe für jeden Nachfolger, dass der zweite Parameter keinen Beginn einer Schleife repräsentiert, wird in Zeile 20 der Wahrheitswert *false* und die leere Menge zurückgegeben und der Algorithmus terminiert.

### Terminierung und Korrektheit

Der Algorithmus 23 überprüft, ob ein bestimmter Knoten *k* den Anfang einer Schleife repräsentiert. Dabei wird ein Zyklus mittels Tiefensuche erkannt. Gelangt der Algorithmus bei der Traversierung an einen Knoten, der bereits besucht wurde, wurde ein Zyklus

erkannt. Der Algorithmus überprüft zusätzlich, ob der Knoten  $k$  den Anfang dieses Zyklusses repräsentiert.

Der Algorithmus terminiert, da wir eine Tiefensuche realisieren und die Ausführung abgebrochen wird, sobald ein Zyklus erkannt wurde. Der Algorithmus ist korrekt, da der Anfang einer Schleife im CSSA-Graphen spezielle Eigenschaften hat, anhand derer der Algorithmus entscheidet, ob der Knoten  $k$  den Anfang einer Schleife repräsentiert.

Der Anfangsknoten für eine Schleife im CSSA-Graphen hat genau zwei Vorgängerknoten. Der eine Knoten wird vor der Schleife ausgeführt. Der andere Knoten ist der letzte Knoten im Schleifenkörper. Es gibt genau einen Knoten am Ende des Schleifenkörpers. Sind wir erneut am Schleifenanfang angelangt, wurden also alle Vorgängerknoten vom aufrufenden Algorithmus oder vom Algorithmus selbst besucht. Da der Algorithmus über den CSSA-Graphen mit Hilfe einer Tiefensuche traversiert, können wir für jeden anderen Knoten diese Eigenschaft ausschließen. Somit können wir einen ausgezeichneten Knoten als Anfang eines Zyklusses erkennen, wenn der Knoten sowie alle seine Vorgänger bereits besucht wurden.

### B.1.3. teilgraphSchleife()

Der Algorithmus `teilgraphSchleife()` ermittelt einen Teilgraph, der einen Schleifenkörper repräsentiert. Dabei ist die Kante, die zurück zum Schleifenanfang führt, nicht im Teilgraph enthalten. Der Algorithmus wird vom Algorithmus `analyseCSSA()` aufgerufen. Die mehrfache Iteration über den Schleifenkörper übernimmt der Algorithmus `analyseCSSA()`.

Der Algorithmus `teilgraphSchleife()` ist in Algorithmus 24 definiert. Beim Aufruf werden dem Algorithmus drei Parameter übergeben. Der erste Parameter enthält den CSSA-Graphen, der mindestens eine Schleife enthält. Aus diesem CSSA-Graphen wird der Teilgraph ermittelt. Der zweite Parameter verweist auf den Startknoten der Schleife, für dessen Schleifenkörper der Teilgraph ermittelt werden soll. Der dritte Parameter enthält die Menge der Knoten, die auf einem Pfad im Schleifenkörper vorkommen. Diese Knotenmenge wurde zuvor durch den Algorithmus `startZyklus()` ermittelt.

### Beschreibung

In der ersten Zeile initialisieren wir die Variablen, die den ermittelten Teilgraph speichern. Die Menge  $V$  enthält die Knoten des Teilgraphs. Die Menge  $E$  enthält die Kanten des Teilgraphs. Bei der Initialisierung speichern wir den Anfangsknoten der Schleife in der Menge  $V$ .

Nun ermitteln wir in den Zeilen 2 – 9 den letzten Knoten, der im Schleifenkörper enthalten ist. Da dieser Knoten ein Vorgänger des Startknotens ist, ermitteln wir in Zeile 2 zuerst alle Vorgänger des Startknotens. In der ermittelten Menge sind genau zwei Knoten enthalten. Der letzte Knoten im Schleifenkörper ist der Vorgänger, der auch im

---

**Algorithmus 24** teilgraphSchleife(CSSA-Graph, startknoten, Pfad)

---

```

1:  $V \leftarrow \{\text{startknoten}\}$ ,  $E \leftarrow \emptyset$ 
2:  $\text{Vorgänger} \leftarrow \text{vorgänger}(\text{startknoten})$ 
3: for all  $n \in \text{Vorgänger}$  do
4:   if  $n \in \text{Pfad}$  then
5:      $V \leftarrow V \cup \{n\}$ 
6:   else
7:      $\text{Vorgänger} \leftarrow \text{Vorgänger} \setminus \{n\}$ 
8:   end if
9: end for
10:  $\text{Besuchen} \leftarrow \text{Vorgänger}$ 
11: while  $\text{Besuchen} \neq \emptyset$  do
12:    $\text{aktuell} \leftarrow \text{hole Knoten aus Besuchen}$ 
13:    $\text{Besuchen} \leftarrow \text{Besuchen} \setminus \{\text{aktuell}\}$ 
14:   if  $\text{aktuell.Typ} = \phi$  then
15:      $(\text{schleife}, \text{Knoten}) \leftarrow \text{startZyklus}(\text{CSSA-Graph}, \text{aktuell}, \text{aktuell}, \emptyset)$ 
16:   else
17:      $\text{schleife} \leftarrow \text{false}$ 
18:   end if
19:   if  $\text{not}(\text{schleife} \wedge \text{vorgänger}(\text{aktuell}) \subseteq V)$  then
20:      $\text{Vorgänger} \leftarrow \text{vorgänger}(\text{aktuell})$ 
21:     for all  $n \in \text{Vorgänger}$  do
22:        $V \leftarrow V \cup \{n\}$ 
23:        $E \leftarrow E \cup \{n, \text{aktuell}\}$ 
24:       if  $n = \text{startknoten}$  then
25:          $\text{Vorgänger} \leftarrow \text{Vorgänger} \setminus \{n\}$ 
26:       end if
27:     end for
28:      $\text{Besuchen} \leftarrow \text{Besuchen} \cup \text{Vorgänger}$ 
29:   end if
30: end while
31:  $\text{gefunden} \leftarrow \text{false}$ 
32:  $\text{Besuchen} \leftarrow \{\text{startknoten}\}$ 
33: while  $\text{not}(\text{gefunden})$  do
34:    $k \leftarrow \text{hole Knoten aus Besuchen}$ 
35:    $\text{Besuchen} \leftarrow \text{Besuchen} \setminus \{k\}$ 
36:    $\text{Nachfolger} \leftarrow \text{nachfolger}(k)$ 
37:   for all  $n \in \text{Nachfolger}$  do
38:     if  $n \notin V$  then
39:        $\text{ende} \leftarrow n$ 
40:        $\text{gefunden} \leftarrow \text{true}$ 
41:     end if
42:   end for
43:    $\text{Besuchen} \leftarrow \text{Besuchen} \cup \text{Nachfolger}$ 
44: end while
45: return  $((V, E), \text{ende})$ 

```

---

übergebenen Pfad enthalten ist. Deshalb speichern wir in Zeile 5 diesen Vorgänger in der Menge  $V$ . Den anderen Knoten löschen wir in Zeile 7 aus der Menge der Vorgänger.

Ausgehend von dem letzten Knoten im Schleifenkörper ermitteln wir nun in den Zeilen 11 – 30 die anderen Knoten im Schleifenkörper. Dabei traversieren wir rückwärts über den CSSA-Graphen. In Zeile 19 überprüfen wir, ob der aktuell betrachtete Knoten den Anfangsknoten einer Schleife repräsentiert. Ist dies der Fall, überprüfen wir zusätzlich, ob die Schleife bereits komplett im Teilgraphen enthalten ist. Ist der betrachtete Knoten kein Anfangsknoten einer Schleife, oder ist die Schleife noch nicht komplett im Teilgraphen enthalten, werden die Zeilen 20 – 28 ausgeführt. Hier ermitteln wir für den betrachteten Knoten zunächst alle Vorgänger. Jeden Vorgänger speichern wir dann in den Zeilen 22 und 23 im Teilgraphen. Haben wir mit dem betrachteten Vorgänger den Startknoten erreicht, wird dieser in Zeile 25 aus der Menge *Vorgänger* gelöscht, um ein erneutes Traversieren über die Schleife zu verhindern. Wurden alle Vorgänger des aktuellen Knotens betrachtet, erweitern wir in Zeile 28 die Menge *Besuchen* um die Vorgänger, da auch ihre Vorgänger noch in den Teilgraphen aufgenommen werden müssen. Wurden alle Vorgänger sowie deren Vorgänger usw. betrachtet, haben wir den Teilgraphen für den Schleifenkörper vollständig ermittelt.

Danach ermitteln wir in den Zeilen 33 – 44 noch den Knoten, der nach dem Schleifenkörper ausgeführt wird. Dabei traversieren wir ausgehend vom Startknoten der Schleife vorwärts über den CSSA-Graphen. Finden wir dabei einen Knoten, der nicht im ermittelten Teilgraphen enthalten ist, repräsentiert dieser Knoten den Knoten, der nach dem Schleifenkörper ausgeführt wird. In Zeile 39 speichern wir diesen Knoten in der Variablen *ende*. Danach wird die Traversierung über den CSSA-Graphen abgebrochen. Schließlich geben wir in Zeile 45 den ermittelten Teilgraphen und den Knoten, der nach dem Schleifenkörper ausgeführt wird, zurück und der Algorithmus terminiert.

### Terminierung und Korrektheit

Der Algorithmus 24 ermittelt den Teilgraphen eines CSSA-Graphen, der einen Schleifenkörper repräsentiert. Dazu traversiert der Algorithmus ausgehend vom Startknoten der Schleife rückwärts über den CSSA-Graphen. Schließlich ermittelt der Algorithmus noch den Knoten, der nach dem Schleifenkörper ausgeführt wird. Dazu traversiert der Algorithmus vorwärts über den CSSA-Graphen

Der Algorithmus terminiert immer, da ein Schleifenkörper nur endlich viele Knoten hat und genau ein Knoten nach dem Schleifenkörper ausgeführt wird. Da bei der Rückwärts-traversierung der Startknoten der Schleife nicht mehr betrachtet wird, wird eine erneute Traversierung über den Schleifenkörper verhindert. Auch bei im Schleifenkörper enthaltenen Schleifen wird die mehrmalige Traversierung verhindert. Da es im CSSA-Graphen immer einen ausgezeichneten Knoten gibt, der nach dem Schleifenkörper ausgeführt wird, endet auch die Vorwärtstraversierung immer.

Der Algorithmus ist korrekt, da alle Knoten im Schleifenkörper besucht werden. Im CSSA-Graph besitzt ein Schleifenkörper immer einen ausgezeichneten letzten Knoten.

Ausgehend von diesem Knoten können wir rückwärts über den CSSA-Graphen traversieren und besuchen dabei alle Knoten, die im Schleifenkörper enthalten sind. Da nach dem Schleifenkörper genau ein Knoten ausgeführt wird, können wir ihn durch Vorwärtstraversierung über den CSSA-Graphen ermitteln. Da dieser Knoten nicht im Schleifenkörper enthalten ist, ist er auch nicht im Teilgraphen enthalten. Somit ist ein Knoten, der bei der Vorwärtstraversierung über den CSSA-Graphen erreicht wird und nicht im Teilgraphen enthalten ist, der Knoten, der nach dem Schleifenkörper ausgeführt wird.

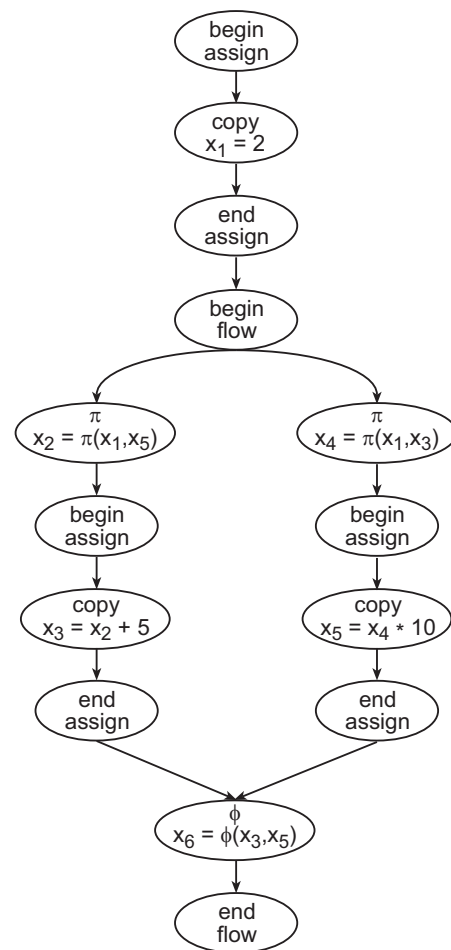
## C. Beispiel

Anhand des folgenden Beispiels wollen wir darstellen, wie in der vorgestellten Analyse die Wertebereiche berechnet. In Abb. C.1 ist ein möglicher Ausschnitt aus einem BPEL-Prozess abgebildet. In Abb. C.1(a) ist der BPEL-Prozess in XML-Notation dargestellt. In Abb. C.1(b) ist der zugehörige CSSA-Graph abgebildet. Zur besseren Lesbarkeit haben wir den CSSA-Graphen angegeben, der die enthaltenen XPath-Ausdrücke nicht in Form des Drei-Adress-Codes repräsentiert.

```

<assign>
  <copy>
    <from> 2 </from>
    <to variable = 'x' />
  </copy>
</assign>
<flow>
  <assign>
    <copy>
      <from> $x + 5 </from>
      <to variable = 'x' />
    </copy>
  </assign>
  <assign>
    <copy>
      <from> $x * 10 </from>
      <to variable = 'x' />
    </copy>
  </assign>
</flow>

```



(a) Der Ausschnitt in XML-Notation.

(b) Der korrespondierende CSSA-Graph.

Abbildung C.1.: Ein Ausschnitt aus einem BPEL-Prozess.

Zunächst berechnet die Analyse den Wertebereich, der bei der ersten Zuweisung entsteht:

$$W(x_1) = \{ ([2, 2], \emptyset) \}$$

Nun wird der Teilgraph des Flows ermittelt und die Analyse wird rekursiv aufgerufen. Die erste Traversierung über den Teilgraphen mit einer Breitensuche ergibt die folgenden Wertebereiche:

$$\begin{aligned} W(x_2) &= \{ ([2, 2], \{x_1\}) \} \\ W(x_4) &= \{ ([2, 2], \{x_1\}) \} \\ W(x_3) &= \{ ([2, 2] \oplus [5, 5], \{x_1, x_2\}) \} = \{ ([7, 7], \{x_1, x_2\}) \} \\ W(x_5) &= \{ ([2, 2] \otimes [10, 10], \{x_1, x_4\}) \} = \{ ([20, 20], \{x_1, x_4\}) \} \\ W(x_6) &= \{ ([7, 7], \{x_1, x_2, x_3\}), ([20, 20], \{x_1, x_4, x_5\}) \} \end{aligned}$$

Der Wertebereich von  $x_2$  berechnet sich aus den Variablen  $x_1$  und  $x_5$ . Da zum Zeitpunkt der ersten Traversierung nur der Wertebereich von  $x_1$  bekannt ist, enthält der Wertebereich von  $x_2$  nur diese Information. Da wir bei der Berechnung des Wertebereichs von  $x_2$  den Wertebereich von  $x_1$  verwendet haben, erweitern wir die Menge der verwendeten Variablen um  $x_1$ . Analog berechnet sich der Wertebereich von  $x_4$ . Der Wertebereich von  $x_3$  berechnet sich nun aus der Addition des ermittelten Wertebereich von  $x_2$  mit der Konstanten 5. Erneut wird die Menge der verwendeten Variablen um die Variable  $x_2$  erweitert. Analog berechnet sich der Wertebereich von  $x_5$ . Schließlich berechnet sich der Wertebereich von  $x_6$  durch Vereinigung der Wertebereiche von  $x_3$  und  $x_5$ , wobei wir die Menge der verwendeten Variablen um diese Variablen erweitern.

Da die Wertebereich im Flow das erste Mal berechnet wurden, wird erneut über den Teilgraphen mit einer Breitensuche traversiert. Hier ergeben sich die folgenden Wertebereiche:

$$\begin{aligned} W(x_2) &= \{ ([2, 2], \{x_1\}), ([20, 20], \{x_1, x_4, x_5\}) \} \\ W(x_4) &= \{ ([2, 2], \{x_1\}), ([7, 7], \{x_1, x_2, x_3\}) \} \\ W(x_3) &= \{ ([2, 2] \oplus [5, 5], \{x_1, x_2\}), ([20, 20] \oplus [5, 5], \{x_1, x_2, x_4, x_5\}) \} \\ &= \{ ([7, 7], \{x_1, x_2\}), ([25, 25], \{x_1, x_2, x_4, x_5\}) \} \\ W(x_5) &= \{ ([2, 2] \otimes [10, 10], \{x_1, x_4\}), ([7, 7] \otimes [10, 10], \{x_1, x_2, x_3, x_4\}) \} \\ &= \{ ([20, 20], \{x_1, x_4\}), ([70, 70], \{x_1, x_2, x_3, x_4\}) \} \\ W(x_6) &= \{ ([7, 7], \{x_1, x_2, x_3\}), ([25, 25], \{x_1, x_2, x_3, x_4, x_5\}), \\ &\quad ([20, 20], \{x_1, x_4, x_5\}), ([70, 70], \{x_1, x_2, x_3, x_4, x_5\}) \} \end{aligned}$$

Die Wertebereiche berechnen sich analog zur ersten Traversierung. Allerdings können wir bei der Berechnung des Wertebereichs von  $x_2$  nun sowohl den Wertebereich von  $x_1$

---

als auch den Wertebereich von  $x_5$  berücksichtigen. Analog enthält der Wertebereich von  $x_4$  sowohl den Wertebereich von  $x_1$  als auch den Wertebereich von  $x_3$ .

Da sich die Wertebereich bei der zweiten Traversierung geändert haben, wird ein drittes Mal über den Teilgraphen mit einer Breitensuche traversiert. Hier ergeben sich die folgenden Wertebereiche:

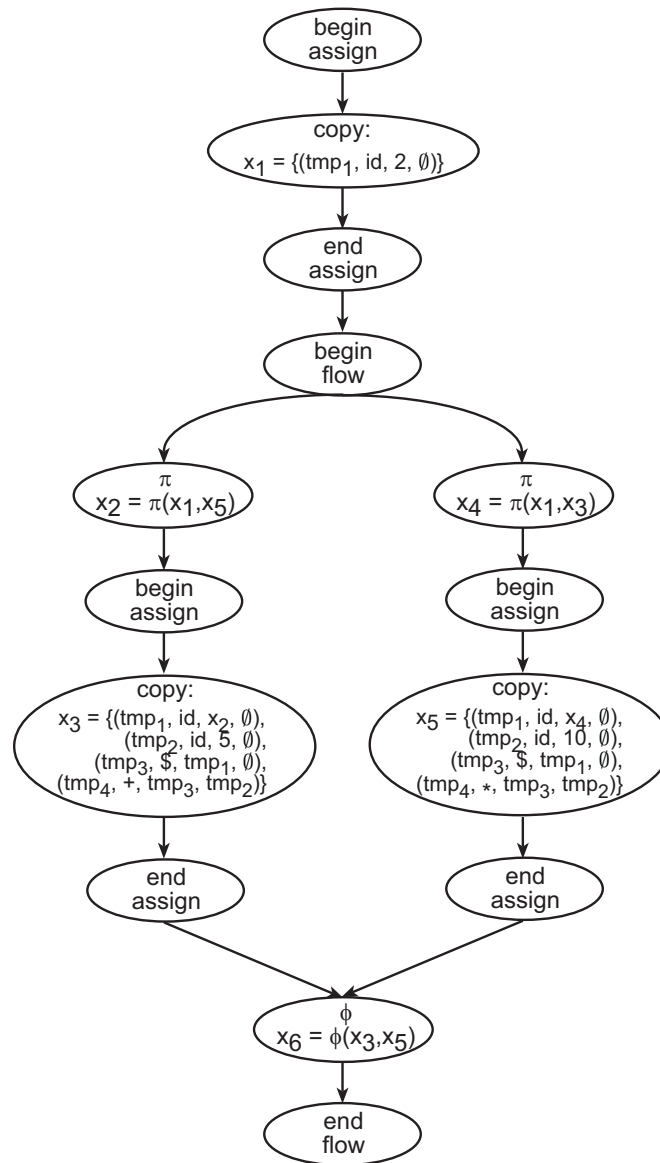
$$\begin{aligned}
W(x_2) &= \{ ([2, 2], \{x_1\}), ([20, 20], \{x_1, x_4, x_5\}) \} \\
W(x_4) &= \{ ([2, 2], \{x_1\}), ([7, 7], \{x_1, x_2, x_3\}) \} \\
W(x_3) &= \{ ([2, 2] \oplus [5, 5], \{x_1, x_2\}), ([20, 20] \oplus [5, 5], \{x_1, x_2, x_4, x_5\}) \} \\
&= \{ ([7, 7], \{x_1, x_2\}), ([25, 25], \{x_1, x_2, x_4, x_5\}) \} \\
W(x_5) &= \{ ([2, 2] \otimes [10, 10], \{x_1, x_4\}), ([7, 7] \otimes [10, 10], \{x_1, x_2, x_3, x_4\}) \} \\
&= \{ ([20, 20], \{x_1, x_4\}), ([70, 70], \{x_1, x_2, x_3, x_4\}) \} \\
W(x_6) &= \{ ([7, 7], \{x_1, x_2, x_3\}), ([25, 25], \{x_1, x_2, x_3, x_4, x_5\}), \\
&\quad ([20, 20], \{x_1, x_4, x_5\}), ([70, 70], \{x_1, x_2, x_3, x_4, x_5\}) \}
\end{aligned}$$

Die Wertebereiche werden analog zu den vorhergehenden Traversierungen berechnet. Allerdings verwenden wir bei der Berechnung des Wertebereichs von  $x_2$  nicht das Element  $([70, 70], \{x_1, x_2, x_3, x_4\})$  aus dem Wertebereich von  $x_5$ , da  $x_2$  bereits in der Menge der besuchten Variablen enthalten ist. Aus diesem Grund ist der Wertebereich von  $x_2$  in der dritten Traversierung gleich dem Wertebereich von  $x_2$  aus der zweiten Traversierung. Analog berechnet sich der Wertebereich von  $x_4$ . Wie wir sehen können, sind die in der dritten Traversierung berechneten Wertebereiche also gleich den Wertebereichen aus der zweiten Traversierung. Damit ist die Analyse des Flows beendet.

Wurde der CSSA-Graph, der den vollständigen BPEL-Prozess repräsentiert, vollständig analysiert, werden die ermittelten Wertebereiche wie folgt zusammengefasst:

$$\begin{aligned}
W(x_2) &= \{ [2, 2] \sqcup_{Int} [20, 20] \} = \{ [2, 20] \} \\
W(x_4) &= \{ [2, 2] \sqcup_{Int} [7, 7] \} = \{ [2, 7] \} \\
W(x_3) &= \{ [7, 7] \sqcup_{Int} [25, 25] \} = \{ [7, 25] \} \\
W(x_5) &= \{ [20, 20] \sqcup_{Int} [70, 70] \} = \{ [20, 70] \} \\
W(x_6) &= \{ [7, 7] \sqcup_{Int} [25, 25] \sqcup_{Int} [20, 20] \sqcup_{Int} [70, 70] \} = \{ [7, 70] \}
\end{aligned}$$

Im Beispiel haben wir für eine bessere Lesbarkeit den CSSA-Graphen verwendet, der die enthaltenen XPath-Ausdrücke nicht in Form des Drei-Adress-Codes repräsentiert. Der Vollständigkeit halber, geben wir für das Beispiel in Abb. C.2 den CSSA-Graphen an, der die XPath-Ausdrücke in Form des Drei-Adress-Codes repräsentiert. Die Berechnung der Wertebereiche im Beispiel bleibt gleich.



**Abbildung C.2.:** Der CSSA-Graph für den Ausschnitt aus einem BPEL-Prozess in Abb. C.1(a) in dem die enthaltenen XPath-Ausdrücke in Form des Drei-Adress-Codes repräsentiert werden.

## Literaturverzeichnis

- [AS85] ALPERN, Bowen ; SCHNEIDER, Fred B.: Defining Liveness. In: *Inf. Process. Lett.* 21 (1985), Nr. 4, S. 181–185
- [ASU99] AHO, Alfred V. ; SETHI, Ravi ; ULLMANN, Jeffrey D.: *Compilerbau*. Oldenburg Verlag, 1999. – englische Originalausgabe: *COMPILERS Principles, Techniques an Tools*. Bell Telephone Laboratories Inc. 1986
- [BFG05] BENEDIKT, Michael ; FAN, Wenfei ; GEERTS, Floris: XPath satisfiability in the presence of DTDs. In: LI, Chen (Hrsg.): *Proceedings of the Twenty-fourth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 13-15, 2005, Baltimore, Maryland, USA*, ACM, 2005. – ISBN 1-59593-062-0, S. 25–36
- [BHL99] BRAY, Tim ; HOLLANDER, Dave ; LAYMAN, Andrew ; W3C (Hrsg.): *Namespaces in XML*. <http://www.w3.org/TR/1999/REC-xml-names-19990114>: W3C, Januar 1999
- [BM04] BIRON, Paul V. ; MALHOTRA, Ashok ; W3C (Hrsg.): *XML Schema Part 2: Datatypes*. <http://www.w3.org/TR/2004/REC-xmlschema-2-20041028>: W3C, Oktober 2004
- [BPSM<sup>+</sup>04] BRAY, Tim ; PAOLI, Jean ; SPERBERG-MCQUEEN, Michael ; MALER, Eve ; YERGEAU, François ; W3C (Hrsg.): *Extensible Markup Language (XML) 1.0*. <http://www.w3.org/TR/2004/REC-xml-20040204>: W3C, Februar 2004
- [CC77] COUSOT, Patrick ; COUSOT, Radhia: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Los Angeles, California : ACM Press, New York, NY, 1977, S. 238–252
- [CCMW01] CHRISTENSEN, Erik ; CURBERA, Francisco ; MEREDITH, Greg ; WEERAWARANA, Sanjiva ; W3C (Hrsg.): *WSDL – Web Services Description Language*. Standard, Version 1.1: W3C, März 2001
- [CD99] CLARK, James ; DEROSE, Steve ; W3C (Hrsg.): *XML Path Language (XPath) 1.0*. <http://www.w3.org/TR/1999/REC-xpath-19991116>: W3C, November 1999
- [Cla99] CLARK, James ; W3C (Hrsg.): *XSL Transformations (XSLT) Version 1.0*. <http://www.w3.org/TR/xslt>: W3C, November 1999

- [Con] CONSORTIUM, Unicode: *The Unicode Standard*.  
<http://www.unicode.org/unicode/standard/standard.html>
- [FUMK04] FOSTER, Howard ; UCHITEL, Sebastián ; MAGEE, Jeff ; KRAMER, Jeff: Compatibility Verification for Web Service Choreography. In: *Proceedings of the IEEE International Conference on Web Services (ICWS'04), June 6-9, 2004, San Diego, California, USA*, IEEE Computer Society, 2004, S. 738–741
- [GLS07] GENEVÈS, Pierre ; LAYAÏDA, Nabil ; SCHMITT, Alan: Efficient static analysis of XML paths and types. In: FERRANTE, Jeanne (Hrsg.) ; MCKINLEY, Kathryn S. (Hrsg.): *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, ACM, 2007. – ISBN 978–1–59593–633–2, S. 342–351
- [God06] GODLINKSY, Artur: *Static analysis to discover communication relevant data flows in BPEL-processes*, Friedrich-Schiller Universität Jena, Diplomarbeit, Oktober 2006
- [GVD04] GENEVÈS, Pierre ; VION-DURY, Jean-Yves: Logic-based XPath optimization. In: MUNSON, Ethan V. (Hrsg.) ; VION-DURY, Jean-Yves (Hrsg.): *Proceedings of the 2004 ACM Symposium on Document Engineering, Milwaukee, Wisconsin, USA, October 28-30, 2004*, ACM, 2004. – ISBN 1–58113–938–1, S. 211–219
- [JE07] JORDAN, Diane ; EVDEMON, John ; OASIS (Hrsg.): *Web Services Business Process Execution Language Version 2.0*.  
<http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html>: OASIS, 2007
- [LMP97] LEE, Jaejin ; MIDKIFF, Samuel P. ; PADUA, David A.: Concurrent Static Single Assignment Form and Constant Propagation for Explicitly Parallel Programs. In: LI, Zhiyuan (Hrsg.) ; YEW, Pen-Chung (Hrsg.) ; CHATTERJEE, Siddhartha (Hrsg.) ; HUANG, Chua-Huang (Hrsg.) ; SADAYAPPAN, P. (Hrsg.) ; SEHR, David C. (Hrsg.): *Languages and Compilers for Parallel Computing, 10th International Workshop, LCPC'97, Minneapolis, Minnesota, USA, August 7-9, 1997, Proceedings* Bd. 1366, Springer, 1997 (Lecture Notes in Computer Science). – ISBN 3–540–64472–5, S. 114–130
- [LMSW08] LOHMANN, Niels ; MASSUTHE, Peter ; STAHL, Christian ; WEINBERG, Daniela: Analyzing Interacting WS-BPEL Processes Using Flexible Model Generation. In: *Data Knowl. Eng.* 64 (2008), Januar, Nr. 1, 38-54.  
<http://dx.doi.org/10.1016/j.datak.2007.06.006>
- [MM06] MARTENS, Axel ; MOSER, Simon: Diagnosing SCA Components Using Wombat. In: DUSTDAR, S. (Hrsg.) ; FIADEIRO, J.L. (Hrsg.) ; SHETH, A.P. (Hrsg.): *Business Process Management, 4th International Conference, BPM*

- 
- 2006, Vienna, Austria, September 5-7, 2006, Proceedings Bd. 4102, Springer, 2006 (Lecture Notes in Computer Science). – ISBN 3-540-38901-6, S. 378–388
- [MMG<sup>+</sup>07] MOSER, Simon ; MARTENS, Axel ; GÖRLACH, Katharina ; AMME, Wolfram ; GODLINSKI, Artur: Advanced Verification of Distributed WS-BPEL Business Processes Incorporating CSSA-based Data Flow Analysis. In: *2007 IEEE International Conference on Services Computing (SCC 2007), 9-13 July 2007, Salt Lake City, Utah, USA*, IEEE Computer Society, 2007, S. 98–105
- [NNH05] NIELSON, Flemming ; NIELSON, Hanne R. ; HANKIN, Chris: *Principles of Program Analysis*. Springer Verlag, 2005
- [OSO07] OSOA ; OPEN SERVICE ORIENTED ARCHITECTURE (Hrsg.): *Service Component Architecture*. <http://www.osoa.org/display/Main/Service+Component+Architecture+Home>: Open Service Oriented Architecture, 2007
- [OVA<sup>+</sup>07] OUYANG, Chun ; VERBEEK, Eric ; AALST, Wil M. P. d. ; BREUTEL, Stephan ; DUMAS, Marlon ; HOFSTEDE, Arthur H. M.: Formal semantics and analysis of control flow in WS-BPEL. In: *Sci. Comput. Program.* 67 (2007), Nr. 2-3, S. 162–198
- [Sch05] SCHMIDT, Karsten: Controllability of Open Workflow Nets. In: DESEL, Jörg (Hrsg.) ; FRANK, Ulrich (Hrsg.) ; Entwicklungsmethoden für Informationssysteme und deren Anwendung (EMISA, RWTH Aachen) (Veranst.): *Enterprise Modelling and Information Systems Architectures* Bd. P-75. Bonn : Köllen Druck+Verlag GmbH, 2005 (Lecture Notes in Informatics (LNI)), S. 236–249
- [Sch07] SCHNURR, Bastian: *Analyse von XPath-Ausdrücken in BPEL-Prozessbeschreibungen*, Universität Stuttgart, Diplomarbeit, Dezember 2007
- [TBMM04] THOMPSON, Henry S. ; BEECH, David ; MALONEY, Murray ; MENDELSON, Noah ; W3C (Hrsg.): *XML Schema Part 1: Structures*. <http://www.w3.org/TR/2004/REC-xmlschema-1-20041028>: W3C, Oktober 2004
- [ZC91] ZIMA, Hans P. ; CHAPMAN, Barbara: *Supercompilers for Parallel and Vector Computers*. ACM Press Frontier Series, Addison-Wesley Verlag, 1991

## Danksagung

Bei der Bearbeitung dieser Diplomarbeit hatte ich das Vergnügen mit verschiedenen Menschen im Dialog zu stehen. In den vielen Diskussionen mit ihnen habe ich immer sehr hilfreiche Hinweise erhalten. An dieser Stelle möchte ich mich bei allen für die großartige Unterstützung bedanken.

Bei der Beantwortung meiner Fragen haben mir Simon Moser und Dr. Wolfram Amme in vielen Diskussionen unermüdlich geholfen. Dafür bedanke ich mich recht herzlich. Mein Dank gilt auch Prof. Dr. Karsten Wolf für seine hilfreichen Hinweise.

Meinem Betreuer Christian Stahl danke ich recht herzlich für sein großes Engagement. Den fleißigen Lesern Daniela Weinberg, Dennis Reinert und Tanja Richter danke ich ebenfalls für ihre hilfreichen Hinweise.

## Erklärung

Hiermit erkläre ich, die vorliegende Arbeit „*Ein Verfahren zur abstrakten Interpretation von XPath-Ausdrücken in WS-BPEL-Prozessen*“ selbstständig und ohne fremde Hilfe verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet zu haben.

Weiterhin erkläre ich hiermit mein Einverständnis, dass die vorliegende Arbeit in der Bibliothek des Instituts für Informatik der Humboldt-Universität zu Berlin ausgestellt werden darf.

Berlin, den 25. März 2008

