Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 44

RESTful BPEL - Erweiterung von BPEL zur Orchestrierung von RESTful Web Services

Markus Fischer

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Frank Leymann
Betreuer/in:	Dipl.-Inf. Florian Haupt
Beginn am:	04.02.2013
Beendet am:	06.08.2013
CR-Nummer:	C.2.4, H.4.1, H.5.4

Kurzfassung

Die Web Service Business Process Execution Language ist eine Sprache zur Modellierung von Geschäftsprozessen, die es ermöglicht Web Services zu orchestrieren, die durch eine WSDL-Schnittstelle definiert sind [TC07]. REST, die Kurzform von „REpresentational State Transfer“, ist ein Architekturstil für Webanwendungen, der bei Einhaltung unter anderem bessere Skalierbarkeit, einfachere System-Architektur und bessere Erweiterbarkeit verspricht [Fieoo].
In dieser Arbeit wird untersucht, welche Möglichkeiten bestehen, BPEL zur Orchestrierung von RESTful Web Services zu verwenden, oder inwiefern BPEL zu diesem Zweck erweitert werden muss. Dazu werden verschiedene Konzepte beschrieben, analysiert und bewertet.
Anschließend wird ein Konzept entworfen und implementiert, das alle notwendigen Anforderungen erfüllt, BPEL zur Orchestrierung von RESTful Web Services zu nutzen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Verzeichnis der Listings

1 Einleitung

Fast jede Firma nutzt Computer um ihr Geschäft zu unterstützen. Sei es zum einfachen Erstellen von Rechnungen oder Kostenvoranschlägen bei sehr kleinen Betrieben, bis zu hoch-digitalisierten Konzernen, bei denen Computer sehr viele Aufgaben übernehmen. Beispielsweise der gesamte Ablauf vom Entgegennehmen einer Bestellung bis zum Zusammenstellen und Verpacken der Lieferung. Nach dem Versand wird die Rechnungsstellung und der Einzug des fälligen Betrages per Lastschriftverfahren zumeist wieder von Computern übernommen, während der menschliche Mitarbeiter nur kontrollierende Tätigkeiten übernimmt.
Wie eine bestimmte Leistung zu erbringen ist, ist in vielen Unternehmen, zur Maximierung der Effizienz, genau definiert. Durch sogenannte Geschäftsprozesse ist beschrieben, welche Aktivitäten in welcher Reihenfolge durchgeführt werden müssen (siehe Kapitel 2.1) um ein bestimmtes Ziel zu Erreichen. Ein Geschäftsprozess kann, als Aktivität, Teil eines übergeordneten Geschäftsprozesses sein, sodass die gesamten Leistungen eines Unternehmens in einem einzigen Geschäftsprozess mit vielen Teil-Prozessen dargestellt werden können. Geschäftsprozesse modellieren demnach, wie der Prozess zur Erbringung einer Leistung, in der Realität durchgeführt werden muss. Teile dieser Prozesse können dabei, wie oben bereits beschrieben, allein von Computern ausgeführt werden. Sie werden Workflows genannt, siehe Kapitel 2.2, und werden von sogenannten Workflow-Management-Systemen ausgeführt. Workflow-Management-Systeme unterstützen zum Einen die Definition von Workflows, zum Anderen liefern sie auch die Funktionalität, Instanzen dieser Workflows tatsächlich auszuführen, siehe Kapitel 2.3.
Damit ein Workflow ausgeführt werden kann, muss er in erster Linie auch von einem System verstanden werden. Dazu wurden Beschreibungssprachen definiert, die es ermöglichen, Workflows mit allen benötigten Informationen zu modellieren.
Große Firmen begannen bereits in den frühen 90er-Jahren damit, ihre eigenen Workflow-Systeme zu entwickeln, während Anbieter von Anwendungssoftware erst mitte der 90er-Jahre begannen, ebenfalls eigene Workflow-Systeme zu entwickeln. Bis zu diesem Zeitpunkt hatte sich kein System als Standard durchgesetzt.
1993 wurde die Workflow Management Coalition[1] gegründet, mit dem Ziel Standards zu schaffen, die das Workflow Management vereinheitlichen, sodass auch verschiedene Workflow-Management-Systeme interoperabel sind. Das dient dem Zweck Anwendern Sicherheit zu geben, dass ihre Investitionen in Workflow-Management geschützt sind. Zu den veröffentlichten Standards gehört beispielsweise das Workflow Referenz Modell, welches allgemein ein Workflow-Management-System, sowie seine Komponenten und deren Schnitt-

[1] http://www.wfmc.org/

stellen untereinander, beschreibt (nähere dazu Informationen finden sich unter [Coa]).
Mit dem Aufkommen von Web Services als plattformunabhängige Softwareanwendungen, deren Implementierung hinter einer WSDL-Schnittstelle verborgen ist, veröffentlichten IBM und Microsoft im Jahr 2002 die Sprache **BPEL** zur Beschreibung von Geschäftsprozessen. Die Aktivitäten der beschriebenen Geschäftsprozesse werden dabei ausschließlich durch Web Services(mit WSDL 1.1-Schnittstellenbeschreibung) implementiert, siehe BPEL - 2.4. Seit dem Jahr 2007 ist BPEL ein offizieller Standard der OASIS[2].
Bereits im Jahr 2000 veröffentlichte Roy Fielding seine Dissertation mit dem Titel „Architectural Styles and the Design of Network-based Software Architectures"[Fie00], in der er unter Anderem den Architektur Stil REST ableitet, siehe 2.5. REST ist ein Architekturstil für Webanwendungen, der sich an der Architektur des WWW orientiert. Im folgenden werden diese Web Anwendungen als REST Services oder RESTful Web Services bezeichnet.
Der OASIS Standard WS-BPEL 2.0 ist allerdings nur bedingt geeignet, solche Webanwendungen zu nutzen. Siehe dazu Kapitel 3. Ziel dieser Arbeit ist es, diesen Umstand zu ändern.

Ziel der Bachelorarbeit

Die Ziele der Bachelorarbeit sind folgende:

- **Erhebung, Analyse, Vergleich und Bewertung verschiedener Ansätze, REST Services für BPEL nutzbar zu machen.**
- **Entwurf eines Konzepts zur Erweiterung von BPEL, um RESTful Web Services nutzbar zu machen.**
- **Prototypische Implementierung des entwickelten Konzepts.**

Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Grundlagen: In Kapitel 2 werden die grundlegenden Begriffe erläutert: Geschäftsprozesse, Workflows, Workflow-Management, BPEL und REST.

Kapitel 3 – Orchestrierung von REST Services mit BPEL: Dieses Kapitel beschäftigt sich mit bestehenden Möglichkeiten, REST Services mit BPEL zu orchestrieren und deren Evaluation.

Kapitel 4 – Konzept zur Erweiterung von BPEL um RESTful Web Services zu orchestrieren: In diesem Kapitel wird das Konzept zur Erweiterung von BPEL definiert und besprochen.

[2] www.oasis-open.org

Kapitel 5 – BPEL4REST - Implementierung: Im Kapitel „BPEL4REST - Implementierung“ wird die tatsächliche Realisierung von BPEL4REST dokumentiert.

Kapitel 6 – Verwandte Arbeiten: Dieses Kapitel zeigt verwandte Arbeiten, beschreibt deren Ansätze und grenzt sie von dieser Arbeit ab.

Kapitel 7 – Zusammenfassung und Ausblick: In diesem Kapitel werden die Ergebnisse dieser Arbeit zusammengefasst und bewertet. Anschließend wird noch ein Ausblick gegeben.

2 Grundlagen

Im diesem Kapitel werden verschiedene Begriffe erklärt, die zum Verständnis dieser Arbeit wichtig sind. Zunächst werden Geschäftsprozesse, Workflows und Workflow-Management-Systeme kurz erläutert, danach wird erklärt auf BPEL und REST eingegangen.

2.1 Geschäftsprozesse

Für den Begriff des Geschäftsprozesses gibt es viele verschiedene Definitionen. Das **GABLER WIRTSCHAFTSLEXIKON** definiert:

„ Folge von Wertschöpfungsaktivitäten (Wertschöpfung) mit einem oder mehreren Inputs und einem Kundennutzen stiftenden Output. Geschäftsprozesse können auf verschiedenen Aggregationsebenen betrachtet werden, z.B. für die Gesamtunternehmung, einzelne Sparten- oder Funktionalbereiche. Der Geschäftsprozess ist zentraler Betrachtungsgegenstand des Business Process Reengineering. “ [GWSS]

Thomas Davenport definiert einen Geschäftsprozess als strukturierte Menge von Aktivitäten mit dem Ziel eine bestimmtes Ergebnis für einen bestimmten Kunden oder Markt herzustellen. Er folgert weiter, dass ein Geschäftsprozess eine bestimmte Ordnung dieser Aktivitäten in Zeit und Raum darstellt, mit einem Anfang und einem Ende sowie klar definierten Ein- und Ausgaben. [Dav93]

Hammer und Champy definieren den Geschäftsprozess als eine Sammlung von Aktivitäten, die eine oder mehrere verschiedene Eingaben nimmt und daraus eine Ausgabe herstellt, die einen Wert für den Kunden darstellt. [HC93]

Wenn man diese drei Definitionen „übereinanderlegt“ und die Gemeinsamkeiten extrahiert, kann man folgende Definition ableiten:

Ein Geschäftsprozess ist eine vorgegebene Menge von Aktivitäten, die in einer bestimmten Reihenfolge ausgeführt werden, um aus einer definierten Eingabe eine definierte Ausgabe zu erzeugen, welche für einen bestimmten Empfänger einen höheren Wert hat als die Eingabe.

In der Realität sind diese Eingaben beispielsweise Rohstoffe, die in einzelnen Aktivitäten schrittweise verarbeitet werden, bis ein Endprodukt entstanden ist, das für jemanden einen höheren Wert darstellt. Als Beispiel sei hier bspw. die Autoproduktion genannt, bei der, grob gesagt, aus Erzen ein Auto entsteht, das für den Kunden einen höheren Wert darstellt, als das zugrunde liegende Erz. Der Kunde muss nicht zwingend ein Endverbraucher sein. Auch ein interner Kunde, beispielsweise eine andere Abteilung, kann Empfänger der Endprodukte von Geschäftsprozessen sein.

2.2 Workflow

Leymann und Roller schreiben in Ihrem Buch „Production Workflow“, dass Prozessmodelle die Struktur eines Geschäftsprozesses der realen Welt darstellen. Sie werden nicht zwangsläufig von einem Computer ausgeführt.
Teile dieser Prozessmodelle können von Computern ausgeführt werden, andere Teile wiederum nicht. Die von Computern ausführbaren Teile bennenen sie mit dem Begriff „workflow model“, siehe Abbildung 2.1. Workflows sind demnach, durch Computer ausführbare, Geschäftsprozesse, und somit eine echte Untermenge der Geschäftsprozesse.
Ein Workflow hat drei Dimensionen [LRoo]:

1. **Prozesslogik:** Aus der Prozesslogik geht hervor, **was** für die Durchführung eines Workflows erforderlich ist. Das heisst: Welche Aktivitäten müssen in welcher Sequenz ausgeführt werden.
2. **Organisation:** Die Organisations-Dimension beschreibt, **wer** welche Aktivität durchführen muss. Das heisst: Welche Rolle, Abteilung, Person muss welche Aktivität ausführen.
3. **IT-Infrastruktur:** Die IT-Infrastruktur-Dimension zeigt **welche** IT-Ressourcen von welcher Aktivität benötigt werden.

[LRoo]

2.3 Workflow-Management-Systeme

Workflows sind Geschäftsprozesse, die von Computern ausgeführt werden. Um Workflows auszuführen, müssen sie zunächst in all ihren Dimensionen definiert werden, um anschließend von einem geeigneten System ausgeführt zu werden. Zu diesem Zweck kommen „Workflow-Management-Systeme (WFMS)“ zum Einsatz. Workfow-Management-Systeme haben die Aufgabe, Funktionalität bereitzustellten Workflows zu modellieren, zu verwalten, Instanzen von Workflows zu erstellen und diese auszuführen. Dazu verfügen sie über Editoren, aber auch über Komponenten, die in der Lage sind, alle Anwendungen anzusteuern, die ein Workflow zur Ausführung benötigt.
Leymann und Roller beschreiben den Aufbau eines Workflow-Management-Systems aus vier Hauptkomponenten [LRoo]:

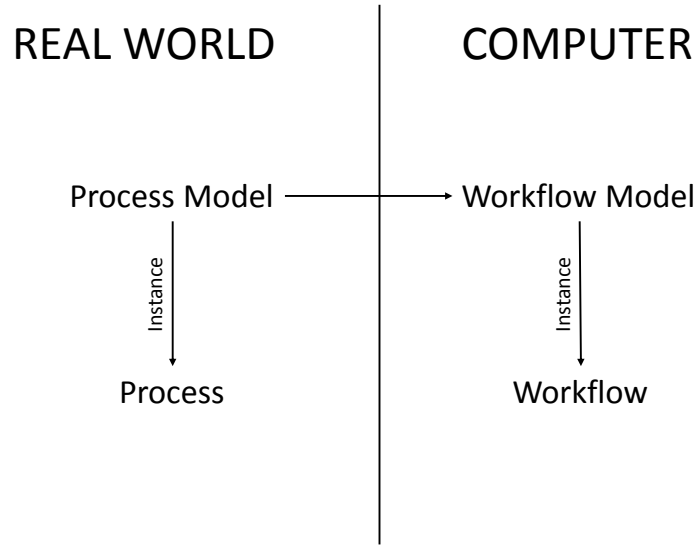

Abbildung 2.1: Prozesse und Workflows, nach [LR00, Abbildung 1.2 Processes and Workflows]

1. **Metamodell:** Das Metamodell definiert allgemein die Konstrukte und Funktionen, die von einem Workflow Management System unterstützt werden. Dazu gehört beispielsweise die Struktur von Prozessmodellen, sowie Operationen, die auf Prozessinstanzen ausgeführt werden können. Ein wichtiger Teil des Metamodells ist die Sprache, in der Workflows definiert werden.
2. **Buildtime:** Die Buildtime stellt Funktionalität bereit, mit der Anwender Workflows in allen drei Dimensionen definieren können.
3. **Runtime:** Die Runtime ist dafür verantwortlich, Workflows nach den Regeln des Metamodells auszuführen. Dazu gehört beispielsweise das Erstellen, Verwalten und Ausführen von Prozessinstanzen.
4. **Datenbank:** Aufgabe der Datenbank ist es, sämtliche Daten zu speichern, die von der Runtime und der Buildtime verwaltet werden. Sie enthält also zum einen Daten der Buildtime, wie beispielsweise Prozessmodelle, aber auch Daten der Runtime, wie beispielsweise Prozessinstanzen.

Siehe Abbildung 2.2 zum Aufbau einer WFMS nach [LR00].

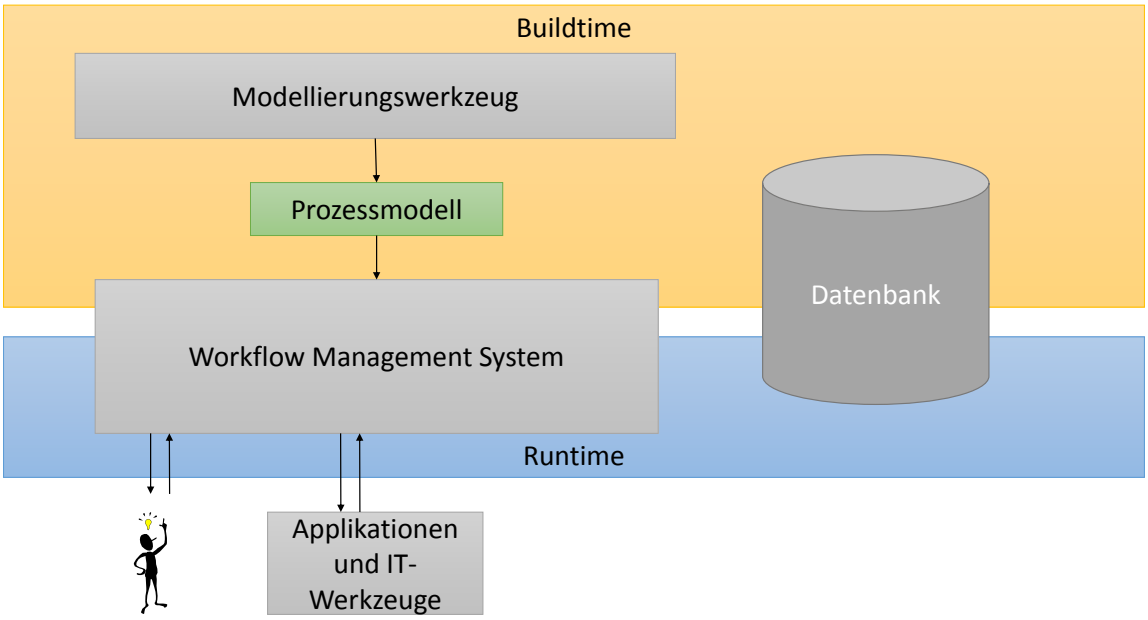

Abbildung 2.2: Komponenten eines Workflow Management Systems, nach [LRoo, Abbildung 3.1]

2.4 BPEL

Das Akronym BPEL ist die Kurzform für WS-BPEL und steht für Web Services Business Process Execution Language. Im weiteren wird nur das Akronym BPEL verwendet. Die aktuelle Version von BPEL ist die Version 2.0, die am 11.04.2007 als OASIS Standard [1] veröffentlich wurde. BPEL ist eine, auf XML basierende, Sprache, mit der es möglich ist, das Verhalten von Geschäftsprozessen zu modellieren.
BPEL ist damit eine Sprache um Workflows zu modellieren und auszuführen, die insofern eingeschränkt sind, dass sie ausschließlich Web Services zur Interaktion mit der Außenwelt nutzen. Im weiteren wird dennoch der Begriff „Geschäftsprozess“ aufgrund seiner Nähe zum Begriff „Business Process“ verwendet.
BPEL wurde entworfen, um Web Services zu orchestrieren, die Aktivitäten implementieren, die von Geschäftsprozessen genutzt werden. Diese Web Services werden dabei durch den W3C Standard WSDL 1.1[CCMW01] beschrieben, so dass BPEL selbst keine Details der Implementierung oder der technischen Realisierung des Nachrichtentransports kennen muss. Ebenfalls sieht BPEL vor, dass durch BPEL definierte Prozesse ebenfalls über WSDL 1.1 angeboten werden und somit wiederum in anderen BPEL-Prozessen genutzt werden können. BPEL selbst kennt 2 verschiedene Arten von Prozessen. Zum einen gibt es ausführbare

[1] https://www.oasis-open.org/

Listing 2.1 Syntax der Definition eines Partnerlink-Typs in WSDL

```
<wsdl:definitions name="NCName" targetNamespace="anyURI" ...>
   ...
   <plnk:partnerLinkType name="NCName">
      <plnk:role name="NCName" portType="QName" />
      <plnk:role name="NCName" portType="QName" />?
   </plnk:partnerLinkType>
   ...
</wsdl:definitions>
```

Prozesse, die darauf ausgelegt sind, von einer BPEL-Engine, auch nebenläufig, ausgeführt zu werden und die deshalb detaillierte Informationen enthalten müssen. zum anderen gibt es Abstrakte Prozesse, die eine beschreibende Rolle haben und deshalb oft technische, zur Ausführung notwendige, Details verbergen. Eine BPEL-Engine ist eine Software, die das Ausführen von BPEL-Prozess-Instanzen ermöglicht. Dazu müssen die BPEL-Prozesse in die Engine gebracht werden (deployed). Ein Beispiel für eine BPEL-Engine ist Apache ODE [2].
Zur Modellierung von Geschäftsprozessen stehen in BPEL verschiedene Konzepte zur Verfügung.
Es gibt in BPEL die Möglichkeit Nachrichten **synchron oder asynchron** mit Web Services zu kommunizieren. Empfangene Nachrichten können innerhalb von BPEL in **Variablen** gespeichert werden. Da BPEL eine XML-basierte Sprache ist und auch das interne Datenmodell auf XML aufbaut, sind auch die Variablen auf XML-Strukturen beschränkt. So gespeicherte Nachrichten können innerhalb von BPEL mittels **XPath 1.1** manipuliert werden. Das Versenden, Empfangen und Verarbeiten von Nachrichten geschieht in BPEL durch **Aktivitäten(activities)**.
Notwendig um mit Web Services zu kommunizieren sind auch sogenannte **Partnerlinks**. Partnerlinks stellen die Verbindung zu Web Services dar. Dazu müssen in Web Services in ihrer WSDL Schnittstelle sogenannte **partnerLinkTypes** definieren. In diesen Partnerlink-Typen können dann Rollen definiert werden, die einem bestimmten portType zugeordnet sind (Listing 2.1). Die Rollen, die von Web Services angeboten werden, können dann einem Partnerlink zugewiesen und der Web Service kann in BPEL aufgerufen werden(Listing 2.2).
Als weiteres Konstrukt stehen in BPEL noch sogenannte **Scopes** zur Verfügung. Scopes sind in BPEL Aktivitäten, die unter anderem ihre eigene Variablen und Partnerlinks haben können. Sie eignen sich dazu, Aktivitäten logisch zu gruppieren, um sie beispielsweise transaktional auszuführen[LLN11]. Die Spezifikation von BPEL ist hier zu finden: [TC07].

2.5 Was ist REST?

REST, kurz für Representational State Transfer ist ein Architekturstil, der von Roy Fielding im Jahr 2000 im Rahmen seiner Dissertation definiert wurde[Fie00]. Fielding nennt REST einen hybriden Stil, der von verschiedenen netzwerk-basierten Architekturstilen abgeleitet und

[2] ode.apache.org

Listing 2.2 Syntax der Definition eines Partnerlinks in BPEL

```
<partnerLinks>
   <partnerLink name="NCName"
      partnerLinkType="QName"
      myRole="NCName"?
      partnerRole="NCName"?
      initializePartnerRole="yes|no"? />+
</partnerLinks>
```

mit zusätzlichen Einschränkungen, die eine einheitliche Verbindungsschnittstelle (uniform connector interface) definieren, kombiniert ist [Fieoo].
Um REST zu definieren, beschreibt Fielding den Null-Stil, der nur eine leere Menge von Einschränkungen enthält. Anschließend erweitert er diese Menge inkrementell um sechs Einschränkungen, bis REST letztendlich definiert ist.

Die erste Einschränkung ist die Beschränkung von REST auf die **Client-Server-Architektur**. Das Kernprinzip der Client-Server-Architektur ist die Trennung von Zuständigkeiten. Durch das Trennen der Zuständigkeit für Benutzerschnittstellen und der Zuständigkeit für Datenhaltung wird die Portabilität der Benutzerschnittstelle über verschiedene Plattformen verbessert und die Skalierbarkeit verbessert sich durch einfachere Serverkomponenten. Der vielleicht wichtigste Aspekt im Web, nach Fielding, ist jedoch die Tatsache, dass verschiedene Komponenten unabhängig voneinander entwickelt werden können.

Die zweite Einschränkung, die Fielding seinem neuen Architekturstil auferlegt, ist das Gebot der **Statuslosigkeit** in der Client-Server-Interaktion. Die Kommunikation zwischen Client und Server muss immer statuslos sein. Jede Anfrage vom Client an den Server muss alle notwendigen Informationen beinhalten, damit der Server die Anfrage versteht und verarbeiten kann.
Diese Einschränkung beeinflusst laut Fielding die Eigenschaften von REST in Bezug auf Sichtbarkeit, Zuverlässigkeit und Skalierbarkeit.
Die Sichtbarkeit wird laut Fielding dadurch verbessert, dass ein Monitoring-System auf nicht mehr schauen muss, als das Datum von Anfragen, um die Natur einer Anfrage zu bestimmen.
Die Zuverlässigkeit verbessert sich, weil teilweise Ausfälle leichter repariert werden können.
Die Skalierbarkeit verbessert sich deshalb, weil kein Status zwischen verschiedenen Anfragen gespeichert werden muss.
Nachteil der statuslosen Kommunikation ist laut Fielding eine verschlechterte Netz-Performanz, weil der Nachrichten-Overhead zunimmt. Zusätzlich verliert der Server an Kontrolle über die Konsistenz des Applikationsverhaltens, weil der Applikationsstatus allein beim Client gehalten wird. Daher ist die gesamte Applikation von der semantisch korrekten Implementierung des Clients abhängig.

Fielding definiert das **Layered System** als dritte Einschränkung. Diese Einschränkung ist eine Weiterentwicklung der **Client-Server** Einschränkung. Das Layered System soll laut Fielding die Internet-Skalierbarkeit verbessern. Der Layered System Stil erlaubt es einer

Architektur aus hierarchischen Schichten aufgebaut zu werden indem das Verhalten einzelner Komponenten eingeschränkt wird, so dass sie nicht über direkt angrenzende Schichten „sehen“ können.

Die vierte Einschränkung, betrifft **Caching**. Das Ziel ist es, dadurch die Netzwerk-Effizienz zu verbessern. Notwendig dazu ist, dass Daten innerhalb von Antworten für bestimmte Anfragen implizit oder explizit als **cacheable** oder **non-chacheable** gekennzeichnet werden. Sollte eine Antwort cacheable sein, kann der Cache des Clients diese Antwort für spätere, äquivalente Anfragen wiederverwenden. Die so gewonnen Vorteile sind, das manche Interaktionen zwischen Client und Server teilweise oder komplett entfallen können. Das verbessert die Effizienz, Skalierbarkeit und die vom Nutzer empfundene Performanz dadurch, dass die durchschnittliche Latenz einer Serie von Interaktionen zwischen Client und Server reduziert wird. Der Nachteil ist, dass Caching die Zuverlässigkeit verschlechtern kann, wenn sich veraltete Daten innerhalb des Caches signifikant von den Daten unterscheiden, die von einer bestimmten Anfrage direkt an den Server, geliefert würden.

Die fünfte Einschränkung ist das Gebot der einheitlichen Schnittstelle. Fielding beschreibt die einheitliche Schnittstelle als „zentrales Feature“, mit dem sich der REST Architekturstil am deutlichsten von anderen netzwerk-basierten Stilen unterscheidet. Laut Fielding vereinfacht sich dadurch die gesamte System-Architektur und die Sichtbarkeit von Interaktionen verbessert sich. Nachteil der einheitlichen Schnittstelle ist eine Verschlechterung der Effizienz, da Informationen in standardisierter Form übertragen werden anstelle von applikations-spezifischer Form. Laut Fielding ist die REST Schnittstelle dafür entworfen, bei großen Hypermedia Daten-Transfers effizient zu sein, um für das Internet optimiert zu sein. Das Resultat aber ist eine Schnittstelle, die nicht optimal ist für andere Formen architekturieller Interaktion.[Fie00, 5.1.5]
Um eine einheitliche Schnittstelle für REST zu beschreiben legt Fielding vier weitere Schnittstellen-Einschränkungen fest: Identifikation von Ressourcen, Manipulation von Ressourcen durch Repräsentationen, selbst-beschreibende Nachrichten und Hypermedia as the engine of applicationen state (HATEOAS). Diese Einschränkungen werden im folgenden Abschnitt erklärt.

Eine grundlegende Abstraktionseinheit in REST, ist die **Ressource**. Innerhalb von REST kann eine Ressource alles darstellen, was einen Namen bekommen könnte. Beispielsweise ein Dokument, ein Service, eine Zusammenfassung anderer Ressourcen etc.
Wichtig ist, dass jede Ressource eindeutig referenziert werden kann. Das entspricht der Schnittstellen-Einschränkung **Identifikation von Ressourcen**. Um dies zu gewährleisten, werden **Uniform Resource Identifier**[3] , kurz URI, verwendet, die einen globalen Namensraum bilden.
Fielding beschreibt Ressourcen als Mapping auf eine Menge von Entitäten. Um Ressourcen zu nutzen, brauchen die Ressourcen auch **Repräsentationen** ihrer selbst, die von Komponenten einer REST-Anwendung, bspw. ein Client, genutzt werden können. Dabei zeigt

[3]http://www.ietf.org/rfc/rfc3986.txt

eine Repräsentation, die vom Server verwaltet wird, den aktuellen Stand einer Ressource an. Um Ressourcen zu verändern, also zu manipulieren, verändert ein Client die Repräsentation nach seinen Wünschen und schickt sie zurück an den Server. Dieser führt dann eine Aktualisierung seiner Ressource gemäß der, vom Client manipulierten, empfangenen Repräsentation durch. Das beschreibt die Schnittstellen-Einschränkung **Manipulation von Ressourcen durch Repräsentationen**. Eine Repräsentation besteht hauptsächlich aus Daten und Meta-Daten, die die Daten beschreiben. Als Beispiel hierzu sei ein konkret existierendes Auto genannt, das über einen Webshop verkauft wird. Die Ressource ist in diesem Beispiel das Auto, das durch verschiedene Repräsentationen dargestellt wird. Diese Repräsentationen können vom Bild des Autos bis zu detaillierten Informationen über das Auto im XML-Format reichen.
Vor allem textuelle Repräsentationen, wie bspw. im XML-Format, können auch Referenzen auf weitere Ressourcen enthalten. Als Beispiel sei hier eine Online-Auktions-Plattform genannt, bei der beispielsweise Repräsentationen von einzelnen Artikeln nicht nur Informationen zum Artikel selbst, sondern auch Referenzen zu anderen Ressourcen enthalten, die mit der repräsentierten Ressource in Verbindung stehen. Beispielsweise den Verkäufer des Artikels, andere Artikel die ähnlich sind oder auch andere Artikel des Verkäufers. Diese Referenzen und Verknüpfungen von Objekten beschreiben **HATEOAS**. Die **selbstbeschreibenden** Nachrichten resultieren in erster Linie aus der Beschränkung auf **statuslose** Interaktion zwischen Client und Server. Das bedeutet, wie weiter oben zum Teil beschrieben, dass Nachrichten alle notwendigen Informationen enthalten müssen, damit der entgegennehmende Server die Nachricht versteht und verarbeiten kann. Dazu werden auch **Standardmethoden** verwandt, die von jeder Ressource verstanden werden.
Im HTTP Umfeld sind diese Standardmethoden vor allem GET, PUT, POST und DELETE, die dem CRUD-Architekturstil ähneln.
REST-Systeme sind also vor allem eine Menge von eindeutig identifizierbaren Einheiten, die alle dieselben Methoden verstehen, und jeweils durch unterschiedliche Repräsentationen dargestellt und manipuliert werden können. REST-konforme Anwendungen sind in diesem Sinne Anwendungen, die sich den Regeln des REST Architekturstils unterwerfen.

Als sechste Einschränkung nennt Fielding noch **Code-On-Demand**. Diese Einschränkung ist aber **optional**. Mit Code-On-Demand ist gemeint, dass die Client-Funktionalität erweitert werden kann, indem Clients Code, in Form von Applets oder Skripten, herunterladen und ausführen können. Das vereinfacht laut Fielding die Clients, indem die Anzahl der vorher zu implementierenden Features verkleinert wird.

3 Orchestrierung von REST Services mit BPEL

Unter der Orchestrierung von REST Services mit BPEL wird im folgenden Kapitel die Nutzung von REST Services in BPEL-Prozessen verstanden. Nachfolgend wird „Orchestrierung von REST Services mit BPEL“ durch „RESTful BPEL“ abgekürzt.
Als REST Services gelten alle Services, die ihre Dienste, über eine REST Schnittstelle, mittels HTTP-Protokoll und HTTP-Verben zur Verfügung stellen.

3.1 Anforderungen

Um RESTful BPEL zu praktizieren müssen bestimmte Anforderungen erfüllt sein. Ein zentrales Prinzip der REST-Architektur, ist das Prinzip der Einheitlichen Schnittstelle[Fie00, 5.1.5 Uniform Interface]. Wichtige Aspekte der einheitlichen Schnittstelle sind die Identifikation von Ressourcen, Manipulation von Ressourcen durch Repräsentationen, selbst-beschreibende Nachrichten und HATEOAS. Folgende Anforderungen werden an RESTful BPEL gestellt:

1. **Die URI von HTTP-Anfragen muss zur Laufzeit eines BPEL-Prozesses manipulierbar sein.**

   Begründung: Die Identifikation von Ressourcen erfolgt im HTTP Umfeld über **Uniform Resource Identifier**[1]. Laut HATEOAS können Ressourcen auch auf andere Ressourcen verweisen. Aus diesem Grund kann die URI bestimmter HTTP-Anfragen erst zur Laufzeit eines BPEL-Prozesses bekannt sein.

2. **Die Content-Negotiation muss für alle REST-Aufrufe in vollem Umfang unterstützt werden.**

   Begründung: REST Ressourcen können über viele verschiedene Repräsentationen verfügen. Es ist möglich, dass ein BPEL-Prozess mehrere verschiedene Repräsentationen der selben Ressource benötigt, beispielsweise, wenn zunächst eine formale Beschreibung und anschließend eine technische Beschreibung benötigt wird. Es muss also möglich sein bei gleichbleibendem Endpunkt einer Ressource den angeforderten Datentyp zu ändern.

[1] http://www.ietf.org/rfc/rfc3986.txt

3. **HTTP-Header müssen für alle REST-Aufrufe eines BPEL-Prozesses manipulierbar sein.**

   Begründung: REST beschränkt die Kommunikation zwischen seinen Komponenten auf selbst-beschreibende Nachrichten. Um Nachrichten mit Meta-Daten zu beschreiben, verwendet HTTP Header-Felder. Weiterhin werden von den Header-Feldern wichtige Daten übermittelt. Zum Beispiel Angaben zum Caching oder zur Authentifizierung.

3.2 Analyse bestehender Ansätze zur Orchestrierung von REST Services mit BPEL

In diesem Kapitel werden verschiedene Ansätze, RESTful BPEL zu praktizieren, besprochen. Möglichkeiten, RESful BPEL zu praktizieren, werden dabei zunächst aufgezeigt und beschrieben. Anschließend werden sie anhand der Anforderungen, definiert in 3.1, untersucht und danach wird ein Fazit über die Eignung, auf diese Weise RESTful BPEL zu praktizieren, gezogen. Als **geeignet** wird eine Möglichkeit nur angesehen, wenn alle Anforderungen aus 3.1 erfüllt sind.

3.2.1 RESTful BPEL mit WSDL 1.1

Quelle:[CCMW01]

Beschreibung

Die WSDL 1.1 Spezifikation beinhaltet ein Binding für HTTP 1.1 für die Verben GET und POST. Es können aber alle HTTP-Verben verwendet werden. Ursprünglich waren diese beiden HTTP-Bindings gedacht, um die Interaktion zwischen Webseiten und Internet-Browsern zu beschreiben. Die Syntax ist in Listing 3.1 zu sehen.

Beschreibung der wichtigsten Elemente aus Listing 3.1

http:adress: Das **location** Attribut im **http:address** Element spezifiziert die Basis-URI für den port.

http:binding: Das **http:binding** Element zeigt an, dass dieses Binding das HTTP Protokoll nutzt, das Attribut **verb** beschreibt, welches HTTP-Verb genutzt werden soll. Die WSDL 1.1 Spezifikation [CCMW01] sagt aus, dass GET und POST übliche Werte sind. Es können weiterhin aber auch andere Werte verwendet werden, sodass alle HTTP-Verben mittels WSDL 1.1 nutzbar sind.

Listing 3.1 Syntax des HTTP-Bindings in WSDL 1.1, Auszug aus der Spezifikation

```
<definitions .... >
    <binding .... >
        <http:binding verb="nmtoken"/>
        <operation .... >
           <http:operation location="uri"/>
           <input .... >
               <-- mime elements -->
           </input>
           <output .... >
               <-- mime elements -->
           </output>
        </operation>
    </binding>
    <port .... >
        <http:address location="uri"/>
    </port>
</definitions>
```

http:operation: Im **http:operation** Element kann über das **location** Attribut eine relative URI für die Operation spezifiziert werden. Die URI aus den **location** Attributen von **http:address** und **http:operation** werden zusammengefasst und bilden die komplette URI.

http:urlEncoded: Das **http:urlEncoded** Element zeigt an, dass alle Nachrichtenteile (message parts), mittels standard URI-Encoding-Regeln(name1=value&name2=value...) in die URI der HTTP-Anfrage kodiert werden. Informationen zum URI-Encoding: [uri99, 17.13/ 17.13.4/ B2.2]

http:urlReplacement: Das **http:urlReplacement** Element zeigt an, dass alle Nachrichtenteile (message parts), mittels einem Replacement-Algorithmus in die URI der HTTP-Anfrage kodiert werden. Dieser Algorithmus funktioniert wie nachfolgend beschrieben.

Funktionsweise des Replacement-Algorithmus:

- Die relative URI des location Attributs von http:operation wird nach einer Menge von Suchmustern abgesucht, **bevor** der Wert des location Attributs von http:operation mit dem Wert des location Attributs von http:address kombiniert wird.
- Es gibt ein Suchmuster je Nachrichtenteil. Der Suchstring ist der Name des Nachrichtenteils, der durch „(“ und „)“ eingeklammert ist.
- Für jeden Treffer, wird der Wert des korrespondierenden Nachrichtenteils in die URI an den Ort des Treffers geschrieben. Die Suche nach Treffern findet statt **bevor** Werte ersetzt werden (ersetzte Werte lösen keine zusätzlichen Treffer aus).
- **Nachrichtenteile dürfen KEINE, sich wiederholende, Werte haben.**

Analyse

1. Die URI wird in WSDL 1.1 jeweils durch das **location**-Attribut der **http:address** und **http:operation** gebildet. Mittels des **http:urlReplacement** Elements und des **http:urlEncoding** Elements lassen sich Teile der relativen URI des location Attributs aus dem **http:operation** durch Werte des Inputs ersetzen. Allerdings werden dazu immer alle Nachrichtenteile verwendet. Es ist also nicht möglich Teile der URI ersetzen zu lassen und dennoch eine Nachricht an den Server zu senden. Weiterhin ist die Unterstützung von **http:urlReplacement** und **http:urlEncoding** nur bedingt gegeben. Oracle bzw. damals noch Sun Microsystems erlaubt sowohl urlReplacement als auch urlEncoding nur bei HTTP-GET[Mico8]. Um allerdings Links zu folgen, die evtl. zu einem anderen Host gehören, muss entweder server-seitigen redirects gefolgt werden oder während der Laufzeit des BPEL-Prozesses, bestehende WSDL-Dateien mittels „Endpunkt-Injektion“ verändert werden, da Partnerlinks in BPEL zur Deploy-Zeit definiert sein müssen. Ansonsten müssen alle benötigten Endpunkte schon vor der tatsächlichen Ausführung bekannt sein.
2. Die WSDL 1.1 Spezifikation beinhaltet ein MIME Binding [CCMW01, 5. MIME Binding], mit welchem der MIME Type von Input und Output angegeben werden kann. Diese Informationen können zur Content-Negotiation verwendet werden. Das MIME Binding wird auch im **binding** Element definiert.
3. Der Zugriff auf HTTP-Header ist mit WSDL 1.1 nicht möglich.

Fazit

Um mit WSDL 1.1 RESTful BPEL zu praktizieren ist es nötig, innerhalb der WSDL-Datei für jede einzelne Ressource(also für jede URI), pro HTTP-Methode die genutzt werden soll, bindings zu definieren. Grund dafür ist die starke Einschränkung von http:urlReplacement und http:urlEncoding. Sollten also auf einer Ressource 5 HTTP-Verben (GET, HEAD, PUT, POST, DELETE) genutzt werden, müssen 5 Bindings, und damit auch 5 portTypes und 5 ports definiert werden. Bei RESTful Web Services, die dem HATEOAS-Prinzip folgen, müssten zudem noch zur Laufzeit Pfade geändert werden. Weiterhin ist die Content-Negotiation sehr stark eingeschränkt und der Zugriff auf HTTP-Header gar nicht möglich. Damit ist WSDL 1.1 für RESTful BPEL **nicht geeignet**.

3.2.2 RESTful BPEL mit WSDL 1.1 Extension for REST

Quelle:[wsd]

Beschreibung

Um die Nutzung von REST innerhalb der ODE zu verbessern, wird innerhalb der ODE eine Erweiterung für WSDL 1.1 angeboten. Dafür wurde WSDL 1.1 um vier spezifische Punkte erweitert bzw. verändert.

Erweiterungen an WSDL 1.1 durch Apache ODE

1. Ein HTTP-Verb pro Operation.
2. Eindeutiges URI Template für alle Operationen.
3. Zugriff auf HTTP-Header.
4. Fehlerbehandlung.

Ein Verp pro Operation: Anstatt das HTTP-Verb im http:binding Element zu definieren wird das Verb eine Ebene tiefer im http:operation Element definiert. Dies führt dazu, dass pro Ressource nur noch ein binding und somit auch nur ein portType und ein port benötigt wird (Listing 3.2).

Listing 3.2 Beispiel der Syntax der WSDL 1.1 Extension for REST [wsd]

```
<definitions ...xmlns:odex="http://www.apache.org/ode/type/extension/http"/>
    <!-- many wsdl elements are ommited to highlight the interesting part -->
    <binding name="blogBinding" type="blogPortType">
        <operation name="GET">
            <odex:binding verb="GET" />
        </operation>
        <operation name="DELETE">
            <odex:binding verb="DELETE"/>
        </operation>
    </binding>
</definitions>
```

Eindeutiges URI Template für alle Operationen: ODE definiert 3 Probleme, die mit der Adressierung des WSDL 1.1 HTTP-Bindings auftreten:

1. Das **location** Attribut der http:operation ist für die ODE erst sichtbar, wenn der Service aufgerufen wird.
2. **http:urlReplacement** wird nur für HTTP-Get unterstützt.
3. **http:urlReplacement** setzt voraus, dass alle Teile der Input-message in der **http:operation location** vorkommen. Mögliche ids müssen also Teil der Nachricht sein. Im HTTP-Body dürfen diese Daten teilweise nicht vorkommen.

Apache ODE WSD L1.1 Extension for REST löst diese Probleme dadurch, dass das **http:operation** Element aus- oder leergelassen werden kann. Die komplette Ressourcen-URI wird im **http:address** Element gesetzt. Weiterhin ist das **http:urlReplacement** gelockert. Es müssen nicht alle Teile der Input-message Teil des URI-Templates sein. Ein Teil kann für die URI, ein anderer für den Message-Body verwendet werden. Ein Beispiel dafür ist in Listing reflst:wsdl11extbsp zu sehen.
Im **http:operation** Element wird die URI im Attribut **location** mit einem Platzhalter definiert: http://blog.org/post/ Durch die Lockerung von urlReplacement kann dieser Platzhalter vor der Ausführung ersetzt werden. Das HTTP-Verb wird in diesem Beispiel erst im Element **odex:binding** angegeben, was, wie zu Beginn des Kapitels beschrieben, dazu führt, dass weniger bindings, portTypes und ports definiert werden müssen.

Listing 3.3 Beispiel der Syntax der WSDL 1.1 Extension for REST mit http:urlReplacement [wsd]

```
<definitions ... xmlns:odex="http://www.apache.org/ode/type/extension/http"/>
   <service name="blogService">
       <port name="blogPort" binding="blogPortType">
           <!-- here is the full URI template, using curly brackets -->
           <http:address location="http://blog.org/post/{post_id}"/>
       </port>
    </service>
    <binding name="blogBinding" type="blogPortType">
       <operation name="PUT">
           <odex:binding verb="PUT" />
           <!-- location attribute intentionally blank -->
           <http:operation location=""/>
           <input>
              <http:urlReplacement/>
              <!-- an additional part can be mapped to the request body even if
                  urlReplacement is used-->
              <mime:content type="text/xml" part="post_content"/>
           </input>
           <output/>
       </operation>
    </binding>
</definitions>
```

Zugriff auf HTTP-Header Da HTTP sehr viel Informationen über Header austauscht, erlaubt Apache ODE WSDL 1.1 Extension for REST sowohl Request-, als auch Response-Header in den Input bzw. Output Elementen zu definieren. Siehe dazu Listing 3.4

Fehlerbehandlung

Für diverse Status-Codes werden von der ODE automatisch Fehler ausgelöst:

- **3xx** Redirections(erst ab 100 redirects)
- **401** Unauthorized
- **408** Request Timeout

Listing 3.4 Beispiel der Syntax der WSDL 1.1 Extension for REST mit Headern [wsd]

```
<definitions ... xmlns:odex="http://www.apache.org/ode/type/extension/http"/>
    <binding name="blogBinding" type="blogPortType">
        <operation name="PUT">
            <odex:binding verb="PUT" />
            <http:operation location=""/>
            <input>
                <http:urlReplacement/>
                <mime:content type="text/xml" part="post_content"/>
                <!-- set a standard request header from a part -->
                <odex:header name="Authorization" part="credentials_part"/>
                <!-- set a custom request header with a static value -->
                <odex:header name="MyCustomHeader" value="ode@apache.org" />
            </input>
            <output>
                <mime:content type="text/xml" part="post_content"/>
                <!-- set 1 response header to a part -->
                <odex:header name="Age" part="age_part"/>
            </output>
        </operation>
    </binding>
</definitions>
```

- **503** Service Unavailable
- **504** Gateway Timeout

Für weitere Status-codes werden Fehler ausgelöst wenn:

- Die operation mindestens einen Fehler im abstrakten Teil und ein Fehler-Binding hat,
- der Content-Type (der Response) Header ein XML-Dokument beschreibt,
- der Response-Body nicht leer ist.

Eine Liste der weiteren Status-Codes ist zu finden unter: [wsd, Fault Support]. Ein Beispiel für die Definition eines Fehlers ist in Listing 3.5 zu sehen.
Definition der HTTP-Status-Codes: [htt99, 10 Status Code Definitions].

Analyse

1. Mit Hilfe der **Apache ODE WSDL 1.1 Extension for REST** ist es möglich mittels **http:urlReplacement** sehr dynamisch auf URI-Veränderungen zu reagieren. Allerdings ist es auch bei dieser Erweiterung nicht möglich die komplette URI zur Laufzeit zu ändern, so dass ein anderer REST Service oder Host schon zur Deploy-Zeit bekannt sein muss.
2. Die Content-Negotiation ist unverändert zu WSDL 1.1.

Listing 3.5 Beispiel der Definition von Fehlern in WSDL 1.1 for REST[wsd]

```
<definitions ... xmlns:odex="http://www.apache.org/ode/type/extension/http"/>
   <portType name="BlogPortType">
       <operation name="PUT">
          <input message="..."/>
          <output message="..."/>
          <fault name="UpdateFault" message="tns:UpdateFault"/>
       </operation>
    </portType>
    <binding name="blogBinding" type="blogPortType">
       <operation name="PUT">
            <odex:binding verb="PUT" />
            <http:operation location=""/>
            <input> ... </input>
            <output> ... </output>
            <!-- fault binding -->
            <fault name="UpdateException">
               <!-- name attribute is optional if there is only one fault for this operation
                   -->
               <!-- <odex:fault name="UpdateFault"/> -->
               <odex:fault/>
            </fault>
       </operation>
    </binding>
</definitions>
```

3. Die Unterstützung für HTTP-Header ist gegeben. Probleme die sich hierbei aber stellen, sind zum Beispiel Authorization-Header. Für Basic Auth ist es recht einfach die richtigen Credentials „von Hand“ zu berechnen. Bei aufwändigeren Verfahren wie bspw. DIGEST-Authentifizierung ist die Extension nicht ausreichend.

Fazit

Die **Apache ODE WSDL 1.1 Extension for REST** erhöht den Komfort, für RESTful BPEL deutlich im Vergleich zum Standard WSDL 1.1.
Hauptkritikpunkt dieser WSDL-Erweiterung ist, dass der Nachteil eines statischen Hosts von WSDL 1.1 erhalten bleibt.
Ein weiterer Kritikpunkt ist die Unterstützung für Header, die zwar weitgehend ausreicht, aber nicht vollständig ist (bspw. bei der Authentifizierung). Somit ist die Apache ODE WSDL 1.1 Extension for REST für RESTful BPEL **nicht geeignet**.

3.2.3 RESTful BPEL mit WSDL 2.0

WSDL 2.0 bietet deutlich mehr Funktionalität zur Verwendung von HTTP als WSDL 1.1. Die wichtigsten Unterschiede in Bezug auf RESTful BPEL werden im Folgenden besprochen.
Quellen: [wsd06a][wsd06b][CCMW01]

Beschreibung

Wesentliche Unterschiede zwischen WSDL 1.1 und WSDL 2.0.:

- **HTTP-Methoden:** Die Angabe des HTTP-Verbs erfolgt in WSDL 2.0 auf der Ebene des **operation** Elements. Dies verringert die Anzahl der benötigten Interfaces(PortTypes), Endpoints(Ports) und Bindings signifikant im Vergleich zur Vorgängerversion WSDL 1.1.
- **Fehlerbehandlung:** WSDL 2.0 erlaubt es Fehler im **binding** mittels **fault** Element zu definieren, die zu HTTP-Status-Codes korrespondieren und ausgelöst werden, wenn dieser Status-Code zurückgegeben wird.
- **HTTP-Header-Unterstützung:** WSDL 2.0 erlaubt es HTTP-Header zu setzen. Dies kann sowohl im Element **fault** direkt im Binding, aber auch in den Elementen **input** und **output** in der **operation** deklariert werden.
- **Authentifizierungs-Unterstützung:** WSDL 2.0 unterstützt HTTP-Authentifizierung für die Authentifizierungsarten **basic** und **digest access**. Dies kann in **endpoints** mit den beiden optionalen Attributen **whttp:authenticationType** und **whttp:authenticationType** realisiert werden.
- **Content-Negotiation:** Die Content-Negotiation mittels MIME-Binding wurde in WSDL 2.0 entfernt und ersetzt durch drei neue Attribute im **operation** Element: **whttp:inputSerialization**, **whttp:outputSerialization** und **whttp:faultSerialization**.
- **URI:** Die URI wird im **endpoint** Element im Attribut **address** deklariert und mit dem **whttp:location** Attribut aus dem **operation** kombiniert. Die bei WSDL 1.1 (und ODE Extension) vorhandenen Relemente **http:urlEncoded** und **http:urlReplacement** gibt es in WSDL 2.0 nicht mehr. Dafür erlaubt es WSDL 2.0, dass zusätzliche Verarbeitung des Werts von **whttp:location** durch die Input-Serialisierung stattfinden kann, bevor der Wert mit der Endpunkt-Adresse kombiniert wird.
- Weiterhin bietet WSDL 2.0 noch Unterstützung für TransferCoding und Cookies.

Für eine komplette Darstellung der Syntax des HTTP-Bindings von WSDL 2.0 siehe Listing 3.6.

Analyse

Neue Funktionalität von WSDL 2.0 wird in vielen Fällen bereits von der **Apache ODE WSDL 1.1 Extension for REST** angeboten.

1. Wie bei WSDL 1.1 und Apache ODE WSDL 1.1 Extension for REST, ist es auch mit WSDL 2.0, nicht möglich die URI eines genutzten REST Service zur Laufzeit zu ändern, ohne die neue URI bereits zur Deploy-Zeit zu kennen.

Listing 3.6 Übersicht der Syntax des HTTP-Bindings von WSDL 2.0 [wsd06b]

```
<description>
  <binding name="xs:NCName" interface="xs:QName"?
           type="http://www.w3.org/2006/01/wsdl/http"
           whttp:methodDefault="xs:string"?
           whttp:queryParameterSeparatorDefault="xs:string"?
           whttp:cookies="xs:boolean"?
           whttp:transferCodingDefault="xs:string"? >
    <documentation />?
     <fault ref="xs:QName"
           whttp:code="union of xs:int, xs:token"?
           whttp:transferCoding="xs:string"? >
       <documentation />*
       <whttp:header name="xs:string" type="xs:QName"
                     required="xs:boolean"? >
         <documentation />*
       </whttp:header>*
       [ <feature /> | <property /> ]*
     </fault>*
     <operation ref="xs:QName"
                whttp:location="xs:anyURI"?
                whttp:method="xs:string"?
                whttp:inputSerialization="xs:string"?
                whttp:outputSerialization="xs:string"?
                whttp:faultSerialization="xs:string"?
                whttp:transferCodingDefault="xs:string"? >
          <documentation />*
       <input messageLabel="xs:NCName"?
              whttp:transferCoding="xs:string? >
         <documentation />*
         <whttp:header ... />*
         [ <feature /> | <property /> ]*
       </input>*
       <output messageLabel="xs:NCName"?
               whttp:transferCoding="xs:string? >
         <documentation />*
         <whttp:header ... />*
         [ <feature /> | <property /> ]*
       </output>*
       <infault ref="xs:QName"
                messageLabel="xs:NCName"? >
         <documentation />*
         [ <feature /> | <property /> ]*
       </infault>*
       <outfault ref="xs:QName"
                 messageLabel="xs:NCName"? >
         <documentation />*
         [ <feature /> | <property /> ]*
       </outfault>*
       [ <feature /> | <property /> ]*
     </operation>*
     [ <feature /> | <property /> ]*
  </binding>
  <service>
    <endpoint name="xs:NCName" binding="xs:QName" address="xs:anyURI"?
              whttp:authenticationType="xs:token"?
              whttp:authenticationRealm="xs:string"? >
      <documentation />*
      [ <feature /> | <property /> ]*
    </endpoint>
    [ <feature /> | <property /> ]*
  </service>
</description>
```

2. In WSDL 2.0 kann durch neue Attribute des operation Elements, **whttp:inputSerialization** und **whttp:outputSerialization**, die Input- und Output-Serialisierung festgelegt werden. Diese Informationen können auch zur Content-Negotiation verwandt werden, da der Wert dieser Attribute, mit kleinen Ausnahmen, die gleichen Werte annehmen kann wie der Accept-Header aus HTTP [htt99, 14.1][wsd06a, 6.4.3]. Durch die Definition im operation Element müssen aber auch hier 2 operations definiert werden, um verschiedene Medientypen der selben Ressource abzurufen.
3. Die Unterstützung für HTTP-Header ist gegeben. Das Problem der Authentifizierung bei der Apache ODE WSDL 1.1 Extension for REST ist bei WSDL 2.0 zumindest für die Authentifizierungsarten Basic und Digest Access gelöst.

Fazit

Abgesehen vom WSDL-bedingten Problem, dass alle REST Service Endpunkte schon vor der Laufzeit einer BPEL-Prozess-Instanz bekannt sein müssen und dass für jeden Medientyp eine oeration notwendig ist, wäre WSDL 2.0 eine gute Möglichkeit um RESTful BPEL zu praktizieren. Leider ist nicht abzusehen, ob WSDL 2.0 jemals mit BPEL eingesetzt werden kann.

In der WS-BPEL FAQ [bpeb] auf der OASIS-Seite befassen sich die Fragen 6 und 7 zum Teil mit WSDL 2.0.
Frage 6 befasst sich damit, welche Web Services in einem BPEL-Prozess genutzt werden können, und ob BPEL auch mit RESTful Web Services arbeiten könnte. Als Antwort darauf steht, dass BPEL jeden Web Service nutzen kann, der einen WSDL 1.1 Vertrag anbietet. BPEL sei aber nicht dafür entworfen mit RESTful Web Services zu arbeiten, weil diese kein WSDL nutzen.
Frage 7 beschäftigt sich damit, welche Auswirkungen es hätte, wenn auch WSDL 2.0 Web Services von einem BPEL-Prozess genutzt werden müssen. Die Antwort darauf ist knapp formuliert, dass BPEL nicht dazu gedacht ist, WSDL 2.0 zu nutzen.
Auf der OASIS-Seite des BPEL-Komitees steht unter dem Stichwort Übersicht, dass der Zweck dieses Komitees darin bestand, an der 2002 veröffentlichte **Busines Process Execution Language for Web Services (BPEL4WS)** weiterzuarbeiten. Die Mitglieder des Komitees beschlossen, dass sie ihre Aufgabe 2007 abgeschlossen haben.[bpea]
Demnach wurden die Arbeiten am BPEL Standard 2007 abgeschlossen und es ist nicht zu Erwarten, dass es bald eine neue Version von BPEL geben wird, die WSDL 2.0 unterstützt. Demnach ist WSDL 2.0 ist also ebenfalls für RESTful BPEL **nicht geeignet**.

3.2.4 RESTful BPEL Wrapped in SOAP

Auch ohne Veränderungen an WSDL 1.1 oder WS-BPEL 2.0 wäre es möglich **RESTful BPEL** zu praktizieren. Der Kunstbegriff „RESTful BPEL Wrapped in SOAP“ ist die Idee, einen ganzen RESTful Web Service hinter WSDL 1.1 zu „verstecken“ und ihn somit für BPEL

nutzbar zu machen. Dazu müssten weder Änderungen an WSDL 1.1, noch an WS-BPEL 2.0 durchgeführt werden. Aufgrund der Tatsache, dass bei jedem Versuch REST in SOAP zu wrappen, neue Software implementiert werden muss, ist es hier nicht nötig, anhand der Anforderungen aus 3.1 zu analysieren.

Beschreibung

Es gibt verschiedene Ansätze einen RESTful Web Service hinter einem WSDL 1.1 Web Service zu verstecken. Die Ausarbeitung und Evaluation einer solchen Aufgabenstellung ist allerdings sehr umfangreich, weshalb hier nur ein einfacher Ansatz erläutert wird.

Eine Möglichkeit REST in WSDL zu wrappen ist, einen WSDL Web Service mit nur einer Operation zu implementieren, der als Middleware zwischen BPEL und REST fungiert. Dazu nutzt eine BPEL-Prozess-Instanz diese eine Operation und übermittelt als Nachricht alle Angaben, die nötig sind eine korrekte HTTP-Anfrage durchzuführen. Die Middleware ist in diesem Szenario auch dafür zuständig, die Antworten von aufgerufenen REST Services wieder umzuwandeln und an die BPEL-Prozess-Instanz zurückzusenden. Auf diese Weise könnten aus BPEL heraus beliebige REST Services angesteuert werden. Um beliebige REST Services zu nutzen muss ein sehr generisches Datenmodell gewählt werden. Eine einfache Variante wäre es , komplette HTTP-Nachrichten, also incl. Meta-Daten zu nutzen. Die Implementierung der Middleware beschränkt sich dann auf HTTP-Funktionalität und einer Transformation von HTTP-Nachrichten in ein XML-Format.

Fazit

Ein Nachteil dieser Variante ist, dass durch die strenge Typisierung von BPEL-Variablen sehr viel zusätzliche Datenmanipulation vorgenommen werden muss. Beispielsweise müsste für eine GET-Anfrage, die sowohl text/xml als auch application/xml akzeptiert, zunächst überprüft werden, ob die Anfrage erfolgreich war. Dazu müsste mittels XPath ermittelt werden, ob der Status-Code der Antwort dem erwarteten Status-Code entspricht. Danach müsste, ebenfalls mit XPath der Content-Type der Antwort überprüft werden, um anschließend durch ein if-Statement zu unterscheiden, in welche Variable der EntityBody der HTTP-Antwort geschrieben wird. Für eine solche Anfrage bräuchte es also mindestens drei Assign-Aktivitäten, sowie eine if-Aktivität. Bei einem BPEL-Prozess mit mehreren REST-Aufrufen steigt so die Komplexität stark an.
Weiterhin wird für diese Variante eine zusätzliche Software benötigt, die Ressourcen verbraucht und zusätzlichen Konfigurationsaufwand bedeutet.
Diese Variante ist demnach in technischer Hinsicht zwar für RESTful BPEL geeignet, der Komfort für den Anwender wird aber, durch die erhöhte Komplexität und damit verbundenen Aufwand, stark beeinträchtigt. Diese Möglichkeit ist also für RESTful BPEL **nicht geeignet**.

4 Konzept zur Erweiterung von BPEL um RESTful Web Services zu orchestrieren

In Kapitel 3 wurden verschiedene Ansätze für RESTful BPEL vorgestellt und anhand der Kriterien aus 3.1 analysiert und bewertet. Von den vorgestellten Ansätzen erfüllt keiner die gestellten Anforderungen, die nachfolgend nochmal aufgelistet sind:

1. **Die URI von HTTP-Anfragen muss zur Laufzeit eines BPEL-Prozesses manipulierbar sein.**
2. **Die Content-Negotiation muss für alle REST-Aufrufe in vollem Umfang unterstützt werden.**
3. **HTTP-Header müssen für alle REST-Aufrufe eines BPEL-Prozesses manipulierbar sein.**

Um dennoch „RESTful" mit BPEL zu Arbeiten wird in Kapitel 4 ein Konzept beschrieben, wie eine Erweiterung für BPEL aussehen kann, die sich die BPEL eigene Möglichkeit der Erweiterung, durch eine **extensionActivity** [TC07, 10.9], verfügbar macht.
Zunächst werden einige allgemeine Informationen zur Erweiterungen gegeben, danach werden Konventionen definiert, die in Kapitel **4.3 - XML-Erweiterung von BPEL** zur Anwendung kommen. Zuletzt wird das Konzept ebenfalls anhand obiger Anforderungen evaluiert.

4.1 Allgemeines

Allgemeine Informationen zur Erweiterung:

- BPEL kann erweitert werden durch Definition neuer Aktivitäten in einem BPEL-Prozess, die nicht Teil der Spezifikation sind. BPEL ermöglicht diese Erweiterung durch das Element **<extensionActivity>**, welches Teil der Spezifiktaion ist [TC07, 10.9].
- Für jedes angebotene HTTP-Verb wird eine ExtensionActivity definiert. Die angebotenen HTTP-Verben sind GET, HEAD, PUT, POST und DELETE.
- Um global HTTP-Meta-Daten (HTTP-Header) für mehrere REST-Aufrufe zu definieren, wird ein manipulierbarer Kontext eingeführt.

- Dieser Kontext wird durch die Bereitstellung eines XML-Schemas für BPEL-Variablen realisiert.
- Der grundlegende Aufbau der ErweiterungsAktivitäten (ExtensionActivities) wird für alle HTTP-Verben gleich sein, mit der Ausnahme von bestimmten Elementen, die nicht konform sind zur Semantik bestimmter Verben. Diese Elemente werden deshalb nicht bei allen Verben vorkommen; beispielsweise Elemente zur Beschreibung des HTTP-message-body bei GET-Anfragen.
- Zur Ausführung von ErweiterungsAktivitäten muss ebenfalls die ausführende BPEL-Engine erweiterbar sein.

4.2 Konventionen

Folgenden Konventionen gelten für den Entwurf dieses Konzepts:

In HTTP unterscheiden sich Message-Body und Entity-Body nur, wenn ein Transfer-Coding angewandt wurde [htt99, 4.3]. In den folgenden Kapiteln werden die Begriffe Message-Body und Entity-Body gleichbedeutend für die Präsenz einer Nachrichten-Entität(also die Repräsentation einer Ressource), innerhalb einer HTTP-Nachricht, verwandt.

Variablenreferenz: In manche Attribute kann ein Verweis auf eine Variable des ausführenden BPEL-Prozesses geschrieben werden. Ist dies der Fall, wird dies durch einschließende $-Zeichen gekennzeichnet. Beispiel: var=„$myBPELVariable$“

Elemente und Attribute werden mit folgenden Multiplizitäten angegeben:

optional: Ein optionales Element kann nicht oder einmal vorhanden sein.

verpflichtend: Ein verpflichtendes Element muss genau einmal vorhanden sein.

beliebig: Ein beliebiges Element kann nicht oder beliebig oft vorhanden sein.

Beschreibung von einzelnen Elementen und Attributen:

Bei der Erklärung von **Attributen** steht nach dem Doppelpunkt zunächst der Typ, dann die Multiplizität, nach dem Semikolon folgt die Beschreibung.

Bei der Erklärung von **Elementen** steht nach dem Doppelpunkt die Multiplizität, nach dem Semikolon folgt die Beschreibung.

4.3 XML-Erweiterung von BPEL

Das Konzept für die Erweiterung wird im Folgenden anhand des XML-Elements der REST-Extension für das HTTP-Verb POST erläutert.

4.3.1 Generelle Syntax

Listing 4.1 Generelle Syntax der Erweiterung anhand von POST

```
<post host="" path="">
       <context ref="">
              ...
       <context/>
       <requestParameters>
              ...
       </requestParameters>
       <responseParameters>
              ...
       </responseParameters>
</post>
```

Erläuterung der einzelnen Elemente der generellen Syntax der Erweiterung, Listing 4.1.

- **post:** verpflichtend; Das **<post>** Element des XML-Beispiels ist das Wurzel-Element der Erweiterung. Es zeigt in erster Linie an, welches HTTP-Verb ausgeführt werden soll. Für die anderen vier Verben heißt dieses Element analog zum Beispiel **<get>**, **<put>**, **<delete>** oder **<head>**. Desweiteren ist das Wurzel-Element für den „Endpunkt“ des REST-Services zuständig, also für die URI. Diese wird aus den Attributen berechnet.

  Attribute:

  - **host:** host ist vom Typ XSD String und kann entweder eine URI als String enthalten, oder aber einen Verweis auf eine Variable, ebenfalls vom Typ XSD String, die ihrerseits eine URI enthält. Wichtig ist, dass dieser String als letztes Zeichen keinen Schrägstrich enthält.
  - **path:** path ebenfalls vom Typ XSD String und enthält entweder eine Variablenreferenz oder einen relativen Pfad. Dieser relative Pfad, der ggf. in der Variable des Typs XSD String steht, darf keinen anführenden Schrägstrich haben.

  Aus den beiden Attributen host und path setzt sich die URI zusammen, an die die HTTP-Anfrage gesendet wird.

  Jedes Wurzel-Element der Erweiterung hat drei direkte Kind-Elemente.

- **context:** verpflichtend; Das **<context>** Element ist ein verpflichtendes Element, das für alle Header zuständig ist, die sich nicht auf die Content-Negotiation beziehen. Dieses Element ist eine Erweiterung des eigentlichen **<context>** Elements um das **<ref>** Attribut. Der Inhalt des ursprünglichen **<context>** Elements wird in der detaillierten Erläuterung besprochen.

  Attribute:

  - **ref:** String, Variablenreferenz, optional; Die, hier angegebene, BPEL-Variable muss vom Typ XSD Element sein und eine Instanz von context beinhalten.

  Die Verarbeitung des context-Elements beginnt mit der Variablenreferenz. Ist eine Variablenreferenz vorhanden, wird das **<context>** Element, in dieser Variable eingelesen und gespeichert. Erst dann werden die Kind-Elemente des eigentlichen **<context>** Elements gelesen. Sollte es dabei dazu kommen, dass aus der Variable und dem vorliegendem **<context>** Element unterschiedliche Werte für den selben HTTP-Header eingelesen werden, wird der Wert aus der Variable mit dem Wert aus dem vorliegendem **<context>** Element überschrieben. Die detaillierte Syntax des **<context>**Elements wird in 4.3.4 beschrieben.

- **requestParameters:** Das **<requestParameters>** Element wird in 4.3.2 besprochen.
- **responseParameters:** Das **<responseParameters>** Element wird in 4.3.3 besprochen.

4.3.2 Das <requestParameters> Element

In dieser Sektion wird das **<requestParameters>** Element besprochen. Siehe dazu Listing 4.2.

requestParameters: verpflichtend; Dieses Element enthält vor allem Informationen, die für den Empfänger der HTTP-Anfrage wichtig sind. Dazu gehören Angaben zur Content-Negotiation und Angaben zu einer etwaigen Request-Entity(HTTP-Message-Body).

Listing 4.2 Detaillierte Syntax des requestParameters Element

```
<requestParameters>
       <contentNegotiation>
              <acceptEntity type="" priority=""/>
              <acceptLanguage priority="">value</acceptLanguage>
       </contentNegotiation>
       <requestEntity type="" entity="">
              <language>value</language>
       </requestEntity>
</requestParameters>
```

- **contentNegotiation:** optional; Dieses Element befasst sich mit HTTP-Content-Negotiation.
- **acceptEntity:** beliebig; Mit Hilfe des **<acceptEntity>** Elements kann der Wert des Accept-Headers[htt99, 14.1] konfiguriert werden.

Attribute:

  - **type:** String, verpflichtend; in diesem Attribut wird der Medientyp eingetragen, der dem Accept-Header hinzugefügt werden soll.

  - **priority:** FloatingPoint zwischen 0 und 1, optional; mit diesem Attribut lassen sich die Medientypen priorisieren gemäß [htt99, 3.9]. Dazu kann dieses Attribut Werte zwischen 0 und 1 annehmen. Als Dezimaltrennzeichen ist ein Punkt zu verwenden und es sind maximal 3 Nachkommastellen erlaubt.

  Der gesamte Accept-Header wird von der Erweiterung aus allen vorhanden **<acceptEntity>** Elementen berechnet und gesetzt.

- **requestEntity:** optional; kann aber maximal einmal vorhanden sein. Es wird zum Beschreiben und Setzen des HTTP-Message-Body verwendet.

Attribute:

  - **type:** String, verpflichtend; In diesem Attribut steht der Medientyp, der später in den Content-Type-Header geschrieben wird.
  - **entity:** String, Variablenreferenz, verpflichtend; Der Inhalt der, in diesem Attribut angegebenen, Variable wird anhand des vorgegebenen Medientyps in die HTTP-Message serialisiert.

- **language:** optional; Das Kind-Element von **<requestEntity>** ist das **<language>** Element. Der Wert dieses Elements wird in den Content-Language-Header geschrieben.

Unterschiede des <requestParameters> Elements bei verschiedenen HTTP-Verben

Beim **<requestParameters>** Element gibt es Unterschiede zwischen den einzelnen HTTP-Verben der Erweiterung. Das **<requestEntity>** Element ist nur bei Verwendung von PUT und POST erlaubt. Bei HEAD, GET und DELETE ist es nicht vorhanden. Die HTTP 1.1 Spezifikation [htt99] verbietet Message-Bodies nicht explizit für HEAD, GET und DELETE. Die Semantik dieser Verben ist aber in der Spezifikation klar definiert.
GET bedeutet, jegliche Information abzufragen, die von einer URI identifiziert wird. Die Semantik von GET kann nur durch „If-...-Header“ zu einem „conditional Get“, oder durch einen Range-Header zu einem „partial Get“ geändert werden. Ein evtl. vorhandener MessageBody macht aber nur Sinn, wenn die Semantik von Get abweichend von der spezifizierten Semantik geändert werden soll [htt99, 9.3].
HEAD ist identisch zu GET, mit der Ausnahme, dass der verarbeitende Server, keinen message-body zurücksenden darf [htt99, 9.4].
DELETE verlangt, dass der Server die Ressource löscht, die von der URI identifiziert wird. Laut Spezifikation kann diese Methode überschrieben werden. In [htt99, 4.3] steht, dass ein Server den Entity-Body ignorieren sollte, wenn die Definition der anfragenden Methode keine definierte Semantik für einen Entity-Body beinhaltet.

4.3.3 Das <responseParameters> Element

In dieser Sektion wird das **<responseParameters>** Element besprochen. Siehe dazu Listing 4.3.

responseParameters: verpflichtend; Dieses Element enthält vor allem Informationen enthalten, die zur Verarbeitung von Antworten auf HTTP-Anfragen durch die Erweiterung benötigt werden. Dazu gehören unter anderem Angaben zur Übertragung von HTTP-Message-Bodies in BPEL und Angaben zu Verhalten beim Empfang bestimmter HTTP-Status-Codes.

Listing 4.3 Detaillierte Syntax des **<responseParameters>** Element

```
<responseParameters>
       <responseHeader variable=""/>
       <acceptEntityMapping type="" variable=""/>
       <catch status="" faultName=""/>
</responseParameters>
```

- **responseHeader:** optional; Im **<responseHeader>** Element kann angegeben werden, wo die Header-Felder einer Antwort gespeichert werden. Um die Daten in BPEL nutzbar zu machen, werden die Header-Felder in Form eines XML-Dokuments der Form ResponseHeader 4.3.5 in eine BPEL-Variable geschrieben.

  Attribute:

  - **variable:** String, Variablenreferenz, verpflichtend; In die, in diesem Attribut angegebene, Variable wird das **<ResponseHeader>** 4.3.5 Element geschrieben wird.

- **acceptEntityMapping:** beliebig; Da Variablen in BPEL streng typisiert sind, können nicht alle Entitäten verschiedener Medientypen in die selbe BPEL-Variable geschrieben werden. Das **<acceptEntityMapping>** dient der Zuordnung von akzeptierten Medientypen zu bestimmten BPEL-Variablen. Dabei wird mittels einem **<acceptEntityMapping>** einer BPEL-Variable genau ein Medientyp zugeordnet.

  Attribute:

  - **type:** String, verpflichtend; in dieses Attribut wird der Medientyp geschrieben.
  - **variable:** String, Variablenreferenz, verpflichtend; in die, in diesem Attribut angegebene, Variable wird der empfangene Message-Body des gleichen Medientyps, wie im Attribut **<type>**, geschrieben.

- **catch:** beliebig; Das **<catch>** Element ist ein Element, das bestimmten HTTP-Status bestimmte BPEL-Fehler zuordnet, die dann von der Erweiterung ausgelöst werden.

  Attribute:

- **status:** String, in diesem Attribut kann angegeben werden für welche Status ein Fehler ausgelöst werden soll. Als Beispiel kann in diesem Feld 4xx stehen um z.B. für alle Fehler-Status den gleichen BPEL-Fehler auszulösen. Ebenfalls können so auch einzelne Status zugeordnet werden.
- **faultName:** String, in diesem Attribut steht der Name des BPEL-Fehlers, der beim Empfang des zugeordneten Status ausgelöst werden soll.

Unterschiede der <responseParameters> bei verschiedenen HTTP-Verben

Beim **<responseParameters>** Element gibt es Unterschiede zwischen den einzelnen HTTP-Verben der Erweiterung. Beim Verb HEAD ist das Element **<acceptEntityMapping>** nicht erlaubt. Bei allen anderen Verben ist es vorhanden. Die HTTP-Spezifikation verbietet explizit einen Message-Body in Antworten auf HEAD Anfragen [htt99, 4.3].

4.3.4 Das <context> Element

Listing 4.4 Detaillierte Syntax des context Elements

```
<context>
       <date value="boolean"/>
       <closeConnection value="boolean"/>
       <authentication>
              <user>...value</user>
              <pass>...value</pass>
       </authentication>
       <cachecontrol>...value</cachecontrol>
       <conditionals>
              <ifMatch>value</ifMatch>
              <ifModifiedSince>value</ifModifiedSince>
              <ifNoneMatch>value</ifNoneMatch>
              <ifRange>value</ifRange>
              <ifUnmodifiedSince>value</ifUnmodifiedSince>
       </conditionals>
       <additionalHeaders>
              <header name="" value=""/>
       </additionalHeaders>
</context>
```

Erläuterung der Elemente und Attribute von context, siehe Listing 4.4

- **date:** optional; Dieses Element zeigt an, ob ein Date-Header gesetzt werden soll. Ist das Element vorhanden, das zugehörige Attribut value aber nicht vorhanden, wird ein Date-Header gesetzt. Kein Date-Header wird gesetzt, wenn das Element entweder nicht vorhanden ist, oder das Attribut **<value>** mit false belegt ist.

Attribute:

  - **value:** Boolean, optional; zeigt an, falls vorhanden ob der Date-Header gesetzt wird.

- **closeConnection:** optional; Dieses Element korrespondiert zum Connection-Header. (HTTP-spec 14.2). In HTTP 1.1 wird im Gegensatz zu früheren Versionen von HTTP jede Verbindung als persistent betrachtet (HTTP-spec 8-1-2). Das **<closeConnection>** Element zeigt an, ob ein Connection-Header mit dem Wert „close“ gesetzt wird. Die Funktionsweise ist dabei die gleiche, wie beim **<date>** Element.

  Attribute:

  - **value:** Boolean, optional; zeigt an, falls vorhanden, ob der Connection-Header mit dem Wert „close“ gesetzt wird.

- **authentication:** optional; Dieses Element korrespondiert zum WWW-Authenticate-Header [htt99, 14.47] und ist für die Authentifizierung zuständig. Ist dieses Element vorhanden, müssen auch die beiden Kind-Elemente **<user>** und **<pass>** vorhanden sein. Diese beiden Elemente sind für die Angabe eines Benutzernamens (**<user>**) und des zugehörigen Passworts (**<pass>**) zuständig. Diese Angaben stehen jeweils in den Elementen. Die Authentifizierung selbst wird mit Hilfe der Authentifizierungsdaten von der Erweiterung übernommen.

- **cachecontrol:** optional; Dieses Element korrespondiert direkt zum Cache-Control-Header [htt99, 14.9]. Der Text-Inhalt des Elements wird von der Erweiterung direkt in einen Cache-Control-Header geschrieben und muss daher HTTP-spezifikationskonform sein.

- **conditionals:** optional; Mit diesem Element und den dazugehörigen Kind-Elementen lassen sich Conditional-Gets ausführen [htt99, 9.3]. Die Kind-Elemente von **conditionals** sind: **<ifMatch>**, **<ifModifiedSince>**, **<ifNoneMatch>**, **<ifRange>** und **<ifUnmodifiedSince>**. Der Text-Inhalt dieser Elemente wird, wie bei **<cacheControl>**, direkt in die entsprechenden Header-Felder geschrieben und muss daher HTTP-spezifikations-konform sein.

- **additionalHeaders:** optional; Das Element **<additionalHeaders>** ist ein Element, um alle weiteren Header zu gruppieren. Das Element **<additionalHeaders>** kann dazu beliebig viele (0..n) **<header>** Elemente als Kinder haben.

- **header:** beliebig; Dieses Element ist dafür vorgesehen, alle möglichen Header-Felder setzen zu können, die nicht bereits durch andere Kind-Elemente des **<context>** Elements abgedeckt sind.

  Attribute:

  - **name:** String, verpflichtend; in diesem Attribut steht der Name des Header-Felds.

  - **value:** String, verpflichtend; in diesem Attribut steht der Wert, der direkt in das entsprechende Header-Feld gesetzt wird.

Listing 4.5 Beschreibung

```
<responseHeader>
       <status></status>
       <content>
              <type>value</type>
              <language>value</language>
              <location>value</location>
       </content>
       <date>value<date>
       <header name="" value=""/>
</responseHeader>
```

4.3.5 Das <responseHeader> Element

In dieser Sektion wird das **<responseHeader>** Element besprochen. Siehe Listing 4.5.
Das **<responseHeader>** Element dient dem Zweck, auch Meta-Daten einer etwaigen HTTP-Antwort verfügbar zu machen.

Erläuterung der einzelnen Elemente aus Listing 4.5.

- **status:** verpflichtend; Dieses Element dient Zum Speichern des HTTP-Antwort-Status.
- **content:** optional; Dieses Element enthält über seine Kind-Elemente Informationen zur Content-Negotiation.
- **type:** optional; Dieses Element enthält den Medientyp aus dem Content-Type-Header der HTTP-Antwort.
- **language:** optional; Dieses Element enthält den Wert des Content-Language-Headers der HTTP-Antwort.
- **location:** optional; Dieses Element enthält den Wert des Content-Location-Headers der HTTP-Antwort.
- **date:** optional; Dieses Element enthält den Wert des Date-Headers der HTTP-Antwort.
- **header:** beliebig; Dieses Element ist dafür vorgesehen, alle möglichen Header-Felder verfügbar zu machen, die nicht bereits durch andere Kind-Elemente des **<responseHeader>** Elements abgedeckt sind.

Attribute:

  - **name:** String, verpflichtend; in diesem Attribut steht der Name des Header-Felds.
  - **value:** String, verpflichtend; in diesem Attribut steht der Wert aus dem entsprechenden Header der HTTP-Antwort.

4.4 Evaluierung des Konzepts anhand der Anforderungen aus 3.1

Evaluierung

1. **Kontrolle über die URI, an die die HTTP-Anfragen geschickt werden**
   Die URI wird in der Extension durch die Attribute **<host>** und **<path>** berechnet. Der Vorteil ist, dass das Berechnen der URI zur Laufzeit geschieht und somit der Endpunkt nicht bereits zur Deploy-Zeit bekannt sein muss. Es ist also möglich die Endpunkte genutzter REST Services zur Laufzeit von BPEL-Prozess-Instanzen zu ändern.

2. **Unterstützung für Content-Negotiation**
   Server-driven Content-Negotiation [htt99, 12.1] in HTTP funktioniert in erster Linie durch die Auswertung gesendeter Accept-Header durch den Server. Dabei errechnet der Server, anhand der akzeptierten Medientypen und deren Prioritäten, den optimalen Content-Type und sendet seine Antwort mit diesem Content-Type. Zu den Accept-Headern gehören Accept, Accept-Charset, Accept-Encoding, Accept-Language und User-Agent. Die Header-Felder Accept-Charset und Accept-Encoding sind hauptsächlich technisch relevant. D.h. dem letztlichen Nutzer der Erweiterung wird es weitestgehend egal sein, welches character set [htt99, 3.4] oder welches content-coding verwendet wird. Diese beiden Felder werden deshalb von der Erweiterung automatisch gesetzt.
   Das Header-Feld User-Agent Feld [htt99, 14.43] enthält Informationen über den Client, der eine HTTP-Anfrage ursprünglich absendet. Er ist daher nicht für einzelne Anfragen wichtig, sondern wird für viele Anfragen genutzt. Dieser Header kann im **<context>** im Element **<additionalHeaders>** als einzelnes **<header>** Element gesetzt werden.
   Die Header-Felder Accept und Accept-Language, sind die Header-Felder, die wirklich relevant sind für den Nutzer. Da sie sich oft, von Anfrage zu Anfrage, unterscheiden, können sie innerhalb des contentNegotiation Elements mit Hilfe der Elemente acceptEntity und acceptLanguage in vollem Umfang, also incl. Prioritäten gemäß [htt99, 3.9], genutzt werden.

   Bei der Agent-driven Content-Negotiation entscheidet der Client darüber, nach initialem Empfang einer Antwort vom Server, welche Repräsentation er akzeptieren will. Die Auswahl basiert auf Informationen in den Header-Feldern oder im Entity-Body der initialen Antwort. Zumindest die Header-Felder der Antwort können mittels responseHeader in BPEL nutzbar gemacht werden. Der Entity-Body kann in diesem Fall innerhalb von BPEL nur genutzt werden, wenn das Format des Entity-Body von der Erweiterung untertützt wird und wenn die agent-driven Content-Negotiation innerhalb des BPEL-Prozesses „von Hand“ durchgeführt wird.

3. **Unterstützung für HTTP-Header**
   Mit dem **<context>** Element, lassen sich alle möglichen HTTP-Header-Felder set-

zen. Dabei kann es zu Einschränkungen kommen, da manche Header automatisch gesetzt werden, z.B. Content-Length, oder aber Implementations-spezifisch sind, wie z.B. Accept-Charset.
Mit dem **<responseHeader>** Element können alle Header-Felder in BPEL gelesen werden.

Fazit

Durch die Möglichkeit, während der Laufzeit von BPEL-Prozess-Instanzen, die Endpunkte genutzter REST Services zu ändern, durch die Unterstützung von Content-Negotiation und durch die Unterstüztung aller HTTP-Header, werden alle Anforderungen für RESTful BPEL erfüllt. Die Erweiterung ist somit dafür **geeignet** RESTful BPEL zu praktizieren.

5 BPEL4REST - Implementierung

In Kapitel 5 wird die Implementierung der Erweiterung für BPEL, mit dem Namen **BPEL4REST**, dokumentiert. Zunächst werden verwendete Technologien kurz vorgestellt, anschließend folgt die Erläuterung der Architektur, der interne Ablauf von BPEL4REST und zuletzt werden noch Einschränkungen von BPEL4REST erläutert.

5.1 Verwendete Technologien - Entwicklungsumgebung

Im Unterkapitel 5.1 werden, für BPEL4REST eingesetzte und verwendete, Technologien kurz beschrieben. Diese sind Apache ODE, Apache Tomcat, Apache HttpComponents Client und Eclipse.

5.1.1 Apache ODE

Die Apache ODE (Orchestration Director Engine) ist eine BPEL-Engine, die von der Apache Software Foundation [apa] entwickelt und betreut wird.
ODE kommuniziert mit Web Services, sendet und empfängt Nachrichten, kümmert sich um Datenmanipulation und Fehlerbehebung gemäß der Beschreibung durch BPEL-Prozess-Definitionen. ODE unterstützt sowohl kurz, als auch langlaufende Prozesse.[odea]

Für BPEL4REST wird die WAR Distribution „experimental branch 2.0-beta8“[1] verwendet. Nur in diesem Branch ist die Funktionalität enthalten, die nötig ist, ODE durch eine ExtensionActivity zu erweitern.

5.1.2 Apache Tomcat

Apache Tomcat ist eine OpenSource Implementierung der **Java Servlet** und **JavaServer Pages** Technologie. Apache Tomcat gehört ebenfalls zur Apache Software Foundation [apa]. Apache Tomcat wird als Servlet container für Apache ODE benötigt.
Für BPEL4REST kommt die Version 7.0(.40) zum Einsatz, verfügbar unter: [tom].

[1] http://ode.apache.org/getting-ode.html

5.1.3 Apache HttpComponents Client

Apache HttpComponents [htta] ist ebenfalls ein Projekt der Apache Software Foundation [apa]. Der Apache HttpComponents Client ist eine HTTP 1.1 konforme Open Source Implementierung eines HTTP 1.1 Clients. Apache HttpComponents wird im Rahmen von BPEL4REST dazu eingesetzt, HTTP-Anfragen zu Senden und zu Empfangen.
BPEL4REST verwendet die Version HttpClient 4.2.5 (GA), verfügbar unter: [httb].

5.1.4 Eclipse

Eclipse ist eine Open Source Entwicklungsumgebung von der Eclipse Foundation [ecla].
Für die Entwicklung von BPEL4REST wurde Eclipse Juno 4.2 Service Release 2 verwendet.
Aktuelle Versionen von Eclipse sind verfügbar unter:[eclb], ältere Versionen unter: [eclc]

5.2 Architektur

In 5.2 wird die Architektur von BPEL4REST beschrieben. Zunächst wird in der Übersicht die grobe Struktur gezeigt, und beschrieben. Anschließend werden die einzelnen Komponenten detailliert beschrieben.

5.2.1 Übersicht

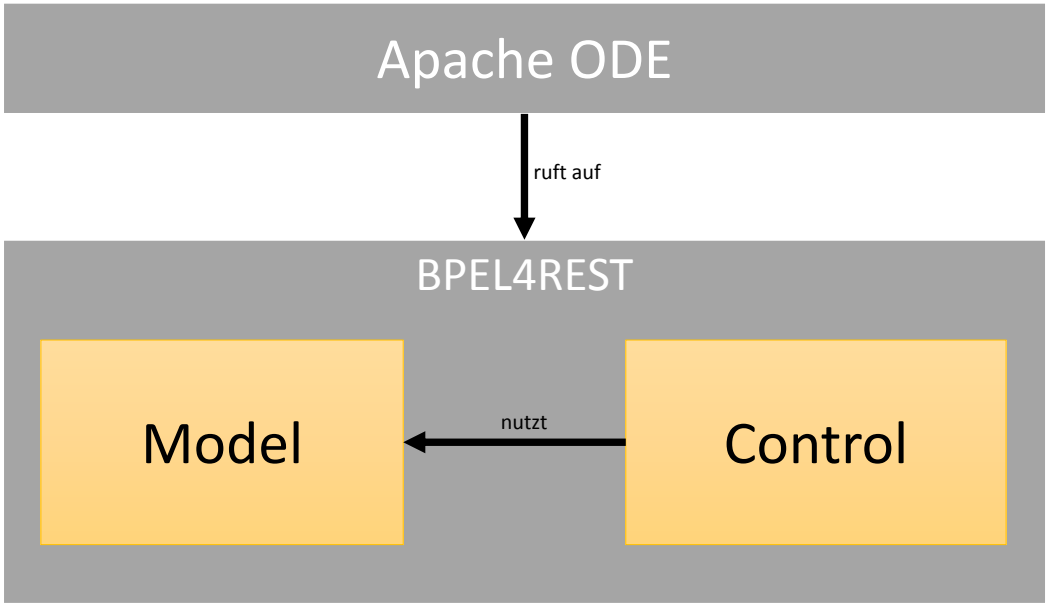

Abbildung 5.1: Übersicht der Architektur von BPEL4REST

In Abbildung 5.1 ist die Grob-Architektur von BPEL4REST zu sehen. Die Architektur von BPEL4REST ist in zwei Komponenten aufgeteilt, die nur durch eine klar definierte Schnittstelle miteinander kommunizieren. Aufgerufen wird BPEL4REST direkt durch Apache ODE. Aufgabe des Models ist einzig die Datenhaltung. Die Aufgabe der Control-Komponente ist die Steuerung der Extension, das Senden und Empfangen von HTTP-Nachrichten, sowie die zugehörige Datentransformation und der Datenaustausch mit dem ausführendem BPEL-Prozess. Details zu den einzelnen Komponenten werden in 5.2.2 und 5.2.3 erläutert.

5.2.2 Komponente Model

Die Detail-Architektur der Modelkomponente, Abbildung 5.2, ist so ausgelegt, dass die tatsächliche Implementierung schnell ausgetauscht werden kann. Die Funktionalität des Models wird dabei über drei Schnittstellen definiert:

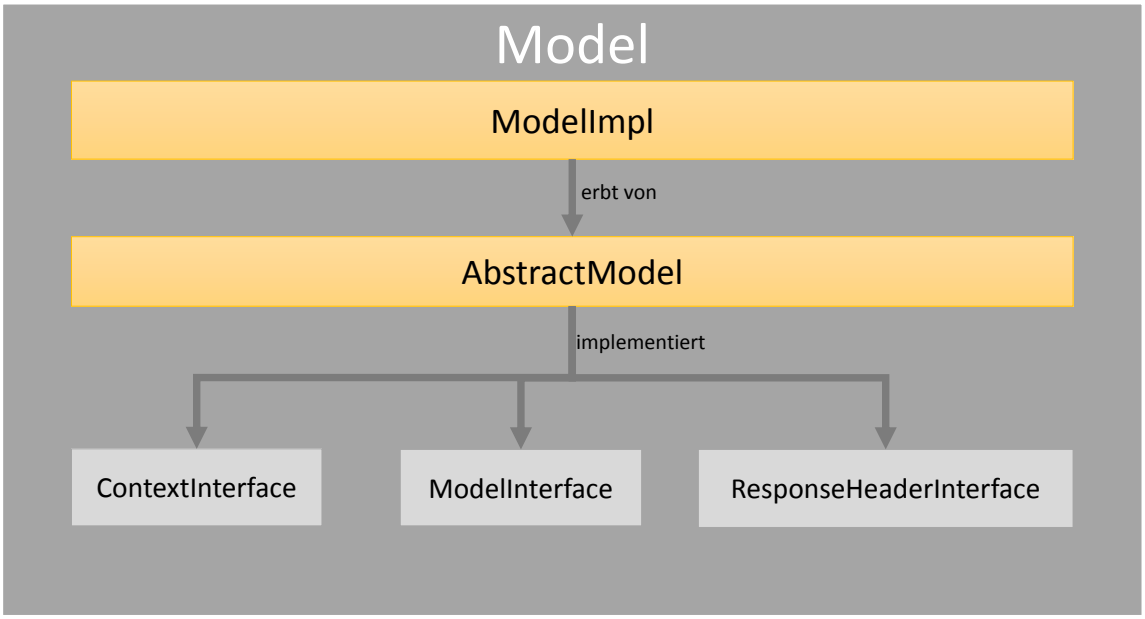

Abbildung 5.2: Architektur des Models im Detail

1. **ContextInterface:** Das ContextInterface definiert die Schnittstelle zum Speichern aller Informationen für das **context** Element der Erweiterung.
2. **ResponseHeaderInterface:** Das ResponseHeaderInterface definiert die Schnittstelle zum Speichern aller Informationen für das **responseHeader** Element der Erweiterung.
3. **ModelInterface:** Das ModelInterface definiert die Schnittstelle zum Speichern aller Informationen, für das Extension Element, das der BPEL4REST-Erweiterung von der ODE übergeben wird.

Die Klasse **AbstractModel** stellt eine abstrakte Klasse dar. In dieser Klasse wird keine zusätzliche Funktionalität definiert. Sie dient nur dem Zweck die drei Schnittstellen (ContextInterface, ResponseHeaderInterface und ModelInterface) zusammenzuführen.

Die Klasse **ModelImpl** wird von der Klasse AbstractModel abgeleitet. Sie liefert also letztlich sämtliche Funktionalität zur Verwaltung der benötigten Informationen der Erweiterung.

5.2.3 Komponente Control

Die Detail-Architektur der Control-Komponente, Abbildung 5.3, ist zweigeteilt. Auf der einen Seite stehen Controller-Klassen, die den Kontrollfluss der Erweiterung steu-

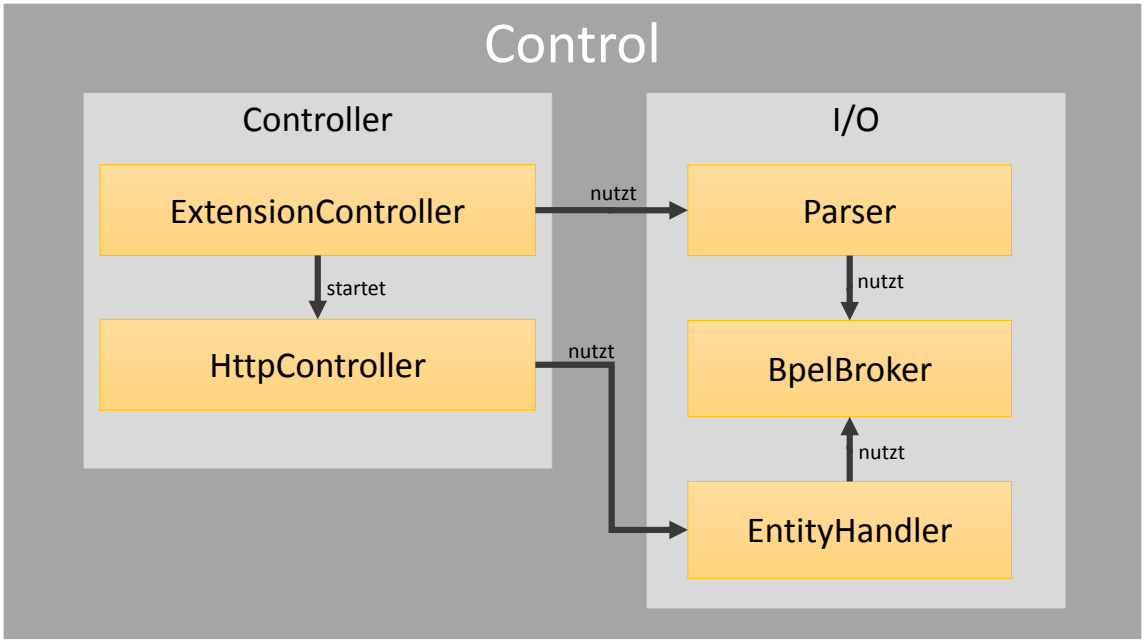

Abbildung 5.3: Architektur der Control-Komponente

ern, auf der anderen Seite steht die I/O, die sich um die Kommunikation mit dem aufrufendem BPEL-Prozess kümmert.

Wichtige Klassen:

- **Controller:**
  - **ExtensionController:** Die Klasse ExtensionController ist die hauptsächliche Controller-Klasse. Zur Instanziierung benötigt diese Klasse Objekte folgender Klassen:
    1. **ExtensionMode:** ExtensionMode ist ein **enum** der anzeigt, welche Methode ausgeführt werden soll. Er kann folgende Werte annehmen: *HEAD, GET, PUT, POST, DELETE*.
    2. **ExtensionContext:** ExtensionContext ist eine Klasse der ODE. Mit der Klasse ExtensionContext ist es möglich auf Daten der ODE zuzugreifen.
    3. **org.w3c.dom.Element:** Das ExtensionElement, also die XML-Repräsentation, siehe 4.3, wird als Instanz der Klasse org.w3c.dom.Element übergeben.

    ExtensionController-Objekte steuern den Ablauf der Erweiterung. Sie sind dafür verantwortlich, alle erfoderlichen Klassen zu instanziieren und entsprechend zu steuern. Diese Klassen sind: ModelImpl, BpelBroker, ExtensionParser, HttpController und EntityHandler.

  - **HttpController:** Die Klasse HttpController übernimmt die Steuerung und Verarbeitung der HTTP Kommunikation. Zu ihren Aufgaben gehört das Erstellen und Absenden von HTTP-Anfragen, sowie das Empfangen und Verarbeiten von HTTP-Antworten. Zur Instanziierung benötigt der HttpController Objekte der Klassen **Model** und **EntityHandler**.

- **I/O:**
  - **BpelBroker:** Die Klasse BpelBroker stellt die Schnittstelle zwischen BPEL4REST und ODE dar. BpelBroker liefert beispielsweise Methoden zum Lesen und Setzen von Variablen in BPEL. Zur Instanziierung benötigt der BpelBroker ein Objekt der Klasse **ExtensionContext**.
  - **ExtensionParser:** Die Aufgabe des ExtensionParsers ist es, das übergebene org.w3c.dom.Element einzulesen und in ein Objekt der Klasse ModelImpl zu speichern. Der ExtensionParser muss mit einem Objekt der Klasse **ModelImpl** und dem, von ODE übergebenem, org.w3c.dom.Element instanziiert werden.
  - **EntityHandler:** Die Klasse EntityHandler kann BPEL-Variablen auslesen und anhand des übergebenen Medientyps in eine HttpEntity umwandeln. Ebenfalls kann sie übergebene Objekte der Klasse HttpEntity wieder anhand des Medientyps umwandeln und in eine BPEL-Variable schreiben. Zur Instanziierung benötigt der EntityHandler Objekte der Klassen **ModelImpl** und **BpelBroker**. Für Details zur Implementierung siehe 5.2.3.

Der EntityHandler im Detail

Der EntityHandler verwendet Implementierungen der abstrakten Klasse EntityTransformer. Diese abstrakte Klasse verfügt über drei abstrakte Methoden, die implementiert werden müssen:

1. **HttpEntity getEntityFromBpel(RequestEntity req):** In dieser Methode wird die BPEL-Variable gelesen und in eine Instanz von HttpEntity umgewandelt, die letztlich vom HttpController verwendet werden kann, um die HTTP Anfrage zu erstellen. Informationen über die BPEL-Variable werden durch das RequestEntity-Objekt übergeben.
2. **void writeEntityToBpel(HttpEntity entity, BPELVariable var)** In dieser Methode wird eine HttpEntity in eine BPEL-geeignete Datenstruktur umgewandelt(org.w3c.dom.Element) und in die übergebene BPELVariable geschrieben.
3. **String[] providedMediatypes()** Von dieser Methode wird ein String-Array zurückgegeben, das alle Medientypen beinhaltet, die von dieser Implementierung verarbeitet werden können. Die unterstützten Medientypen können auch als Regex-Ausdruck angegeben werden.

Der EntityHandler registriert bei seiner Instanziierung alle angegebenen Implementierungen von EntityTransformer und ordnet sie ihren, mittels **providedMediatypes()** definierten, Medientypen zu. Wird nun eine der beiden öffentlichen Methoden **getEntityFromBpel()** und **writeEntityToBpel()** der Klasse **EntityHandler** aufgerufen, wählt der EntityHandler eine geeignete Implementierung von EntityTransformer aus, instanziiert sie und führt mit ihrer Hilfe die geforderte Funktion aus.

Dieser Mechanismus macht es sehr einfach, die Erweiterung um neue Medientypen zu ergänzen.

Verarbeitung von Daten, die nicht dem XML-Format entsprechen

Die Typisierung von Variablen erfolgt in BPEL über XML-Schema. Da im REST-Umfeld auch andere Formate verwendet werden, beispielsweise JSON oder CSV, ist es nötig, diese gesondert zu behandeln.
BPEL4REST löst dieses Problem dadurch, dass Implementierungen von EntityTransformer für Datenformate die nicht XML-basiert sind, eine bijektive Abbildung zwischen XML und dem von XML abweichenden Datenformat definiert.
BPEL4REST betrachtet dabei in erster Linie den Fall, dass eine REST-Ressource keine XML-Repräsentation anbietet und ein BPEL-Prozess nur deshalb ein abweichendes Format wählt. Es ist also nicht im Sinne von BPEL4REST komplexe XML-Strukturen in ein anderes Format abzubilden. Im Falle von JSON als abweichendem Format, implementiert der zugehörige EntityTransformer also eine Abbildung von JSON nach XML nach JSON. Diese Einschränkung ist deshalb wichtig, da XML oftmals komplexer ist, als ein abweichendes Format (Extrembeispiel: CSV).
BPEL4REST nutzt zur Implementierung des EntityTransformer für „application/json“ die externe Java-Bibliothek **JSON-lib**[2], die unter der **Apache Software License, Version 2.0** steht.

5.3 Ablauf der Erweiterung

Der Kontrollfluss (Abbildung 5.4) der Erweiterung beginnt mit einer Implementierung der Klasse **AbstrctSyncExtensionOperation**, indem ODE die Methode **runSync(ExtensionContext context, Element element)** aufruft.
Hier wird dann der ExtensionController mit dem entsprechendem ExtensionMode instanziiert und anschließend gestartet.
Der ExtensionController kümmert sich zunächst um die Instanziierung des Models und des BpelBrokers. Diese werden anschließend vom Parser benötigt, der zunächst vom ExtensionController instanziiert wird, dann die übergebenen Daten einliest und in das Model

[2]http://json-lib.sourceforge.net/

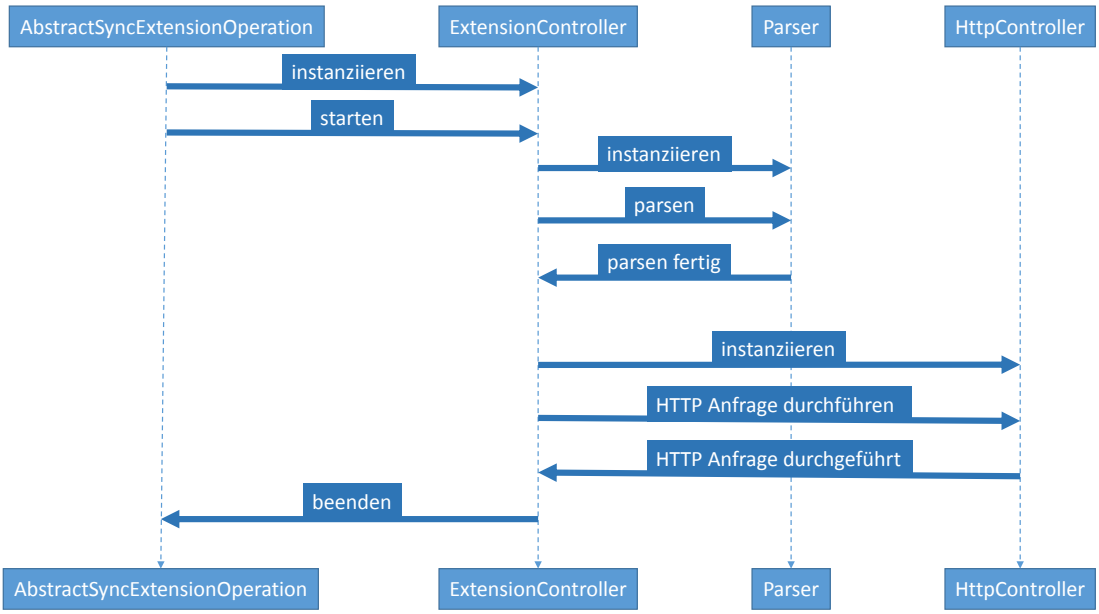

Abbildung 5.4: Ablauf der Erweiterung

speichert.
Nach Abschluss des Parsens instanziiert der ExtensionController den HttpController und startet ihn anschließend. Nach Abschluss der HTTP-Kommunikation beendet schließlich der ExtensionController die Erweiterung.

5.4 Einschränkungen

In diesem Kapitel werden die Einschränkungen beschrieben, denen die BPEL4REST-Implementierung unterliegt:

- **Modellierung:** BPEL4REST wird von keinem Modellierungstool unterstützt. Das **extensionActivity** Element muss also „von Hand“ modelliert werden.
- **Laufzeit-Umgebung:** BPEL4REST kann nur mit einer speziellen Version der BPEL-Engine Apache ODE genutzt werden. Diese Version ist der **Experimental Branch beta 2.0**.
- **Medientypen:** Derzeit beherrscht BPEL4REST die Medientypen „text/plain“, „application/json“ und alle XML-Typen.
- **Validierung:** Derzeit findet keine Validierung des übergebenen Extension-XML-Elements statt.

- **HTTP-Header:** Derzeit werden manche HTTP-Header noch von BPEL4REST ignoriert. Beispielsweise Content-Encoding.

6 Verwandte Arbeiten

Dieses Kapitel beschreibt Arbeiten, die thematisch mit dieser Arbeit verwandt sind. Bei der Recherche nach BPEL und REST stößt man dabei stets auf den Namen Cesare Pautasso und JOpera. JOpera [jop] ist ein Werkzeug zur Service Komposition für Eclipse[ecla]. JOpera bietet eine visuelle Modellierungsumgebung und eine Engine zur Ausführung von Service-Kompositionen. Laut User Manual[1] werden die Service-Kompositions-Modelle in JOpera auf einer höheren Abstraktionsebene als traditionelle „BPM/BPEL Sprachen“ definiert und decken dabei sowohl strukturelle, als auch Verhaltens-Aspekte ab.
Cesare Pautasso ist Assistenzprofessor der Fakultät Informatik der Universität von Lugano. Er verfasste unter anderem die Arbeiten „BPEL for REST“[Pau08] und „RESTful Web service composition with BPEL for REST“[Pau09], die nachfolgend erläutert werden.
Weiterhin werden noch die Arbeiten „BPEL light“, „Towards Resource-Oriented BPEL“ und „Bite: Workflow Composition for the Web“ kurz vorgestellt.

6.1 „RESTful Web service composition with BPEL for REST“ - Cesare Pautasso

Dieser Abschnitt befasst sich mit den Arbeiten [Pau08] und [Pau09] von Cesare Pautasso. Inhaltlich weichen diese beiden Arbeiten nur unwesentlich voneinander ab, weshalb nachfolgend die neuere Arbeit, **RESTful Web service composition with BPEL for REST** von 2009, die auch im Druck erschienen ist, erläutert wird.
Pautasso beschreibt darin, wie BPEL erweitert werden muss, um die Komposition von RESTful Web Services und traditionellen (WSDL) Web Services innerhalb von BPEL zu ermöglichen. Weiterhin zeigt er auch ein Konzept, Teile von BPEL-Prozessen als RESTful Web Service zu veröffentlichen (wie dies für WSDL bereits der Fall ist).
Im Kapitel **Introduction** erläutert Pautasso, dass BPEL derzeit als Standardsprache zu Komposition von Web Services verwendet wird und geht darauf ein, dass BPEL-Prozesse zum einen Web Services konsumieren, aber auch selbst als Web Service konsumiert werden können.
Als „andere Art der Abstraktion, basierend auf REST“ beschreibt er RESTful Web Services als Technologie, die die Annahmen von BPEL herausfordern. Er führt weiter aus, dass die meisten RESTful Web Services nicht durch WSDL beschrieben werden und es somit nicht möglich ist, existierende Sprachen und Werkzeuge für sie zu benutzen, die WSDL zur

[1] http://www.jopera.org/docs/help/jop_1.html#1.1

Grundlage haben. Er benennt verschiedene Sprachen, WADL[2], WRDL[3], NSDL[4], WDL[5], die zur Beschreibung von RESTful Web Services vorgeschlagen wurden, mit dem Einwand, dass die meisten tatsächlich existierenden APIs auf Dokumentation für menschliche Adressaten angewiesen sind.

Im Kapitel **Motivation** beschreibt Pautasso zunächst die enge Kopplung zwischen BPEL und WSDL und zeigt, dass diese Einschränkung viele Erweiterungen für BPEL begründet hat. Beispielsweise BPEL4People zur Interaktion mit menschlichen Partnern und weitere.
Er geht auf den existierenden Ansatz ein, REST-Aufrufe aus BPEL durch eine WSDL 2.0 Schnittstelle zu leiten, führt aber auch an, dass BPEL nur WSDL 1.1 unterstützt, und diese Lösung von einem theoretischen Standpunkt aus unbefriedigend ist, weil diese Lösung Prinzipien von REST, wie die Ressourcen-Abstraktion und deren Interaktions-Mechanismen hinter einem SOA-Ansatz versteckt. Pautasso führt weiterhin an, das WSDL 2.0 zum Einen noch nicht weit verbreitet ist, vor allem nicht zur Beschreibung von RESTful Web Services.

In Kapitel 3 **Composing RESTful Web services** geht Pautasso auf grundlegende Regeln von REST ein und erläutert Probleme die dadurch in BPEL auftreten:

- **Ressourcen Addressierung durch URI**: Pautasso schreibt, dass die Schnittstelle eines RESTful Web Services aus einer Menge von Ressourcen besteht, die durch URIs identifiziert werden, die nicht immer im Voraus bekannt sind. Er folgert, dass ein spätes Binding von URIs im REST-Umfeld eine wichtige Rolle spielt und BPEL for REST deshalb auch Binding zur Laufzeit unterstützen muss.
- **Einheitliche Schnittstelle:** Pautasso erläutert, dass mit allen identifizierten Ressoucen, durch die selben 4 Methoden: PUT, GET, POST und DELETE, interagiert werden kann. Weiterhin erklärt er die Semantik dieser Methoden und dass sie durch synchrone HTTP Anfrage-Antwort Kommunikation aufgerufen werden können.
- **Selbst-beschreibende Nachrichten:** Nach einer kurzen Erklärung selbst-beschreibender Nachrichten folgert Pautasso, dass HTTP-Header eine Anforderung an die BPEL for REST Erweiterung darstellen.
- **HATEOAS[6]:** Nach kurzer Erläuterung von HATEOAS extrahiert Pautasso die Anforderung an BPEL for REST, dass die Ressourcen URI eines BPEL-Prozesses dynamisch generiert werden kann. Weiterhin formuliert er, dass es möglich sein muss URIs von empfangenen HTTP-Nachrichten zu extrahieren und für weitere REST-Aufrufe zu verwenden.

In Kapitel 4 **BPEL for REST extension** stellt Pautasso die Erweiterungen und Veränderungen von BPEL vor, die von BPEL for REST definiert werden. Diese teilt er auf in Erweiterungen

[2]Web Application Description Language
[3]Web Resource Description Language
[4]Norm's Service Description Language
[5]Web Description Language
[6]Hypermedeia as the engine of application state

zum Aufrufen von RESTful Web Services, Erweiterungen zum Veröffentlichen von Prozessen als RESTful Web Services und kleine BPEL-Erweiterungen und Veränderungen.
Pautasso definiert zum Aufruf von REST Services 4 neue Aktivitäten, <get>, <post>, <put> und <delete>. Diese neuen Aktivitäten sind auch in der Lage mit Fehlern in der Kommunikation umzugehen, indem sie bei bestimmten HTTP-Status-Codes BPEL-Faults auslösen, die von Fault-Handlern verarbeitet werden können. Die Struktur für diese 4 Aktivitäten ist in Listing 6.1 dargestellt.
Mit Hilfe des **catch** Elements lassen sich bestimmte FaulHandler mit bestimmten Status-Codes assoziieren. Sollte im Attribut **response_headers** eine Variable angegeben sein, wird diese genutzt um die HTTP-Header der HTTP-Antwort zu speichern. Die Attribute **request** und **response** sind für Variablenreferenzen vorgesehen, in denen die request und response-payloads, gespeichert sind oder gespeichert werden sollen. HTTP-Header können durch das **header** Element definiert werden.

Listing 6.1 Struktur der neuen BPEL Aktivitäten nach [Pau09, Abbildung 3.]

```
<get uri="" response="" response_headers=""?>
       <header name="">*value</header>
       <catch code="">*...</catch>
       <catchAll>?...</catchALL>
</get>

<post uri="" request="" response="" response_headers=""?>
       ...
</post>

<put uri="" request="" response=""? response_headers=""?>
       ...
</put>

<delete uri="" response=""? response_headers=""?>
       ...
</delete>
```

Zum Veröffentlichen eines Prozesses als RESTful Web Service definiert Pautasso das **resource** Element, welches Prozessen erlauben soll, abhängig vom Erreichen ihrer Deklaration, dynamisch Ressourcen zu veröffentlichen. Sobald das **resource** Element erreicht ist, soll die zugehörige URI veröffentlicht werden. Sollte der BPEL-Scope, in dem das **resource** Element sichtbar ist, von der Prozessausführung verlassen werden, sollen Anfragen an die bereits veröffentlichte URI mit 404, Not Found, beantwortet werden. Innerhalb des **resource** Elements können wie in einem Scope Variablen deklariert werden. Diese Variablen sollen dabei nur innerhalb des **resource** Elements sichtbar sein.
Um auf Anfragen zu Antworten, definiert Pautasso die Request Handler Elemente **onGet**, **onPut**, **onDelete** und **onPost** als Kind-Elemente von **resource**. Sollte eine Anfrage stattfinden und das entsprechende Element nicht vorhanden sein, soll die BPEL-Engine mit 405, Method Not Allowed, antworten. Innerhalb dieser Elemente können die in BPEL vorhandenen strukturellen Aktivitäten zur Berechnung der Antwort verwendet werden. Ihre Anfragen können diese Elemente mit Hilfe des **respond** Elements beantworten, wobei das **code** Attribut zur

Definition des HTTP-Status-Codes bestimmt ist. Die Struktur dieser Erweiterungen ist in Listing 6.2 dargestellt. Um HTTP-Verb-Semantik zu garantieren definiert Pautasso noch folgende Einschränkungen:

- Um Sicherheit zu garantieren hat der **onGet**-Request Handler nur Leserechte auf Variablen.
- Da nur POST nicht idempotent is, kann das **onPost** Element mit dem **isolated** Attribut markiert werden. Dieses Attribut funktioniert ähnlich zu dem isolated Attribut von Scopes.

Die Struktur dieser Erweiterungen ist in Listing 6.2 ersichtlich. Zu den kleinen Erweiterun-

Listing 6.2 Struktur der Deklaration von Ressourcen in BPEL-Prozessen nach [Pau09, Abbildung 4.]

```
<resource uri="">
<variable>*
<onGet>?              ...</onGet>
<onPut>?              ...</onPut>
<onDelete>?           ...</onDelete>
<onPost isolated="false"? >?        ...</onPost>
</resource>

<respond code=""?>
       <header name="">*value</header>
       payload
</respond>
```

gen und Veränderungen definiert Pautasso, dass die **exit**-Aktivität zusätzlich zum Beenden der Prozessausführung auch noch den Status mit allen assoziierten Ressourcen verwirft. Weiterhin soll die statische Variablentypisierung optional werden und das messageType-Attribut, das WSDL-basiert ist, wird nicht mehr genutzt.

In Kapitel 5 **Reference architecture** zeigt Pautasso wie die Referenzarchitektur einer BPEL for REST-Engine aussieht. Diese ist dargestellt in Abbildung 6.1. Pautasso schreibt, dass die vorgeschlagenen Aktivitäten als BPEL extension activities gesehen werden können, für die im Back-End ein spezielles Ausführungsmodul bereitgestellt wird. Die Verwaltung des Status von Prozess-Ressourcen kann laut Pautasso von Mechanismen zur Status-Haltung übernommen werden, die bereits in einer BPEL-Engine vorhanden sind. Das Veröffentlichen von Prozess-Ressourcen und deren URIs soll von einem „servlet-ähnlichem“ Mechanismus übernommen werden, der zusammen mit der BPEL-Engine im zugehörigen Applikations-Container betrieben wird. Dieses Servlet soll HTTP-Anfragen von Clients bearbeiten, indem es sie an die entsprechenden Request Handler weiterleitet. In Kapitel 6 **Example** zeigt Pautasso anhand eines Beispiels, wie die vorgeschlagenen Erweiterungen genutzt werden können, in Kapitel 7 **Discussion** bewertet und bespricht Pautasso die vorgeschlagenen Erweiterungen. Kapitel 8 **Related work** befasst sich mit verwandten Arbeiten und in Kapitel 9 **Conclusion** wird noch ein Fazit gezogen.

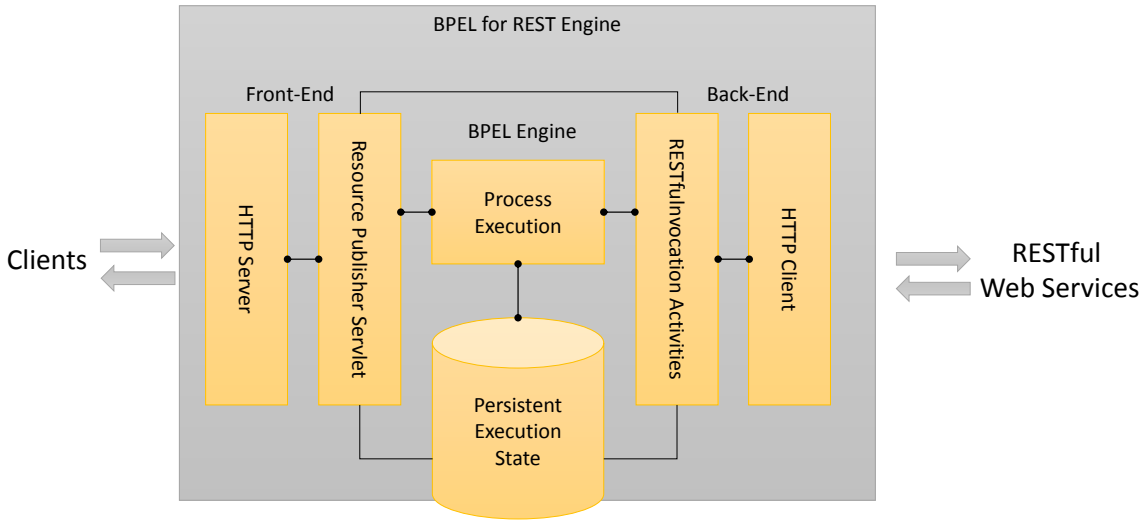

Abbildung 6.1: Referenz Architektur für die BPEL for REST Erweiterung nach[Pau09, Abbildung 6.]

6.1.1 Abgrenzung zu BPEL4REST

BPEL4REST unterscheidet sich in mehreren Punkten wesentlich von BPEL for REST. Ziel von BPEL4Rest ist es, die Orchestrierung von RESTful Web Services durch BPEL zu ermöglichen. Der Ansatz, den BPEL4REST dabei verfolgt ist, dass der Benutzer nicht über detaillierte Kenntnisse von HTTP verfügen muss. Durch die umfangreichere Gestaltung der XML-Struktur der einzelnen Aktivitäten, ist es für Nutzer leichter Meta-Daten nach ihren Wünschen zu beeinflussen, ohne dabei die Namen von HTTP-Headern genau zu wissen. Weiterhin übernimmt BPEL4REST weitere Aufgaben, wie beispielsweise die Berechnung des Authentifizierungs-Headers und kümmert sich um die Serialisierung und Deserialisierung von gesendeten bzw. empfangenen Daten.
Ein weiterer Unterschied ist, dass BPEL4REST nur die Orchestrierung von RESTful Web Services ermöglicht. Nicht aber die Veröffentlichung von BPEL-Prozessen als REST-Service. Diese Entscheidung wurde getroffen, weil BPEL sehr eng an WSDL 1.1 gekoppelt ist. Die Motivation hinter BPEL4REST ist, es überhaupt erst möglich zu machen, mit RESTful Web Services zu arbeiten. Die Funktionalität einen BPEL-Prozess zu veröffentlichen, so dass er von anderen BPEL-Prozessen konsumiert werden kann, ist aber bereits in BPEL enthalten. Die Veröffentlichung als REST-Service ist deshalb redundant und erhöht die Komplexität ohne einen entscheidenden Vorteil zu bringen.
Pautasso erweitert BPEL nicht nur um weitere Konstrukte, sondern ändert auch die Semantik von existierenden Elementen. Um diese Erweiterungen zu realisieren muss eine existierende

BPEL-Engine erweitert werden. BPEL4REST nutzt den Mechanismus der **extensionActivity**, der bereits in BPEL vorhanden ist, was dazu führt, dass keine Änderungen an einer BPEL-Engine selbst durchgeführt, sondern nur zusätzliche Funktionalität implementiert werden muss.

6.2 BPEL light

Der Konferenzbeitrag „BPELlight“ ([NLKL07]) von Jörg Nitzsche, Tammo van Lessen, Dimka Karastoyanova und Frank Leyman, behandelt die Idee eines WSDL-losen BPEL. Dazu erweitern die Autoren BPEL um neue „WSDL-lose“ Sprach-Elemente, analog zu den WSDL-basierten Aktivitäten und Partnerlinks.
[NLKL07] identifiziert zunächst zwei wichtige Defizite von BPEL: Zum einen die eingeschränkte Wiederverwendbarkeit von (Teilen von) Prozessen, zum anderen die mangelnde Flexibilität in Bezug auf Schnittstellen, die sie beide von der engen Kopplung von WSDL und BPEL ableiten.
BPELlight definiert ein neues Interaktionsmodell durch zwei neue Elemente namens **conversation** und **interactionActivity**. Das **conversation** Element bildet das WSDL-lose Äquivalent zum **partnerLink**. Die **interactionActivity**, ist von den Autoren so konzipiert, dass sie alle BPEL-eigenen Interaktions-Aktivitäten darstellen kann. Dazu benötigte Informationen, werden durch Attribute des **interactionActivity** Elements bereitgestellt.
Weiterhin führt [NLKL07] ein neues Element ein, das bereits, in ähnlicher Form, in BPEL 1.1 vorhanden war, in BPEL 2.0 aber nicht mehr. Das **partner** Element erlaubt es verschiedene **conversation** Elemente zu gruppieren und einem bestimmten Partner zuzuordnen.
Zuletzt definiert BPELlight noch eine Erweiterung der Assign-Aktivität, die es erlaubt eine Partner-Identifikation in das **partner** Element zu kopieren. Dazu wird das **to** Element um ein **partner** Attribut erweitert.
In ihrer Beurteilung schreiben die Autoren, dass BPELlight die Prozesslogik von den Schnittstellenbeschreibungen trennt. Die Schnittstellen werden separat von BPELlight definiert und an Prozess-Aktivitäten gebunden. Weiterhin können die Schnittstellen auch in jeder IDL[7] beschrieben werden.

6.3 Towards Resource-Oriented BPEL

Der Artikel „Towards Resource-Oriented BPEL“ ([Ove]) von Hagen Overdick befasst sich damit, BPEL zur Modellierung von Ressourcen zu erweitern.
Overdick befasst sich zunächst mit der Beschreibung der einheitlichen Schnittstelle[8] von HTTP, indem er die Verben GET, PUT, POST und DELETE erklärt. Anschließend zeigt er anhand eines komplexeren Beispiels, wie diese Schnittstelle ressourcen-orientiert eingesetzt

[7] Interface Description Language
[8] Uniform Interface

werden kann.
Laut Overdick hat BPEL zwar keine expliziten Möglichkeiten Ressourcen-Status zu modellieren, dafür aber durch das **scope** Element eine implizite. Er beschreibt, dass eine Statustransition durch POST-Nachrichten ausgelöst werden kann. GET-, PUT- und DELETE-Nachrichten können nach Overdick jedoch beliebig oft verarbeitet werden, aufgrund ihrer spezifizierten Eigenschaften Sicherheit(nur GET) und Idempotenz. Er schlägt also vor BPEL-Prozesse zur Ressourcen-Modellierung einzusetzen, indem Prozesse als Ressourcen angesehen werden. Der Status dieser Ressourcen wird durch Scopes modelliert. Die Interaktion mit der Umgebung find durch **Event Handler** statt, die auf GET, PUT und DELET reagieren. Weiterhin schlägt Overdick einen „POST Empfänger" vor, der bei Empfangen von POST-Anfragen eine Status-Transition durchführen kann. Das von Overdick vorgeschlagene Prinzip ist in Abbildung 6.2 dargestellt. Overdicks Ansatz ist mehr oder weniger mit Pautassos Arbeit vergleichbar, da auch Pautasso Prozesse nach einem Ähnlichen Prinzip als Ressource zur Verfügung stellt.

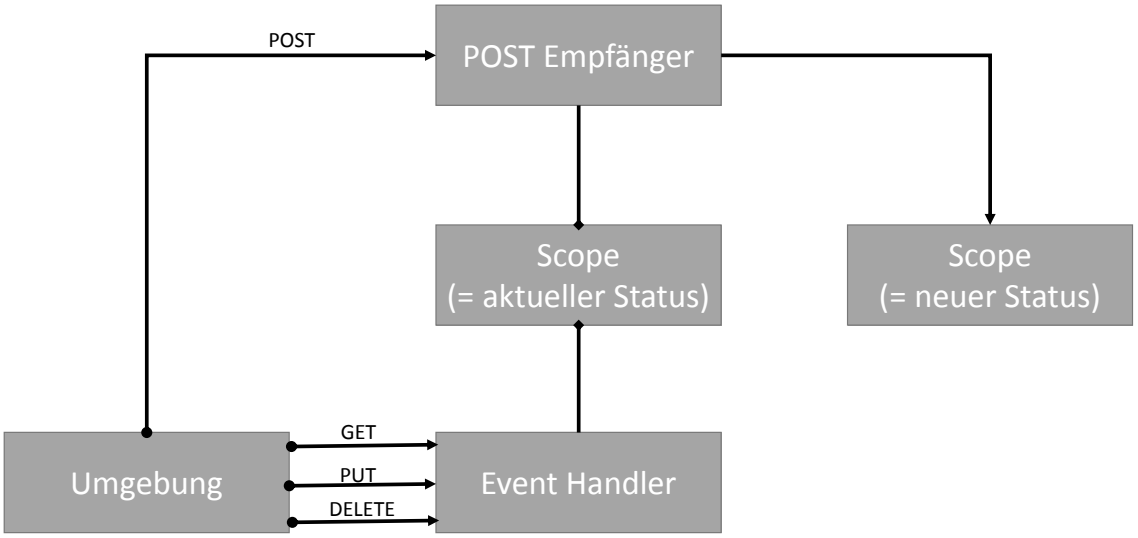

Abbildung 6.2: Wie BPEL zur Modellierung von Ressourcen-Status genutzt werden kann, nach [Ove, Abbildung 3.]

6.4 Bite: Workflow Composition for the Web

Der Artikel „Bite: Workflow Composition for the Web" ([CDKL]) definiert die Sprache **Bite**. Bite ist eine Sprache zur Komposition von Workflows in einer REST-basierten Umgebung.

Ziel der Autoren ist es, die Sprache so zu entwerfen, dass Prozess-Instanzen, als Ressource angeboten werden und dass diese Prozess-Instanzen auch mit anderen Ressourcen interagieren können. Die Funktionalität ist also letztendlich analog zu BPEL, mit dem Unterschied, dass Bite auf eine REST-basierte Umgebung ausgelegt ist. Weiterhin erlaubt es Bite seine Variablen mit **content-types** zu assoziieren. Dadurch wird die Content-Negotiation in Bite automatisiert und vereinfacht.
Wie BPEL kann auch Bite um neue Aktivitäten erweitert werden und liefert dazu auch einen Mechanismus, wie diese neuen Aktivitäten implementiert und registriert werden können.
Im Gegensatz zu BPEL gibt es in Bite keine Scopes und keine FaultHandler. Von den strukturellen BPEL-Aktivitäten sind lediglich die Elemente **while** und **pick** in Bite übernommen worden. Um ein leichtgewichtiges Prozessmodell zu realisieren versucht Bite „Workflow scripting“ zu ermöglichen. Damit sind hauptsächlich folgende Konventionen gemeint:

- **„use implies definition“:** Damit is gemeint, dass Variablen auch genutzt werden können, ohne sie vorher zu deklarieren oder explizit zu typisieren.
- **„convention over configuration“:** Bite gibt vor, dass Rückgabewerte von Aktivitäten durch implizit typisierte Variablen, mit dem selben Namen wie die Aktivität, realisiert werden.
- **„Radical reduction of extraneous constructs while eliminating levels of indi- rection“:** Aufrufziele von Aktivitäten werden als URL oder als Variablen definiert. Es wird keine Typisierung der Ressourcen benötigt. Das steht im radikalen Gegensatz zu BPEL, bei dem Aufrufziele durch Partnerlinktypen definiert sein müssen, die deutlich mehr Informationen benötigen.

Zuletzt ermöglicht Bite noch eine flexible Konfiguration von Prozessen, indem es erlaubt, Werte von Variablen auch außerhalb der Workflow-Definition zu setzen. Dies ist ähnlich zu dem Konzept von Properties in java.

7 Zusammenfassung und Ausblick

BPEL4REST ermöglicht es der Sprache BPEL, nicht nur WSDL-basierte, sondern auch RESTful Web Services zu orchestrieren. Im Verlauf der Arbeit wurden zunächst wichtige Begriffe erläutert, die zum Verständnis der Arbeit wichtig sind. Diese sind: Geschäftsprozesse, Workflows, Workflow-Management-Systeme, BPEL und REST.
Anschließend wurden Möglichkeiten und Ideen aufgezeigt inwiefern es bereits möglich ist, oder möglich wäre, RESTful Web Services durch BPEL zu orchestrieren. Diese Ideen wurden dann anhand von Anforderungen, die aus den REST-Prinzipien abgeleitet sind, analysiert und bewertet.
Bei der Analyse wurde festgestellt, dass vor allem die erste Anforderung, zur Laufzeit eines BPEL-Prozesses die URI eines HTTP-Aufrufs zu manipulieren, durch durch die vorgestellten Ansätze nicht ausreichend erfüllt wird.
Die Nutzung von **WSDL 1.1** (siehe Kapitel 3.2.1) oder der **WSDL 1.1 Extension for REST** (siehe Kapitel 3.2.2) zur Beschreibung von REST Schnittstellen ist dadurch zur Orchestrierung von RESTful Web Services durch BPEL **nicht geeignet**.
Weiterhin wurde auf **WSDL 2.0** (siehe Kapitel 3.2.3) eingegangen. Diese Möglichkeit scheitert an der Tatsache, dass BPEL sehr eng an WSDL 1.1 gekoppelt ist. Es ist unwahrscheinlich, dass BPEL zukünftig auch WSDL 2.0 unterstützt, da das OASIS Kommittee seine Arbeit an BPEL als abgeschlossen erachtet.
Ebenfalls wurde noch ein Konzept vorgestellt, bei dem ein WSDL 1.1 Web Service die Kommunikation mit RESTful Web Services übernimmt (siehe Kapitel 3.2.4). Diese Möglichkeit hat aber den Nachteil, dass die Komplexität von BPEL-Prozessen erhöht wird, weil viele zusätzliche Aktivitäten in BPEL nötig sind, die Nachrichten zu verarbeiten, die von diesem Web Service empfangen werden.

Im folgenden Kapitel wurde ein Konzept entworfen, das mit Hilfe eines existierenden BPEL-Konstrukts, eine Erweiterung für BPEL definiert. Diese Erweiterung verfolgt das Ziel, allen Anforderungen, RESTful Web Services durch BPEL zu orchestrieren, zu entsprechen.
Kern des Konzepts ist es, die Struktur einer **extensionActivity** [TC07, 10.9] zu definieren, die alle geforderten Funktionalitäten abbildet. Die Erweiterung beherrscht 5 HTTP-Verben: GET, PUT, POST, DELETE und HEAD.
Zur vereinfachten Anwendung von HTTP-Headern, definiert die Erweiterung das **context** Element (siehe 4.3.4), mit dem es möglich ist, allgemeine Header-Felder für mehrere Aufrufe zu setzen.
Weiterhin wurden XML-Elemente eingeführt, die die HTTP-Content-Negotiation unterstützen. Ebenfalls definiert die Erweiterung XML-Elemente, die es erleichtern, BPEL-Variablen mit den korrekten Medientypen zu assoziieren. Mit dem neu definierten **responseHeader** Element, wurde die Möglichkeit geschaffen, alle empfangenen Meta-Daten einer HTTP-

Antwort in eine BPEL-Variable zu schreiben und sie somit für BPEL-Prozesse verfügbar zu machen.
Durch die anschließende Evaluierung des Konzepts, wurde festgestellt, dass die Erweiterung alle gestellten Anforderungen erfüllt, wenngleich eine Form der HTTP-Content-Negotiation, die „agent-driven Content-Negotiation“, trotzdem „von Hand“ durchgeführt werden muss.

Im Kapitel **BPEL4REST - Implementierung** wurde die prototypische Implementierung des Konzepts dokumentiert. Dazu wurden zunächst Technologien vorgestellt, die zur Entwicklung dieser Erweiterung eingesetzt wurden.
Im weiteren Verlauf des Kapitels wurde zunächst die grobe Architektur, die aus den beiden Komponenten „Model“ und „Control“ besteht, dargestellt und erläutert. Anschließend folgten detaillierte Beschreibungen der Komponenten und ihrer einzelnen Bestandteile, sowie eine Beschreibung des internen Ablaufs der Erweiterung.
Zuletzt wurden in diesem Kapitel noch die Einschränkungen beschrieben, denen die Implementierung von BPEL4REST unterliegt.

Das Kapitel 6, befasste sich mit verwandten Arbeiten. Dazu gehört die Arbeit mit dem Titel „RESTful Web service composition with BPEL for REST“ von Cesare Pautasso, die sich mit der Komposition von RESTful Web Services in BPEL beschäftigt.
Die Arbeit „RESTful Web service composition with BPEL for REST“ (siehe Kapitel 6.1) schlägt eine Erweiterung von BPEL vor, geht dabei aber noch einen Schritt weiter als BPEL4REST. Pautasso definiert zum Einen neue Aktivitäten, zur Orchestrierung von RESTful Web Services, zum Anderen definiert er aber auch Konstrukte, die es ermöglichen, Teile eines BPEL-Prozesses als RESTful Web Service zu veröffentlichen. Da Pautassos Vorschlag signifikante Änderungen an BPEL bedeuten, sodass Änderungen an BPEL-Engines nötig wären, macht er auch einen Vorschlag, wie die Architektur, einer für REST erweiterten BPEL-Engine, aussehen kann.
BPEL light beschreibt ein Konzept die Kopplung von WSDL und BPEL zu lösen, während die Arbeit [Ove] von Overdick einen Vorschlag macht, BPEL-Prozesse als Ressourcen zu veröffentlichen, ohne die Orchestrierung von RESTful Web Services mit einzubeziehen.
Mit der Sprache **Bite** wird in der Arbeit [CDKL] eine an BPEL-angelehnte Workflow-Sprache definiert. Bite ist eine leichtgewichtige Sprache mit „Scripting Konventionen“, die es ermöglicht, Prozesse als Ressourcen zu veröffentlichen, aber auch dass Prozesse mit anderen Ressourcen interagieren können.

Ausblick

In dieser Bachelorarbeit wurde BPEL4REST entworfen und implementiert: Eine Erweiterung für BPEL, die das **extensionActivity** Element nutzt, um BPEL zur Orchestrierung von RESTful Web Services zu befähigen.
Die Implementierung von BPEL4REST ist als Prototyp anzusehen und bietet daher noch viele Möglichkeiten zur Verbesserung:

- Zur frühzeitigen Vermeidung von Fehlern, in der Ausführung von BPEL4REST, kann eine Validierung des übergebenen XML-Elements gegen ein XML Schema implementiert werden. Damit verbunden kann ein automatisiertes Parsing des übergebenen XML-Elements implementiert werden, indem beispielsweise Model-Klassen aus dem XML Schema generiert werden.
- Derzeit beherrscht BPEL4REST den Umgang mit den Medientypen **text/plain**, **application/json** und allen **XML-Typen**. Weitere Medientypen, die noch nicht von BPEL4REST untertützt werden, sind beispielsweise **CSV-Typen** oder **text/html**.

BPEL4REST wurde für die OpenSource BPEL-Engine **Apache ODE** implementiert, ist aber derzeit nur mit dem **Experimental Branch** nutzbar. Laut [odeb] sind keine weiteren Veröffentlichungen dieses Branches geplant. Allerdings sollen Features dieses Branches in die 1.x Veröffentlichungen der ODE integriert werden.
Da das **extensionActivity** Element Teil der Spezifikation von BPEL ist, ist anzunehmen, dass auch die Funktionalität, die von BPEL4REST genutzt wird, irgendwann in den aktuellen Veröffentlichungen der ODE enthalten sein wird.

Literaturverzeichnis

[apa] The Apache Software Foundation. URL http://www.apache.org/. (Zitiert auf den Seiten 45 und 46)

[bpea] OASIS Web Services Business Process Execution Language (WSBPEL) TC. URL https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel. (Zitiert auf Seite 31)

[bpeb] WS-BPEL FAQ. URL https://www.oasis-open.org/committees/download.php/23858/. (Zitiert auf Seite 31)

[CCMW01] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana. Web Services Description Language (WSDL) 1.1, 2001. URL http://www.w3.org/TR/wsdl. (Zitiert auf den Seiten 16, 22, 24 und 28)

[CDKL] F. Curbere, M. Duftler, R. Khalaf, D. Lovell. Bite: Workflow Composition for the Web. (Zitiert auf den Seiten 61 und 64)

[Coa] W. M. Coalition. The Workflow Reference Model. URL http://www.wfmc.org/reference-model.html. (Zitiert auf Seite 10)

[Dav93] T. Davenport. *Process Innovation: Reengineering Work Through Information Technology*. Harvard Business School Press, Boston, 1993. (Zitiert auf Seite 13)

[ecla] Eclipse. URL http://www.eclipse.org/. (Zitiert auf den Seiten 46 und 55)

[eclb] Eclipse Downloads. URL http://www.eclipse.org/downloads/. (Zitiert auf Seite 46)

[eclc] Older Versions Of Eclipse. URL http://wiki.eclipse.org/Older_Versions_Of_Eclipse. (Zitiert auf Seite 46)

[Fieoo] R. T. Fielding. Architectural Styles and the Design of Network-based Software Architectures, 2000. URL http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm. (Zitiert auf den Seiten 3, 10, 17, 18, 19 und 21)

[GWSS] P. D. R. L. GABLER WIRTSCHAFTSLEXIKON, D. M. Siepermann, P. D. G. Schewe. Geschäftsprozess. URL http://wirtschaftslexikon.gabler.de/Archiv/5598/geschaeftsprozess-v10.html. (Zitiert auf Seite 13)

[HC93] M. Hammer, J. Champy. *Reengineering the Corporation: A Manifesto for Business Recolution*. Harper Business, 1993. (Zitiert auf Seite 13)

[htta] Apache HttpComponents. URL http://hc.apache.org/. (Zitiert auf Seite 46)

[httb] HttpComponents Downloads. URL http://hc.apache.org/downloads.cgi. (Zitiert auf Seite 46)

[htt99] Hypertext Transfer Protocol – HTTP/1.1, 1999. URL https://tools.ietf.org/html/rfc2616. (Zitiert auf den Seiten 27, 31, 34, 36, 37, 39, 40 und 42)

[jop] JOpera for Eclipse. URL www.jopera.org. (Zitiert auf Seite 55)

[LLN11] T. van Lessen, D. Lübke, J. Nitzsche. *Geschäftsprozesse automatisieren mit BPEL*. dpunkt.verlag GmbH, 2011. URL http://www.bpelbuch.de. (Zitiert auf Seite 17)

[LR00] F. Leymann, D. Roller. *Production Workflow Concepts and Techniques*. Prentice Hal, Inc, 2000. (Zitiert auf den Seiten 7, 14, 15 und 16)

[Mic08] S. Microsystems. Using the HTTP Binding Component, 2008. URL http://docs.oracle.com/cd/E19182-01/820-0595/cnfg_http-bc-get-processing_r/index.html. (Zitiert auf Seite 24)

[NLKL07] J. Nitzsche, T. van Lessen, D. Karastoyanova, F. Leymann. BPEL light. In *5th International Conference on Business Process Management (BPM 2007)*. Springer, 2007. (Zitiert auf Seite 60)

[odea] Apache ODE. URL http://ode.apache.org/. (Zitiert auf Seite 45)

[odeb] Getting ODE. URL http://ode.apache.org/getting-ode.html. (Zitiert auf Seite 65)

[Ove] H. Overdick. Towards Resource-Oriented BPEL. (Zitiert auf den Seiten 7, 60, 61 und 64)

[Pau08] C. Pautasso. BPEL for REST. *Proc. of the 6th International Conference on Business Process Management (BPM 2008), Milan, Italy*, 2008. (Zitiert auf Seite 55)

[Pau09] C. Pautasso. RESTful Web service composition with BPEL for REST. *Data and Knowledge Engineering, Volume 68, Issue 9*, 2009. (Zitiert auf den Seiten 7, 8, 55, 57, 58 und 59)

[TC07] O. W. S. B. P. E. L. W. TC. Web Services Business Process Execution Language Version 2.0, 2007. URL http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html. (Zitiert auf den Seiten 3, 17, 33 und 63)

[tom] Tomcat 7 Downloads. URL http://tomcat.apache.org/download-70.cgi. (Zitiert auf Seite 45)

[uri99] HTML 4.01 Specification, 1999. URL http://www.w3.org/TR/html401. (Zitiert auf Seite 23)

[wsd] WSDL 1.1 Extensions for REST. URL ode.apache.org/extensions/wsdl-11-extensions-for-rest.html. (Zitiert auf den Seiten 7, 24, 25, 26, 27 und 28)

[wsd06a] Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language, 2006. URL http://www.w3.org/TR/2006/CR-wsdl20-20060327/. (Zitiert auf den Seiten 28 und 31)

[wsd06b] Web Services Description Language (WSDL) Version 2.0 Part 2: Adjuncts, 2006. URL http://www.w3.org/TR/2006/CR-wsdl20-adjuncts-20060327/. (Zitiert auf den Seiten 7, 28 und 30)

Alle URLs wurden zuletzt am 01. 08. 2013 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift