Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D – 70569 Stuttgart

Diplomarbeit Nr. 3096

Iteration und wiederholte Ausführung von Aktivitäten in Workflows

Bo Ning

Studiengang:	Informatik
Prüfer:	Jun.-Prof. Dr.-Ing. Dimka Karastoyanova
Betreuer :	Dipl.-Inf. Mirko Sonntag
Begonnen am:	01.September 2010
Beendet am:	13.März 2011
CR-Klassifikation:	H.4.1 Workflow-Management

Meinem Mann und meiner Tochter, Jindong und Shuning

zum Dank für eure liebevolle Unterstützung,
derer ich mir zu jeder Zeit sicher sein konnte.

Meinem Betreuer, Dipl.-Inf. Mirko Sonntag

zum Dank für Ihre Hilfe und Betreuung.

Jun.-Prof. Dr.-Ing. Dimka Karastoyanova

zum Dank für Ihre Zusage, dass ich die Diplomarbeit schreiben darf.

Inhaltsverzeichnis

Abbildungsverzeichnis

Verzeichnis der Listings

Tabellenverzeichnis

Abkürzungsverzeichnis

BPEL	Business Process Execution Language
DU	Deployment Unit
EPR	Endpoint Reference
FK	Fremd Key
HTTP	Hypertext Transfer Protocol
IT	Informationstechnologie
JACOB	Java Concurrent Objects
JMS	Java Message Service
JPA	Java Persistence API
MOM	Message-oriented Middleware
OASIS	Organization for the Advancement of Structured Information Standards
ODE	Orchestration Director Engine
OMG	Object Management Group
ORM	Object-Relational Mapping
SOA	Service-orientierte Architektur
UDDI	Universal Description, Discovery, and Integration
W3C	World Wide Web Consortium
WfMS	Workflow Management System
WS-BPEL	Web Services Business Process Execution Language
WSDL	Web Services Description Language
XML	Extensible Markup Language
YAWL	Yet Another Workflow Language

1. Einleitung

Auf der einen Seite sind die neuen Sprachen für Workflow nach den Bedürfnissen von den Geschäftsprozessen entworfen worden, die auf einer eindeutigen Welt basieren und diese Welt bezieht sich auf die ebenfalls eindeutigen Dateien. Aber im Vergleich zu den Geschäftsprozessen, laufen die echten Prozesse auf einer wahren Welt und müssen mit der Ungewissheit und auch der Instabilität über die Durchführungsumgebung rechnen. Deshalb müssen die neuen Modellierungselemente entworfen werden, die die Fähigkeiten haben, solche Probleme von Ungewissheit und Instabilität lösen zu können. Es gibt viele Faktoren, die die obengenannten Probleme verursachen können. Zur Zeit laufen sehr viele Geschäftsprozess innerhalb eines Netzwerks, wenn die Verbindungen zum Netzwerk plötzlich nur für kurze Zeit unterbrochen sind, können die Geschäftsprozesse falsche Nachrichten erhalten und zu fehlerhaften Ergebnissen führen, weil die Geschäftsprozesse sich nicht auf die neusten Situationen reagieren können [EKU+10].

Auf der anderen Seite können die traditionellen Workflow-Technologien und Prinzipien der Service-orientierten Architektur (SOA) angewandt werden, um die Wissenschaftler in ihren Experimenten zu unterstützen [SK10]. Die Workflow-Eigenschaften legen die Anforderungen an die Wissenschaftler fest. Die Unterstützung von IT-Systemen kann den Wissenschaftlern bei den Experimenten, den Berechnungen, und auch den Simulationen helfen. Solche Eigenschaften helfen die Wissenschaftlern, die Experimente zu entwerfen, durchzuführen, zu überwachen, zu analysieren, die Ergebnisse zu verteilen und wieder darzustellen, und auch den Ansatz von dem Trial-and-Error-Verfahren zu verfolgen, das ein typischer Ablauf in E-Science ist [SK10].

1.1 Motivation

Die Workflow-Technologien sind wegen der folgenden Ursachen im wissenschaftlichen Umfeld noch nicht ganz eingeführt worden. Die Hauptursache ist, dass die Definitionen von Workflow innerhalb des geschäftlichen und wissenschaftlichen Bereichs unterschiedlich sind. Es gibt ein Bedürfnis, die Definition von den zwei Gemeinden abzustimmen und abzuklären [SK10]. Die andere Ursache sind die fehlenden Eigenschaften, die von der Wissenschaft für die vollständige und intuitive Unterstützung bei den wissenschaftlichen Simulationen, komplexen Berechnungen und Experimenten unbedingt benötigt werden [SK10]. So muss ein neues Workflow Management System (WfMS) entwickelt werden, das auf traditionellen Workflow-Technologien basiert und die Bedürfnisse der Wissenschaftler und der wissenschaftlichen Anwendungen erfüllen kann.

1.2 Aufgabendefinition

Weil die wissenschaftlichen Experimente in einem Trial-and-Error-Verfahren nur unzureichend durch existierende Workflow-Konzepte ermöglicht werden, muss ein Teil des Workflows für die Konvergenz von Ergebnissen oder für die Reaktion auf Fehler erneut ausgeführt werden können.

Ziel dieser Arbeit soll ein Konzept für die wiederholte Ausführung von bereits abgelaufenen Workflow-Teilen erstellt werden. Zwischen der Iteration der Workflow-Teile und ihrer Wiederholung soll unterschieden werden. Die Operation „Iteration“ verhält sich wie eine

Schleife. Bei der Wiederholung muss der zu wiederholende Teil erst rückgängig gemacht werden. Beide Operationen sollen an jeder beliebigen Stelle im Workflow durch einen manuellen Eingriff eines Wissenschaftlers realisiert werden. Das Konzept soll auch durch eine prototypische Implementierung in Apache ODE gezeigt werden.

1.3 Aufbau der Arbeit

Die vorliegende Arbeit ist in folgender Weise gegliedert:

Kapitel 2 - Grundlagen: Hier wird zunächst SOA kurz vorgestellt. Im Anschluss daran werden die Webservices und als letztes BPEL vorgestellt. Alle Inhalte in diesem Kapitel sind die Informationen von Technologien, die in dieser Arbeit benötigt sind.

Kapitel 3 - Dieses Kapitel beschäftigt sich mit dem grundlegenden Aufbau und auch den wichtigen Komponenten der Apache ODE.

Kapitel 4 - In diesem Kapitel werden drei relevante Arbeiten vorgestellt.

Kapitel 5 - Dieses Kapitel beschreibt das Konzept von dieser Arbeit. Dabei werden einige Beispielfälle vorgestellt.

Kapitel 6 - Hier wird die prototypische Implementierung der Funktionen Iterate und Reexecute an der Apache ODE vorgestellt.

Kapitel 7 - Diese Diplomarbeit wird mit diesem Kapitel abgeschlossen.

2. Grundlagen

Bei diesem Kapitel handelt sich um einen Überblick über die Grundlagen dieser Arbeit. Dazu werden die für diese Arbeit relevanten Teile von SOA, Webservices und BPEL vorgestellt.

2.1 Service-orientierte Architektur

Der Begriff „Service-orientierte Architektur“ (SOA) wurde 1996 von dem Marktforschungsunternehmen Gartner erstmalig genutzt [RNS96]. Es gibt keine allgemein akzeptierte Definition von SOA. Die Service-orientierte Architektur kann als ein architektonisches Konzept betrachtet werden, das die verschiedenartigen Systeme ermöglicht, die reichhaltigen Geschäftskommunikationen zu integrieren [WCL+05]. Service-orientierte Architektur kann auch als ein Architekturmuster der Informationstechnik aus dem Bereich der verteilten Systeme betrachtet werden, um Dienste von IT-Systemen zu strukturieren und zu nutzen [WSOA]. Dennoch wird häufig die Definition von der OASIS aus dem Jahr 2006 zitiert:

„SOA ist ein Paradigma für die Strukturierung und Nutzung verteilter Funktionalität, die von unterschiedlichen Besitzern verantwortet wird [SOA06].“

2.1.1 Einführung von SOA

Die wichtigsten Elemente von einer Service-orientierten Architektur können als ein Tempel arrangiert werden [Mel10]. Um eine SOA definieren zu können, werden die Merkmale in Anlehnung an [Dos05] vorgestellt.

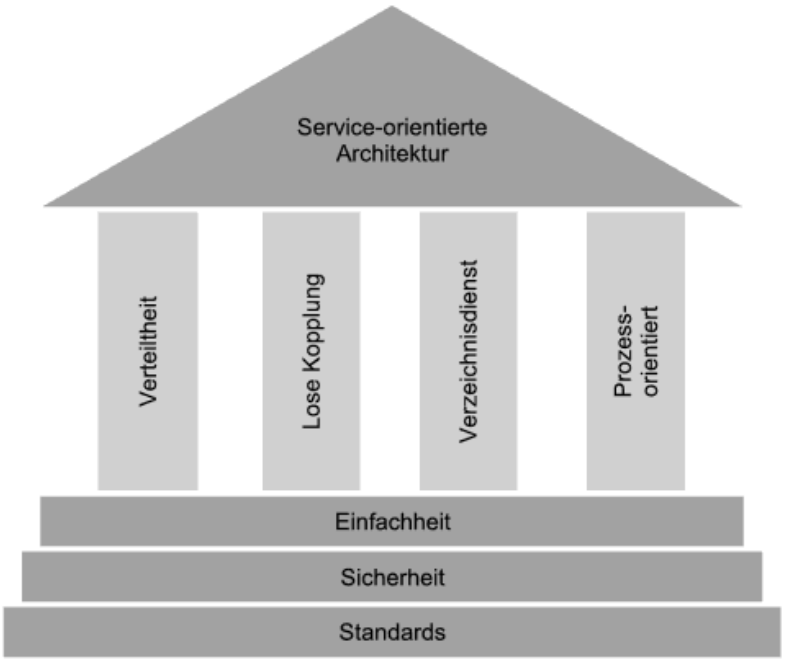

Abbildung 2.1 SOA als ein Tempel [Mel10]

Das Dach vom Tempel:

Das Dach von diesem Tempel ist dann die SOA und sie hat die vier unterstützenden Säulen und auch die drei Stufen. Alle Säulen und Stufen werden gleich vorgestellt.

Die Säulen vom Tempel:

Verteiltheit - Die SOA ermöglicht, die Dienste über das Netz zu verteilen. Eine SOA unterstützt jede moderne Architektur, d.h. SOA ist unabhängig von Plattform und die von System unabhängigen Dienste können dadurch zur Verfügung gestellt werden.

Lose Kopplung - Dienste können von Anwendungen oder anderen Diensten bei Bedarf dynamisch gesucht, gefunden und eingebunden werden [Mel10]. Die lose Kopplung kann auf einer Seite die Interoperabilität zwischen den interagierenden Partner verbessern. Auf der anderen Seite kann die lose Kopplung die Dienste stabil machen, wenn ein von den Partnern benutzter Service sich geändert hat. Eine lose Kopplung ermöglicht dann die Wiederverwendung von Diensten [Son08].

Verzeichnisdienst - Um die dynamischen Aufrufe von Diensten zu ermöglichen, müssen diese Dienste zunächst gesucht und dann gefunden werden. Durch einen Verzeichnisdienst kann ein bestimmter Dienst erhältlich sein, der schon im Verzeichnisdienst registriert ist. Es kann nicht nur dem Verzeichnisdienst sondern auch dem Repository zur Verfügung gestellt werden. Der Unterschied zwischen den Beiden liegt darin, dass Repository nicht nur den Verweis auf diese Metadaten hat, sondern auch die Daten über die Dienste. Der Verzeichnisdienst hat dagegen nur den Verweis auf diese Metadaten [Mel10].

Prozessorientert - Eine Service-orientierte Architektur hat keine Beschränkungen bezüglich der Komplexität eines Dienstes [Int06]. Das hat Vorteile für den Dienstanbieter und Dienstnutzer. Auf der Seite von Dienstanbieter kann er einen Dienst schnell und mit geringerem Aufwand entwickeln. Der Dienstnutzer kann dann solch einen Dienst schnell benutzen. Prozesse erlauben die Orchestrierung von einem Dienst oder mehreren Diensten, die in neue Dienste auf einer höheren Ebene umgewandelt werden. Es führt zu einem rekursiven Aggregationsmodell: ein Dienst kann aus anderen Diensten bestehen und kann auch in einem höheren Dienst sein [Son08].

Die Stufen vom Tempel:

Standards - Offene Standards bieten die Investitionssicherheit [Int06]. Um einen Dienst zu benutzen, muss der Dienstnutzer die Fähigkeiten haben, mit den Diensten zu kommunizieren. Das bedeutet, dass die Schnittstelle von den Diensten in einer maschinenlesbaren Form beschrieben werden muss. Ohne die offenen Standards ist es leider nicht realisierbar, dass ein Dienstnutzer den Dienst von einem nicht bekannten Dienstanbieter verstehen kann.

Einfachheit - Die Einfachheit bezieht sich nicht auf die Technologie, weil diese durchaus nicht immer einfach sind. Stattdessen bezieht sie sich auf die Anwendungen von SOA, die durch die Methode der Automatisierung und durch die Unterstützung von Werkzeugen vereinfacht werden können [Son08]. Die Einfachheit kann viele Anforderungen erfüllen und eine schnelle Umsetzung der Anwendung einer SOA ermöglichen.

Sicherheit - Die Sicherheit ist die wichtigste Aufgabe von SOA. Die Sicherheit bedeutet die

Authentifikation, die Integrität, die Vertraulichkeit usw. Sie ist sehr kritisch für den Erfolg von SOA bei Geschäftsanwendungen.

2.1.2 SOA-Dreieckmodell

Die grundlegenden Prinzipien von der Arbeitsweise der SOA können in einem SOA-Dreieckmodell dargestellt werden (siehe Abbildung 2.2).

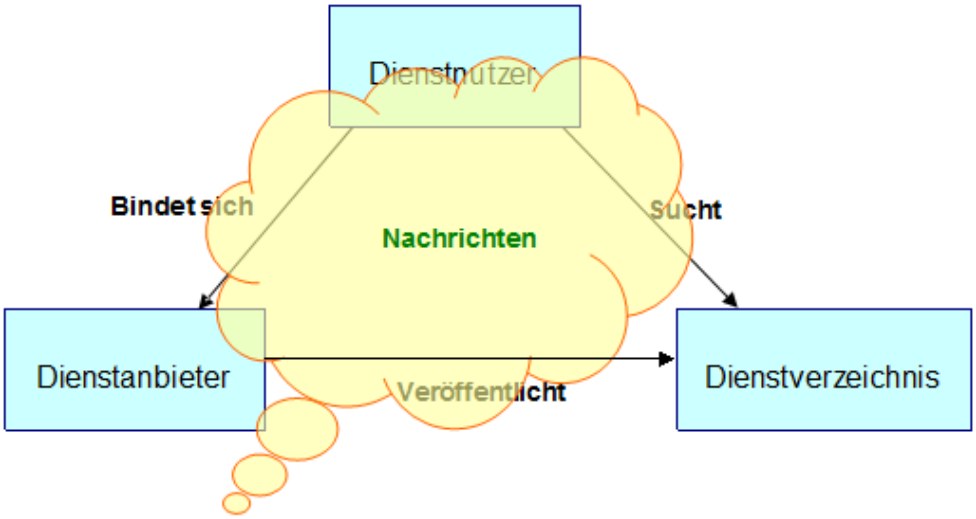

Abbildung 2.2 SOA-Dreieckmodell

- Dienst (Service)

Die Definition des Dienstes kommt ursprünglich aus den Dienstleistungen, dies hat zur Folge, dass es Unterschiede zwischen Produkten und Diensten gibt. [Mas07] Der Dienst ist im SOA ein eigenständiges Softwareelement, z.B. ein Programm oder auch eine Softwarekomponente und kann lokal oder über ein Netzwerk durch Nachrichten-basierte Kommunikation (z.B. SOAP) von anderen aufgerufen werden. Der Dienst bietet die Funktionen nach außen an. Der Dienst soll in einer von Maschinen lesbaren Form geschrieben sein und auf ihn kann nur über die vorgeschriebene Schnittstelle zugegriffen werden.

- Dienstanbieter (Service Provider)

Der Dienstanbieter hat die Verantwortungen, die von ihm angebotenen Dienste bereitzustellen und auch die Verfügbarkeit dieser Dienste sicherzustellen. Er muss nicht alle angebotenen Dienste eigenständig implementieren, wenn er über das Netz mehrere kleine, einfache Dienste in einem neuen und umfangreicheren Dienst zusammensetzt. Der Betrieb, die Datensicherung und die Wartung von den Diensten gehören auch zu den Aufgaben des Dienstanbieters. Der Dienstanbieter registriert seine Dienste bei einem Dienstverzeichnis. Im Dienstverzeichnis kann der Dienstnutzer die benötigten Dienste finden und sie anschließend nutzen. Der Dienstanbieter muss auch die Authentifisierung ermöglichen, um zu überprüfen, ob ein Dienstnutzer berechtigt ist, die Dienste zu benutzen.

- Dienstnutzer (Service Consumer)

Der Dienstnutzer kann an dieser Stelle direkt mit dem Klienten in einer traditionellen Client-

Server-Architektur verglichen werden [Mel10]. Es ist für den Dienstnutzer sehr wichtig, dass sein gewünschter Dienst zu finden und zu liefern ist. Wenn der Dienstnutzer einen bestimmten Dienst braucht, wird er den Dienst zuerst im Dienstverzeichnis suchen. Wenn seine Suche erfolgreich ist, verbindet er sich dann über ein Protokoll mit dem Dienstanbieter und das Protokoll muss den beiden bekannt sein. Wer den Dienst benutzt und wer den Dienst zur Verfügung stellt, ist nicht wichtig.

- Dienstverzeichnis (Service Registry)

Das Dienstverzeichnis vermittelt zwischen dem Dienstanbieter und dem Dienstnutzer. Es ist verantwortlich dafür, dass die Dienste des Dienstanbieters in das Verzeichnis eingetragen werden können und die eingetragenen Dienste auch im immer aktiven und verfügbaren Zustand sind. Mit UDDI kann der richtige Dienst gefunden werden, und damit können auch die ausführlichen Informationen über den Dienst erhältlich sein. UDDI ermöglicht es den Dienstanbietern die Implementierungen von ihren Diensten zu veröffentlichen und ermöglicht es auch den Dienstnutzer den richtigen Dienstanbieter zu finden, der die besten Dienste nach den Bedürfnissen des Dienstnutzers anbietet [WCL+05].

2.2 Webservices

Webservices sind die am meisten verwendete Technologie, um SOA, ein abstraktes Architekturmodell, zu implementieren. Um Webservices einheitlich vorzustellen, werde ich den Begriff anhand der Definition vom W3C erläutern.

Die Definition von Webservices lautet nach dem W3C wie folgt:

„ A Web Service is a software system designed to support interoperable machine- to-machine interaction over a network. It has an interface described in a machine-processable format(specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards [W3].“

Webservices ermöglichen eine standardisierte direkte Interaktion mit anderer Software unter der Verwendung der Nachrichten in Form von XML über das Netz. Dabei wird das HTTP-Protokoll oder andere Protokolle, zum Beispiel JMS, zum Transport genutzt. SOAP kann zur Kommunikation als Nachrichtenformat genutzt werden. Das SOA-Dreieck-Modell kann jedoch durch die Technologien von Webservices gezeigt (siehe Abbildung 2.3) werden. Im Laufe der Zeit haben sich drei Technologien für Webservices etabliert. Dies sind dann SOAP, WSDL und UDDI.

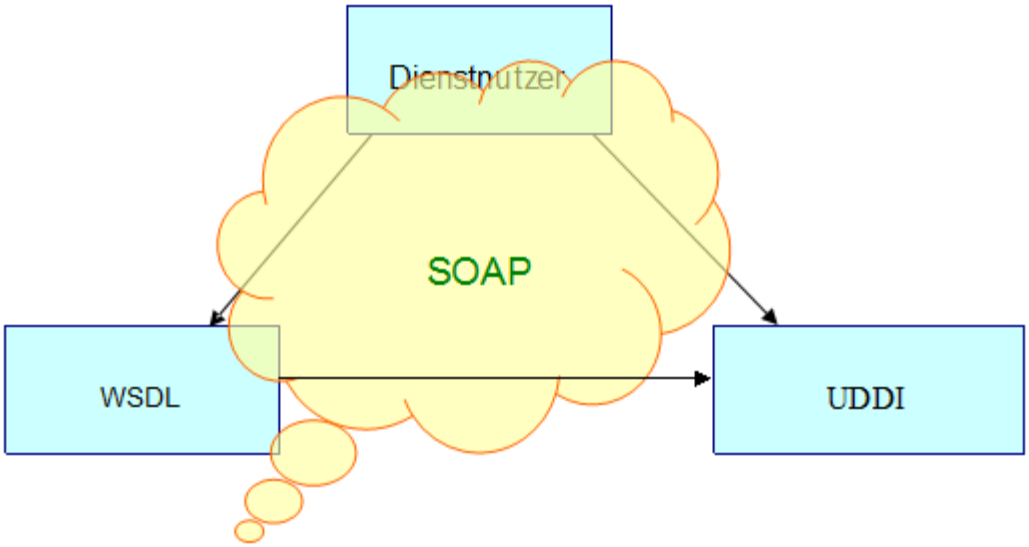

Abbildung 2.3 SOA-Dreieckmodell auf Webservices angepasst

2.2.1 SOAP

SOAP ist dafür entwickelt, die Kosten und Komplexität der Integration der Anwendungen, die auf verschiedenen Plattformen aufgebaut sind, zu reduzieren [WCL+05]. SOAP basiert auf XML und beschreibt die Art und Weise, wie die Informationen zwischen den Kommunikationspartnern innerhalb einer verteilten Umgebung ausgetauscht werden können. Bei der Übertragung von Daten werden die Parameter von Aufrufenden übergegeben, die in Form einer XML-Struktur an den verarbeitenden Kommunikationspartner weitergegeben werden [HW05]. Dieser Partner sendet dem Client auch die Ergebnisse der aufgerufenen Anwendung in Form einer XML-Struktur zurück.

Weil die XML-Repräsentation der Objekte und Strukturen die Daten sehr groß machen kann und deshalb von der Netzwerkbandbreite ziemlich stark abhängig ist, hat SOAP seine Nachteile im Anwendungsbereich der Kommunikation zwischen mobilen Endgeräten und einem Server, wo eine performante Übertragung besonders wichtig ist. Im Bereich der Maschinensteuerung ist SOAP auch nicht geeignet, weil die Daten dabei in Echtzeit übertragen werden müssen.

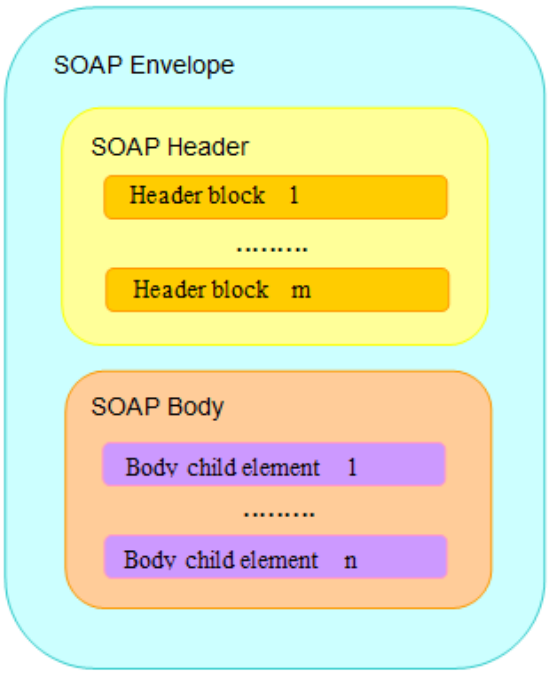

Abbildung 2.4 Die allgemeine Struktur einer SOAP-Nachricht

Eine SOAP-Nachricht wird von einem initialen Sender zu einem finalen Empfänger übermittelt. Aber es kann auch dazwischen andere Knoten geben, die als Mittler genannt werden. Der gesamte Weg von dieser Nachricht zwischen dem initialen Sender und dem finalen Empfänger wird Nachrichtenweg genannt. Die Abbildung 2.5 zeigt ein Beispeil eines Nachrichtenwegs.

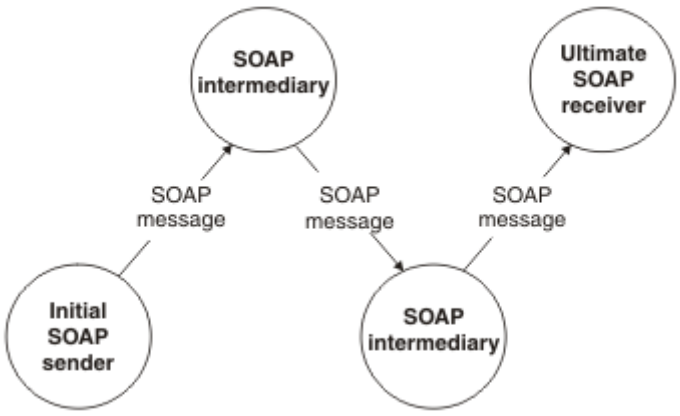

Abbildung 2.5 Der Nachrichtenweg von SOAP [IBM]

Eine SOAP-Nachricht besteht aus 3 Teilen:

Envelope

Der Envelope ist das Wurzelelement in einer SOAP-Nachricht. Der Envelope wirkt wie ein Umschlag für diese gesamte SOAP-Nachricht und dient als ein Container für die Anfrage und die Anwort innerhalb dieser SOAP-Nachricht. Der Envelope definiert den Rahmen, was in einer SOAP-Nachricht enthalten ist [Ulm].

Header

Der Header ist optional und kann aus vielen Headerblöcken bestehen. Die Headerblöcke enthalten Daten für die Bearbeitung auf SOAP Knoten. Header spezifizieren durch das role-Attribut, wer sie verarbeiten soll [Bur05]. Im Header können Meta-Informationen, beispielweise zum Routing, zur Verschlüsselung, zur Transaktionsidentifizierung oder zur Authentifizierung (Login, Passwort), untergebracht werden [Vik08] [WSOAP]. Wenn der Envelope ein Header enthält, muss der Header das erste Element in diesem Envelope sein [W3S].

Body

Der Body stellt die eigentliche Nachricht dar, die weitergeleitet werden soll. Ein Beispielcode zeigt den Body in einer SOAP-Nachricht. Im Code wird eine Suchanfrage an die Google-Suchmaschine geschickt. Die aufgerufene Methode im Code ist 'doSearchinGoogle'. Anschließend folgen Hinweise, auf welche Weise die Daten decodiert werden sollen. Dies wird im Code in Form eines XML-Schemas angegeben. Um diese Methode aufzurufen, braucht man zwar auch die einzelnen Parameter. Der 'key'-Tag ist dann dieser benötige Parameter, der den Suchstring darstellt. Hier wird 'aaaa' als der Suchstring gesendet.

```
<SOAP-ENV:Body>
   <m:doSearchinGoogle xmlns:m="urn:GoogleSearch"
       <SOAP-ENV:encodingStyle =
        http://schemas.xmlsoap.org/soap/encoding/>
       <key xsi:type = "xsd:string">aaaa</key>
    </m:doSearchinGoogle>
</SOAP-ENV:Body>
```

Listing 2.1 Der Body in einer SOAP-Nachricht

2.2.2 WSDL

WSDL ist die Abkürzung von ' Web Services Description Language' und auch XML-basiert. Während SOAP in erster Linie für Remote-Prozedur- oder -Funktionsaufrufe verwendet wird, ermöglicht WSDL die Beschreibung der Schnittstelle von Services [Man07].

WSDL wird innerhalb von Web Services im folgenden Bereich benutzt.

Beschreibung über einen Service für seinen Client - In diesem Fall wird der schon veröffentlichte Service für seinen Client beschrieben. Diese Beschreibung enthält die Angabe über die Nachrichten, Operationen, den Nachrichtenaustausch und auch die Lage des

Services. Zusätzlich enthält diese Beschreibung auch den Mechanismus, wie der Service benutzt werden muss. Das Hauptziel von WSDL in diesem Bereich ist es, dem Client von diesem Service zu ermöglichen, den Service zu benutzen [WCL+05].

WSDL 1.1

WSDL 1.1 ist die allgemeine Norm für die Beschreibung der Web Services. Es wird von vielen Anbietern im Bereich der Werkzeugentwicklung und Laufzeit unterstützt [WCL+05].

Das WSDL-Dokument kann in zwei Gruppen von Abschnitten unterteilt werden. Die erste Gruppe wird als abstrakter Teil bezeichnet, weil sie die abstrakten Definitionen enthält. Die andere Gruppe ist dann der konkrete Teil, der die konkreten Beschreibungen enthält. Der abstrakte Abschnitt definiert die eingehenden und ausgehenden Nachrichten der verschiedenen Operationen auf plattform- und sprachunabhängige Weise. Weil sie keine Elemente über die Spezifikationen von Maschinen oder Sprachen enthalten, können Dienste definiert werden, die mit verschiedenen Technologien implementiert werden können. Die spezifischen Funktionen wie Serialisierung der Nachrichten werden in der unteren Abschnittsgruppe (der konkrete Teil) mit den konkreten Beschreibungen angegeben [Tap04]. Der konkrete Abschnitt zeigt, wo und wie die Dienste adressiert werden können [Son08]. Zur syntaktischen Struktur von WSDL 1.1 siehe die Abbildung 2.6.

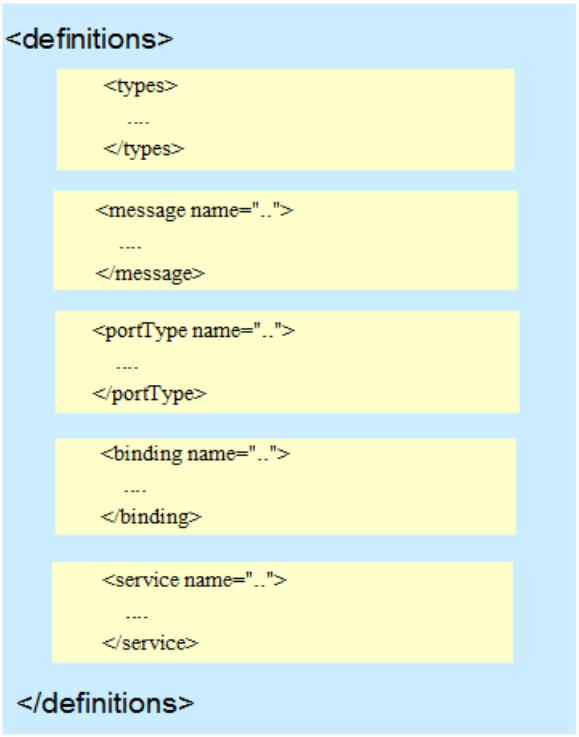

Abbildung 2.6 Die syntaktische Struktur von WSDL 1.1 [WCL+05]

Abstrakte Definitionen

- **Types**
  Types sind die maschinen- und sprachunabhängigen Typedefinitionen, die zur Beschreibungen der Nachrichten (Message) dienen [Tap04].
- **Message**
  Message repräsentiert eine abstrakte Definition über die Daten, die übertragen werden.
- **PortType**
  PortType ist eine Menge von abstrakten Operationen. Jede Operation bezieht sich auf eine eingehende und auch eine ausgehende Nachricht sowie auf Fehlernachrichten [Tap04]. Es gibt vier Verschieden Operationstypen, wie die Abbildung 2.7 zeigt.

Operation	Beschreibung
One-way	Service erhält eine Nachricht ohne Anwort
Request-response	Service bekommt eine Nachricht und produziert auch eine Antwortsnachricht
Solicit-response	Service sendet eine Nachricht und bekommt auch eine Antwort
Notification	Service sendet eine Nachricht und erhält nichts als Antwort

Tabelle 2.1 Die vier verschiedenen Operationen in WSDL1.1

Konkrete Beschreibungen

- **Binding**
  Binding verweist auf die einzelne Operation im PortType-Abschnitt. Binding spezifiziert ein konkretes Übertragungsprotokoll und auch das Format der Nachrichten, die von einem PortType definiert sind.
- **Port**
  Port spezifiziert den Ort, an dem ein Service erreichbar ist.
- **Service**
  Service wird benötigt, um eine Menge von zugehörigen Ports zu Aggregieren [Tap04].

2.3 BPEL

Dieser Abschnitt basiert im Wesentlichen auf der Spezifikation der BPEL-Version 2.0 [OASIS]. BPEL ist die Abkürzung für Business Process Execution Language. BPEL ist eine XML-basierte Sprache zur Beschreibung von Geschäftsprozessen, baut auf den Ideen der Workflow-Definitions-Sprachen WSFL und XLANG auf [Ste08] und wurde von OASIS als WS-BPEL 2.0, kurz BPEL, standardisiert.

2.3.1 Prozesse

Die grundlegenden Konzepte von WS-BPEL können in zwei Bereichen verwendet werden, die abstrakt und ausführbar sind [OASIS]. Abstrakte Prozesse bieten eine abstrakte Sicht auf die Geschäftsprozesse und ausführbare Prozesse dienen der konkreten Implementierung von solchen Geschäftsprozessen [Ste08].

Ein abstrakter Prozess könnte einige benötigte konkrete betriebliche Einzelheiten ausblenden [OASIS]. Abstrakte Prozesse verweisen auf das Geschäftsprotokoll oder Nachrichtenaustauschprotokoll und das Geschäft- und Nachrichtenaustauschprotokoll beschreibt nur die äußerlich sichtbaren Interaktionen zwischen den Geschäftspartnern. In abstrakten Prozessen gibt es keine Inhalte über die Geschäftslogik von den individuellen Geschäftspartnern. Ein Unternehmen kann seinen abstrakten Prozess vor einem existierenden Geschäftsprozess vorzeigen, um seinem Partner zu zeigen, wie mit ihm zu interagieren ist. Ein abstrakter Prozess kann als eine Projektion von einem ausführbaren Prozess betrachtet werden.

Im Vergleich zu dem abstrakten Prozess enthält der ausführbare Prozess die Geschäftslogik von Partnern. Im ausführbaren Prozess werden alle Einzelheiten beschrieben und definiert, um alle Instanzen auf einer Workflowengine ausführen zu können. Die Einzelheiten können die Typen von Variablen, die Attribute, die Ausdrücke und auch die Geschäftslogik sein. Weil die ausführbaren Prozesse oft die innerbetrieblichen Informationen enthalten, werden sie dann als Geheimnisse von Unternehmen betrachtet.

2.3.2 Variablen

Zur Zwischenspeicherung von Werten stehen typisierte Variablen zur Verfügung. Sie dienen sowohl als Eingabeparameter als auch Ausgabeparameter von Aktivitäten [Ste08]. Variablen können auch verwendet werden, um das Verhalten vom Prozess zu steuern. Variablen bieten die Möglichkeit, die Nachrichten beizubehalten, die die Zustände eines Geschäftsprozesses darstellen. Solche Nachrichten werden von Partnern empfangen oder zu Partnern gesendet. WS-BPEL benutzt drei unterschiedliche Typen von Variablen: WSDL Message Type, XML Schema Type und XML Schema Element [OASIS]. XPath ist die vorgegebene Sprache für die Verarbeitung und Abfrage von Variablen [WCL+05].

2.3.3 Aktivitäten

In WS-BPEL wird zwischen basischen und strukturierten Aktivitäten unterschieden. Die basischen Aktivitäten sind die grundlegenden Aktivitäten, welche die elementaren Schritte des Verhaltens von Prozessen beschreiben. Im Gegensatz zu den basischen Aktivitäten beinhalten strukturierte Aktivitäten auch die anderen Aktivitäten und ermöglichen deshalb die Komposition von komplexen Prozessen zugelassen werden [Sch07]. Die strukturierte

Aktivität definiert die Geschäftslogik für alle Aktivitäten, die sie beinhaltet [WCL+05]. Das Verhalten von Aktivitäten in einem Geschäftsprozess ist wie folgt beschrieben [LR00]:

- Eine Aktivität ruft einen anderen Geschäftsprozess auf, der vollkommen unabhängig von dem ursprünglichen Geschäftsprozess arbeitet.

- Eine Aktivität ruft einen anderen Geschäftsprozess auf und wartet, bis dieser neue Geschäftsprozess abgearbeitet ist.

- Eine Aktivität ruft einen anderen Geschäftsprozess auf, eine später ausgeführte Aktivität wartet, bis dieser neue Geschäftsprozess abgearbeitet ist.

<invoke>, <receive> und <reply> sind die basischen Aktivitäten, mittels derer die Kommunikation mit Geschäftspartnern ermöglicht wird. Die Nachrichten können durch <invoke> und <reply> gesendet werden und durch <receive> empfangen werden. <invoke> kann benutzt werden, um die Operationen der Webservices von Geschäftspartnern aufzurufen. Eine <receive>-Aktivität empfängt eine einzelne eingehende Nachricht. Wenn der Werte des Attributs "createInstance" der <receive>-Aktivität auf "yes" gesetzt ist, wird eine neue Prozessinstanz instantiiert und begonnen [Son08]. <reply> ermöglicht das gleichzeitige Senden einer Beantwortung an den Partner. <while>, <sequence> und <flow> sind die Aktivitäten, die die Geschäftsprozesse strukturieren können. <while> kann eine Schleife realisieren, um die enthaltenen Aktivitäten mehrmals ausführen zu können. Ein sequentieller Ablauf kann durch <sequence> definiert werden. Dabei können die enthaltenen Aktivitäten in einer vorgeschriebenen Reihenfolge nacheinander durchgeführt werden. <flow> definiert dagegen einen parallelen Ablauf, d.h. die enthaltenen Aktivitäten können demgegenüber parallel durchgeführt werden [Ste08]. In <flow> dienen die <links>-Konstrukte dazu, dass die Abhängigkeiten zwischen den einzelnen enthaltenen Aktivitäten bestimmt werden können. Die basischen und strukturierten Aktivitäten werden in der folgenden Tabelle angezeigt (siehe bitte die Abbildung 2.8).

Aktivitäten	Basische Aktivität	Strukturierte Aktivität	Beschreibung
<invoke>	√		Aufruf von Web Services
<receive> <reply>	√		Bereitstellung von Web Services
<assign>	√		Update von Variablen und Partnerlink
<throw>	√		Signalisieren von internen Fehlern
<wait>	√		Verzögerte Ausführung
<empty>	√		Nichts zu tun
<extensionActivity>	√		Anfügen der neuen Typen von Aktivitäten
<exit>	√		Sofortige Beendigung eines Prozess
<rethrow>	√		Propagierung von Fehlern
<sequence>		√	Sequenzielle Verarbeitung
<if>		√	Bedingte Ausführung
<while>		√	Wiederholte Ausführung
<repeatUntil>		√	Wiederholte Ausführung
<pick>		√	selektive Verarbeitung von Ereignissen
<flow>		√	Parallele Verarbeitung unter Berücksichtigung von Abhängigkeiten der Ausführung
<forEach>		√	Verarbeitung von mehrfachen Bereichen

Tabelle 2.2 Die basischen und strukturierten Aktivitäten in WS-BPEL

2.3.4 Korrelationsmengen (Correlation Sets)

Die Korrelationsmengen werden in <invoke>, <receive>, und <reply> verwendet [OASIS]. Die verschiedenen Instanzen eines BPEL-Prozesses können sich anhand von Korrelationsmengen unterscheiden lassen. Wenn zu einem Zeitpunkt mehrere Instanzen eines BPEL-Prozesses aktiv sind, werden die Daten von den Korrelationmengen benutzt, um die eingehenden Nachrichten an die richtige Instanz weiterzureichen.

2.3.5 Scopes und deren Handler

<scope> ist die einzige strukturierte Aktivität, die die Aktivitäten beinhaltet, die nicht nur innerhalb dieser <scope>-Aktivität sind, sonder auch mittels Handlern an dieser <scope>-Aktivität angebracht werden können [WCL+05]. Die Gültigkeitsbereiche können anhand von der <scope>-Aktivität festgelegt werden. In einer <scope>-Aktivität können für einen einzigen Gültigkeitsbereich eigene Variablen und eigene Handler definiert werden Die definierten Variablen und Handler sind dann nur in diesem Gültigkeitsbereich gültig. Der Prozess hat auch eigene <scope>-Aktivität als der globale Gültigkeitsbereich, die auch Prozessscope genannt werden kann. Die <scope>-Aktivität kann vier unterschiedliche

Handler haben. Diese sind dann <compensationHandler>, <faultHandlers>, <terminationHandler> und <eventHandlers>. Nachdem eine <scope>-Aktivität gestartet wird, übergibt sie die Kontrolle an alle enthaltenen Aktivitäten und macht auch die <eventHandlers>- und <faultHandlers>-Aktivitäten lauffähig. Nachdem diese <scope>-Aktivität beendet wurde, deaktiviert sie diese Handlers. Wenn diese <scope>-Aktivität erfolgreich beendet wurde, dann macht sie die <compensationHandler> lauffähig [WCL+05].

Jeder Scope inklusive dem Processscope kann eine Menge von <eventHandlers>-Aktivitäten haben, die gleichzeitig laufen können und aufgerufen werden können, sobald ein entsprechendes Event passiert [OASIS]. Ein <eventHandlers> dient dazu, auf eine bestimmte eingehende Nachricht zu reagieren. Wenn eine entsprechende Nachricht ankommt, kann eine Aktivität, die in diesem Event Handler festgelegt ist, gleich ausgeführt werden. Ähnlich kann ein <eventHandlers> auch die Aktivitäten definieren, die später zu einem bestimmten Zeitpunkt durchgeführt werden sollen.

Ein Fault Handler definiert die Aktivitäten, die für den Fall durchgeführt werden, dass ein entsprechender Fehler (Fault) innerhalb des Gültigkeitsbereiches vorkommt, aber dieser Fault noch nicht in einem enthaltenen Gültigkeitsbereich abgefangen und bearbeitet wurde [Ste08]. Wenn ein Fault nicht behandelt und bearbeitet werden kann, wird dieser Fault an den umschließenden Gültigkeitsbereich weitergeleitet werden [Ste08]. Der Processscope ist der weiteste Gültigkeitsbereich in einem Prozess. Wenn ein Fault im Processscope auch nicht abgefangen werden kann, dann wird der Prozess als fehlerhaft beendet. Die von einem Fault Handler spezifizierten Aktivitäten werden in <catch> oder <catchAll> definiert.

Die Aktivitäten, die für den entsprechenden Gültigkeitsbereich im Fall der Durchführung einer Kompensation ausgeführt werden sollen, können in einer <compensationHandler>-Aktivität definiert werden. Aber die Voraussetzung dafür, dass solche in <compensationHandler> enthaltenen Aktivitäten ausgeführt werden, ist, dass dieser Gültigkeitsbereich bereits beendet und verlassen wurde [Ste08]. Nicht alle Gültigkeitsbereiche dürfen Compensation Handler haben und dies ist auch der Fall bei Prozess bzw. Prozessscope. Ist in einem Scope kein Compensation Handler vorhanden, so muss bei der Ausführung eines BPEL-Prozesses standardmäßig ein Compensation Handler erstellt werden.

Termination Handler ist dann der letzte Handler-Type. Ein Termination Handler kann auch im Scope definiert werden. Anhand vom <terminationHandler> kann eine erzwungene Terminierung aller in dieser <scope>-Aktivität enthaltenen Kindaktivitäten durchgeführt werden. Ähnlich wie die <compenstionHandler>-Aktivität wird auch ein Standard-Termination-Handler definiert, wenn keiner vorhand ist. Es ist aber auch erlaubt, dass ein Prozess kein Termination Handler besitzt.

3. Apache ODE

Die Apache ODE (Orchestration Director Engine) wurden von der Apache Software Foundation entwickelt und gehört auch zu den Top-Level-Projekten [WODE]. Sie ist unter einer Open-Source-Lizenz in Version 2.0 verfügbar. Die aktuellste stabile Version der Apache ODE ist 1.3.4, die auch in dieser Arbeit erweitert werden soll.

3.1 Grundlagen der Apache ODE

Die Apache ODE kann die Geschäftsprozesse ausführen, die nach dem Standard von WS-BPEL geschreiben sind. Sie kommuniziert mit Web Services, sendet und empfängt die Nachrichten, bearbeitet die Daten und behandelt auftretenden Fehlern auf die Weise, wie es die BPEL-Prozesse vorher definiert hatten. Die Apache ODE kann nicht nur die Kurzzeit-Ausführung sondern auch die Langzeit-Ausführung unterstützen und kann damit alle in der Anwendung enthaltenen Services orchestrieren [ODE].

Die Apache ODE kann in folgenden drei verschiedenen Varianten verfügbar gemacht werden:

- WAR Deployment - hier wird die ODE als WAR gepackt und kann irgendwo entgepackt werden. In der Entwicklungsumgebung wird das WAR-File "ode-axis2-war-1.3.4 "im Ordner "axis-war\target" in den Orner "webapp" von Tomcat kopiert. Der Name des WAR-Files muss normalerweise in "ode" umbennant werden. Wenn Tomcat gestartet wird, läuft die Apache ODE auch. Wenn die Apache ODE erfolgreich deployed wurde, kann die Seite "http://localhost:8080/ode" richtig im Browser angezeigt werden.

- JBI Deployment - hier ist die Apache ODE als eine ZIP-File gepackt und kann in einem JBI Container deployed werden [Sch11].

- SMX4 OSGI Deployment

Die Eigenschaften der Apache ODE:

- ODE unterstützt den Standard von WS-BPEL 2.0 und auch den Standard von BPEL4WS 1.1.aber mit einigen Abweichungen.

- ODE unterstützt zwei verschiedene Kommunikationsebenen bezüglich Axis 2 und JBI.

- ODE unterstützt das Binding von HTTP und auch WSDL und erlaubt die Aufrufe der Webservices in REST.

- ODE macht es möglich, die Variable der Prozesse von außen zu einer Tabelle in einer Datenbank abzubilden.

- ODE bietet eine Managementschnittstelle für die Prozesse, Instanzen und auch Nachrichten an.

3.2 Abweichungen vom WS-BPEL 2.0 Standard

Der Standard WS-BPEL 2.0 kan durch die Apache ODE nur mit einigen Abweichungen unterstützt werden. Diese Abweichungen betreffen die Variablen und die folgenden Aktivitäten.

- **Variablen**

```
<variables>
    <variable name = "BPELVariableName" messageType = "QName" ?
              type = "QName"? element = "QName" ?> +
              from-spec?
    </variable>
</variables>
```

Listing 3.1 Variablen mit inlinern "from-spec" [OASIS]

Der inliner "from-spec", damit eine Variable eventuell initialisert werden kann, wird jetzt noch nicht von ODE unterstützt. Ein Patch in ODE-236 dafür ist momentan aber verfügbar.

- **<receive>**

In der <receive>-Aktivität kan die Syntax von <fromPart> leider noch nicht von ODE unterstützt werden. Dabei muss ein Attribut nämlich *variable* verwendet werden. Außerdem können nur die Variablen von Nachrichten im *variable* Attribute referenziert werden, obwohl die Variablen von Elementen nach der Spezifikation auch erlaubt sind. Mehrere Start-Aktivitäten werden auch nicht von der ODE unterstützt, wobei die Benutzung von *initiate = "join"* ausgeschlossen ist. Die ODE kann die in der Spezifikation geschriebenen Anordungsrichtlinien nicht gewährleisten, wobei die ODE noch toleranter als die Spezifikation ist. Listing 5.2 zeigt einen BPEL-Code, der in BPEL nicht erlaubt ist, aber in der ODE zulässig ist.

```
<flow>
    <receive ... createInstance="yes" />
    <assign ... />
</flow>
```

Listing 3.2 Ein von ODE erlaubtes und von BPEL verbotenes Beispiel [ODE1]

Außerdem kann die ODE leider zwischen *conflictingRequest* und *conflictingReceive* nicht unterscheiden. Bei Beiden handelt es sich um die Fehler in Bezug auf mehrere unerledigte Anfragen auf ein einziges *partner-link/operation/messageExchange* Tupel. Das bedeutet, dass

conflictingReceive im Fall von *conflictingRequest* immer geworfen wird.

In der ODE wird das *validate*-Attribut einer <receive>-Aktivität ignoriert, weil die ODE die Validation von Variablen nicht unterstützt.

- **<reply>**

Die Anpassungen von der <reply>-Aktivität in ODE spiegeln auch einige von der <receive>-Aktivität wider. Die Syntax von <toPart> kann nicht in ODE unterstützt werden und das Attibut *variable* muss in den Variablen, die auf Nachrichten basieren, referenziert werden.

- **<invoke>**

Wie <receive> und <reply>-Aktivitäten kann die Synax von <toPart> und <fromPart> nicht unterstützt werden. Ähnlich muss in den Attributen *inputVariable* und *outputVariable* auf die Nachrichten-basierten Variablen referenziert werden. Das *validate*-Attribut hier wird auch in der ODE ignoriert.

- **<assign>**

Das *validate* Attribut von einer <assign>-Aktivität, das in der BPEL-Spezifikation definiert ist, um die Variablen zu validieren, wird nicht in der ODE unterstützt. Deshalb kann der Fehler *invalidVariables* niemals durch eine <assign>-Aktivität geworfen werden. Um die benutzten Sprachen innerhalb einer Anweisung festzustellen, wird das *expressionLanguage* statt des *queryLanguage*-Attributs in der ODE verwendet [Sch11].

- **<pick>**

Die <pick>-Aktivität hat die gleichen Einschränkungen in der ODE wie die oben beschriebene <receive>-Akvtivität.

- **<compensate>**

Die <compensate>-Aktivität entspricht der Spezifikation nicht. Die Aktivität hat die gleiche Auswirkung und die gleiche Syntax wie die <compensateScope>-Aktivität in der ODE.

- **<validate>**

Diese Aktivität kann noch nicht von der ODE implementiert werden. Die Prozesse, die <*validate*> enthalten, können zu Compilerfehler führen.

3.3 Die Architektur der Apache ODE

In diesem Abschnitt werden nur die grundlegenden Informationen über die Architektur von der Apache ODE vorgestellt. Die Hauptziele bei der Entwicklung der Apache ODE waren, eine vertrauenswürdige, stabile Workflow-Engine zu entwickeln, die aus vielen Komponenten zusammengebaut wird. Die Engine muss dabei die Fähigkeit haben, die langlebigen Prozesse von BPEL durchführen zu können [Sch11]. Um die Apache ODE näher zu betrachten, wird in diesem Abschnitt die Architektur von der ODE vorgestellt. Dazu steht die folgende Abbildung 3.1 zur Verfügung.

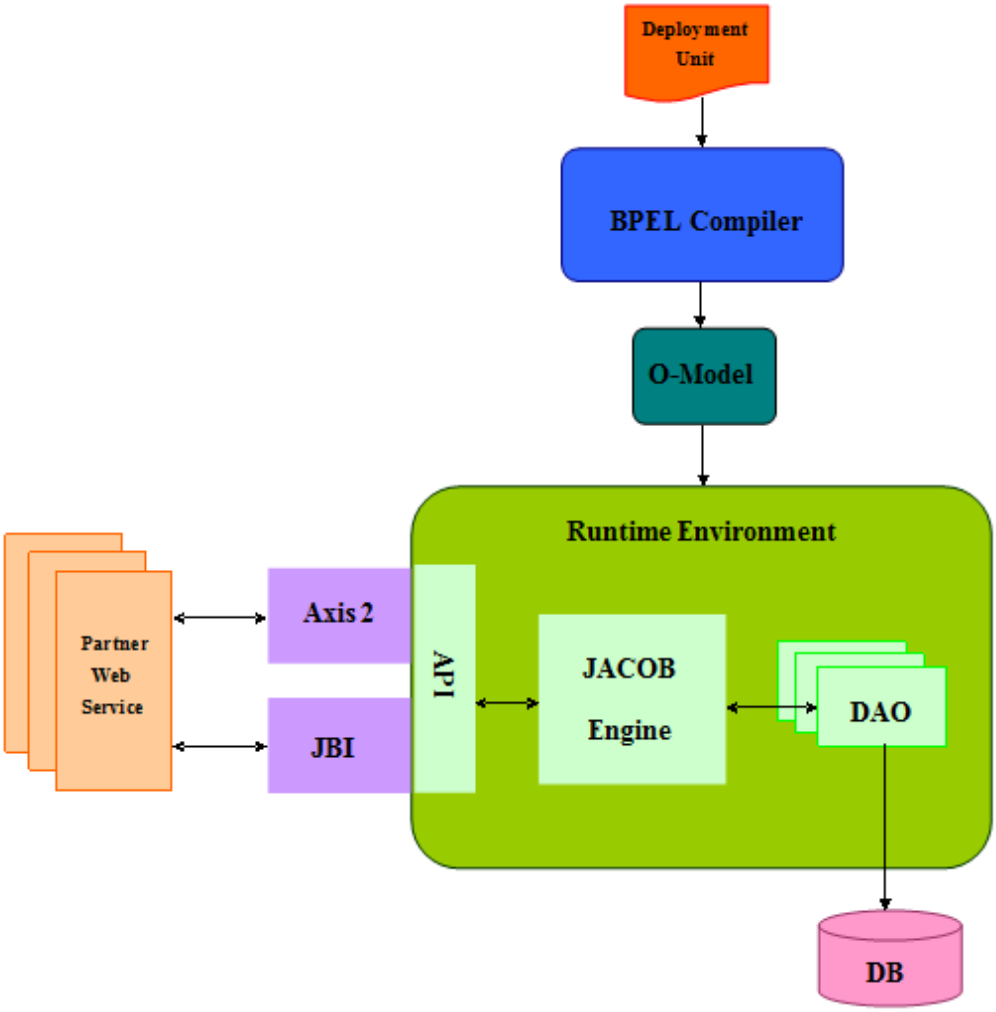

Abbildung 3.1 ODE-Architektur [Son08]

Die Hauptkomponenten der Apache ODE-Architektur sind ein ODE-BPEL-Compiler, eine ODE-BPEL-Engine-Runtime, ODE Data Access Objects (DAOs) und auch der ODE Integration Layer (IL). In dieser Abbildung wird der ODE Integration Layer in zwei Komponenten beziehungsweise als Axis 2 und JBI dargestellt. Die vier Hauptkomponenten sind auch in vier verschiedenen Farben dargestellt.

Die ODE-BPEL-Compiler (blau) ist verantwortlich dafür, dass die einzelnen BPEL-Artefakte, die Dokumente des BPEL-Prozesses, WSDL- und auch XML-Schemata in einen durchführbaren Prozess konvertiert werden können. Diese Dokumente können durch den Compiler in eine für die ODE-BPEL-Runtime-Engine gut verständliche und lesbare Form umgewandelt werden [Ste08]. Diese vom ODE-BPEL-Compiler kompilierten Ergebnisse werden dann für die ODE-BPEL-Engine Runtime (grün) bereitgestellt. Die-Engine-Runtime ist der Kern und die Hauptkomponente in der ODE und enthält die Data Access Objects (DAO) als Schnittstelle für die Persistenzebene(hellgrün), eine API(hellgrün) für Axis 2 und JBI und auch eine Engine (hellgrün), die das Prozessmodel nach seiner Definition ausführt. Die Persistenzebene ermöglicht es, die Informationen über die Prozesse, die Instanzen und auch Deployment Units (DUs) in der Datenbank (rosa) abzuspeichern. Der Geschäftspartner

kann mittels des Integration Layers (lila) von der Engine aufgerufen werden.

3.4 ODE-BPEL-Compiler und ODE-Objekt-Modell

Wie in Abschnitt 3.3 schon vorgestellt wurde, wandelt der Compiler BPEL-Prozesse in eine für die Engine gut verständliche Form um. Dabei werden die BPEL-Prozesse von dem Compiler kontrolliert und auf die Kompatibilität zu dem ODE-Objekt-Modell untersucht [Ste08]. Wenn der BPEL-Prozess erfolgreich kompiliert wurde, ist das Ergebnis nach der Kompilierung ein ausführbarer Prozess. Wenn Fehler bei der Kompilierung auftreten, wird eine Liste von Fehlernachrichten erstellt, die auf die Probleme in den Quelldateien hinweist. Die übergegebenen BPEL-Prozesse werden zuerst vom Compiler überprüft und zum ODE-Objekt-Modell (O-Model) kompiliert. Das erfolgreich kompilierte ODE-Objekt-Modell wird dann in einer *.cbp* Datei abgespeichert [Ste08] (siehe die Abbildung 3.2). Nachdem die BPEL-Prozesse in eine Form von ODE-Objekt-Modell umgewandelt wurden, werden sie dann ausgeführt, dabei helfen aber auch die DAOs. DAOs ermöglichen die Daten auf eine sichere Weise zu speichern und können zur Persistenz der Daten beitragen.

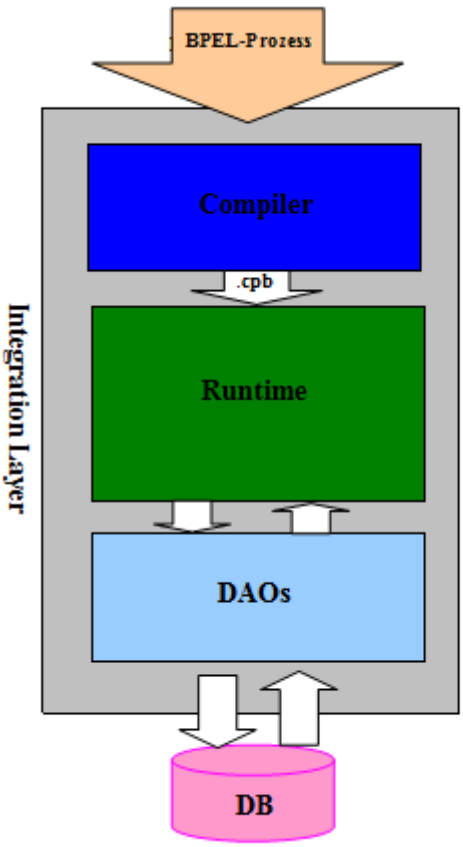

Abbildung 3.2 Verwaltung eines BPEL-Prozesses in der Apache ODE [Ste08]

Mindestens ein passendes Objekt im ODE-Objekt-Modell wird für jede Aktivität, für jeden Prozess und für jeden Handler beziehungsweise Event Handler, Termination Handler, Compensation Handler und auch Fault Handler vom ODE-Compiler erzeugt. Die folgende Abbildung 3.3 zeigt einen Teil aus dem ODE-Objekt-Modell.

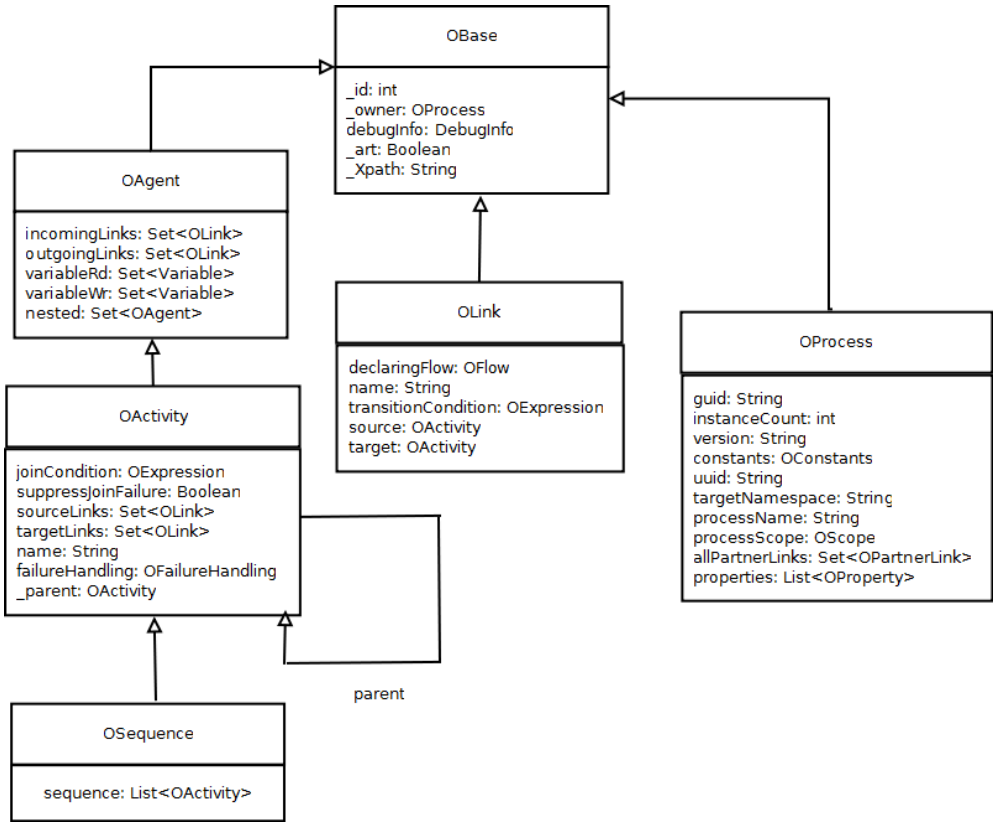

Abbildung 3.3 Teil aus dem Klassendiagramm der Objekte im ODE-Objekt-Modell

Die Klasse *OBase* ist die oberste Klasse im ODE-Objekt-Modell und alle anderen Klassen erben von dieser Klasse. Die Klasse *OBase* hat zwei wichtige private Attribute "_id" und "_owner". Durch die zwei Attribute kann ein Objekt präzise identifiziert werden. Das Attribut "_owner" deutet den Prozess an, zu dem das Objekt gehört. Die Klasse *OProcess* stellt einen BPEL-Prozess dar und erbt ebenfalls von der Klasse *OBase*. Die Klasse *OAgent* erbt auch von der Klasse *OBase*, die eingehende und ausgehende Links und lesbare und schreibbare Variablen besitzt. Die Klasse *OLink* erbt von der Klasse *OBase* und hat zwei Attribute "source" und "target", die auf die Quellenaktivität und die Zielaktivität verweisen. In dieser Arbeit ist es sehr sinnvoll, die Klasse *OActivity* zu verstehen, weil diese Klasse die grundlegende Klasse für alle Aktivitäten darstellt. Wie in der Abbildung gezeigt hat, erbt die Klasse *OActivity* von der Klasse *OAgent*. Alle Aktivitätentypen in BPEL, die von Apache ODE unterstützt werden, erben von der Klasse *OActivity* . Das Attribut "parent" verweist auf die Vateraktivität dieser betroffenen Aktivität. Die <sequence>-Aktivität in BPEL wird in ODE als das Objekt *OSequence* dargestellt und hat eine Attribute "sequence", das auf die Kinderaktivitäten von dieser <sequence>-Aktivität verweist.

3.5 ODE-BPEL-Engine-Runtime

Die ODE-BPE- Engine-Runtime existiert innerhalb des BPEL-Runtime Moduls und bietet die Ausführung von den kompilierten BPEL-Prozesse an. Um die BPEL-Prozesse auszuführen, müssen unterschiedliche BPEL-Konstrukte von der Runtime implementiert werden. Die

Logik, wann eine neue Instanz erzeugt werden soll, muss auch von der Runtime implementiert werden. Zu welcher Instanz eine eingehende Nachricht gehört, muss auch von der Runtime selbst entschieden werden. Die Process Management API, die eine Schnittstelle für den Benutzer mit der ODE anbiete, wird auch von der Runtime realisiert. Um eine zuverlässige Ausführung von Prozessen in einer unzuverlässigen Umgebung zu garantieren, basiert die Runtime auf DAOs, die Persistenz anbieten können. Die DAOs werden im nächstem Abschnitt beschrieben.

JACOB

Mittels des ODE *Java Concurrent Objects (Jacob) Framework* können die BPEL-Sprachkonstrukte auf der Ebene von Workflow-Instanzen implementiert werden. *Jacob* hat einen Mechanismus zur Verfügung gestellt, der die folgenden zwei wichtigen Aufgaben gewährleisten kann:

1. Die Persistenz des Zustandes während der Ausführung.

2. Die Nebenläufigkeit.

Mit Hilfe von *Jacob* kann die Implementierung der BPEL-Konstrukte vereinfacht werden, weil nur die BPEL-Logik nicht aber Engine-spezifische Logik implementiert werden müssen. Die folgende Abbildung stellt beispielweise einen Teil aus dem Klassendiagramm für die Objekte auf der Instanzebene dar.

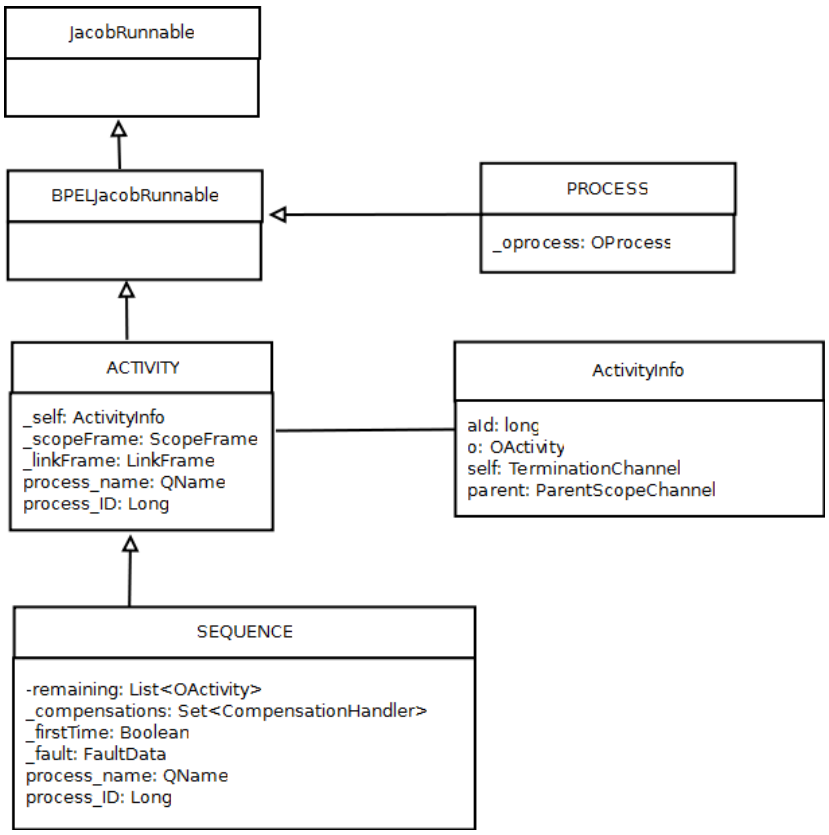

Abbildung 3.4 Teil aus dem Klassendiagramm für die Objekte auf der Instanzebene

Die Klasse *JacobObject* ist die oberste Klasse für die Objekte auf der Instanzebene. Die Klasse *JacobRunnable* erbt von der Klasse *JacobObject. JacobObject* wird als ein einfaches *Closure* beschrieben [Yu10]. Ein Closure kann auch als eine Programmfunktion bezeichnet werden, die beim Aufruf einen Teil von ihrem Erstellungskontext reproduziert, selbst wenn dieser Kontext außerhalb dieser Programmfunktion nicht mehr existiert [WCL].

Alle auf der Instanzebene existierenden Aktivitäten, Handler und Prozesse erben von der Klasse *JacobRunnable,* welche eine run-Methode vordefiniert hat. Alle Akvititäten, die von *JacobRunnable* erben, müssen ebenfalls auch die run-Methode besitzen, die die Hauptfunktionalität der jeweiligen Aktivität implementiert. Mittels des Attributes "_self" von der Klasse *ACTIVITY* kann das Objekt auf ein Objekt des ODE-Objekte-Modells verweisen, weil die Klasse *ActivityInfo* durch das Attribut "o" auf die Klasse *OActivity* verweisen kann. Aktivitäten, die zu verschiedenen Prozessinstanzen gehören, können beispielsweise auf dasselbe Objekt des ODE-Objekte-Modells zugreifen.

Wenn eine neue Prozessinstanz erzeugt wird, wird eine neue Instanz von der Klasse *BPELRuntimeContextImpl* erzeugt, die das Interface *BPELRuntimeContext* implementiert. Die Klasse implementiert die BPEL Prozessinstanzen. In der Klasse *BPELRuntimeContextImpl* stehen viele Funktionen zur Verfügung. Beispielsweise kann in dieser Klasse die Methode *terminate()* aufgerufen werden, um den Prozess vorzeitig zu beenden. Die Funktionalität, die in der Klasse *BPELRuntimeContextImpl* enthalten ist, wird nicht mehr direkt im *Jacob Framework* implementiert [Ste08].

Virtual Processing Unit(VPU) und ExecutionQueue

Gleich wenn die neue Instanz der Klasse *BPELRuntimeContextImpl* erzeugt ist, wird eine neue Instanz der Klasse *JacobVPU* auch erzeugt. Die *JacobVPU* ist der eigentliche Ort, wo die wirkliche Ausführung stattfindet. Wenn ein *JacobObject* aufgerufen wird, wird das *JacobObject* in der VPU als Continuation aufgelistet. Um ein *JacobObject* auszuführen, kann das *JacobObject* mittels *Continuation* mit der run-Methode dieses *JacobObjects* verbunden werden.

Um alle Artefakte in der Warteschlange zu verwalten, dient eine *ExecutionQueue* als Container für die *VPU,* weil alle von der *VPU* verwalteten Teile in der *ExecutionQueue* abgelegt werden. In der *ExecutionQueue* ist es möglich, die Artefakte aus der Warteschlange zu holen oder sie wieder in die Warteschlange zu legen. *ExecutionQueue* zeichnet auch gleichzeitig die Statistik der Ausführungen auf. Wenn eine Execution gestoppt wird, kann der Status von der *VPU* serialisiert und abgespeichert werden, wenn der Status später noch einmal gebraucht wird.

3.6 ODE Data Access Objects

Die ODE Data Access Objects vermitteln die Interaktionen zwischen der *ODE-BPEL-Engine-Runtime* und der untergeordneten Datenbank. Normalerweise ist die Datenbank eine relationale JDBC-Datenbank und in diesem Fall können die DAOs mittels OpenJPA implementiert werden. Es ist auch möglich, eigene DAOs zu implementieren, wobei ein Mechanismus zur Gewährleisung der Persistenz nicht mittels JDBC entwickelt werden kann. Zur Zeit können die folgenden von der ODE-BPEL-Engine geforderten Aufgaben durch die DAOs erledigt werden:

- Aktive Instanzen - Die Instanzen, die schon erzeugt sind, können durch DAOs abgefragt werden.

- Das Routing der Nachrichten - Welche Nachricht wird von welcher Instanz gewartet?

- Variablen - Die Werte der BPEL Variablen für jede Instanz sind mittels DAOs abgespeichert.

- Partner Links - Die neuesten Werte der Partner Links in BPEL für jede Instanz sind auch mit Hilfe von DAOs abgepeichert.

- Der Status von der Prozessausführung - *Jacob "persistent virtual machine"* serialisiert den Status von der Prozessausführung [ODE2].

3.7 Die BPEL Management API

Die BPEL Management API besteht aus zwei Teilen, nämlich dem *Process Definition Management Interface* und dem *Process Instance Management Interface,* und stellt die Funktionalität zur Verfügung, um die BPEL Prozesse und auch die zu den Prozessen gehörenden Instanzen zu verwalten. Mit Hilfe von der BPEL Management API kann der Benutzer beispeilerweise wissen, welche Prozesse schon durch die Engine deployed wurden, was die InstanzId für diese Prozesse ist, und die Zustände von den Aktivitäten.

Process Definition Management Interface

Diese API hat untenstehende Operationen definiert:

- *list* - listet die Informationen für alle, oder einige Definitionen der Prozesse auf. Die Operation *listProcesses()* ist zuständig dafür, dass ein bestimmter Prozess aufgelistet werden kann. Die Operation *listAllProcesses()* ermöglicht, dass die Informationen von allen in der Engine verfügbaren Prozessen, wie die ID, die Zustände, die Version usw. gezeigt werden.

- *details* - listet die ausführlichen Informationen über die Definition von spezifizierten Prozessen auf. Die Informationen für einzelnene Prozesse inklusive einer Übersicht dieser Instanz kann durch die Operation getProcessInfo() aufgelistet werden.

- *set-properties* - verändert die Eigenschaften. Die Operation *setProcessProperty()* kann eine Eigenschaft als einen einfachen Typen für den Prozess setzen.

- *activate* - aktiviert einen Prozess. Die Operation *activate()* kann einen Prozess aktivieren.

- *retire* - zieht einen Prozess zurück. Der Prozess kann durch die Operation *setRetired()*zurückgezogen werden und kann nicht mehr gestartet werden.

Process Instance Management Interface

Diese API definiert folgende Operationen:

- *list* - listet die Informationen über alle oder einige Instanzen auf. Die Operation listAllInstances() kann alle vorkommenden Instanzen auflisten.
- *detail* - listet die ausführlichen Informationen über die vorgegebene Instanz auf. Die Operation *getInstanceInfo()* ist eine Operation dafür.

- *suspend* - unterbricht eine Prozessintanz vorübergehend. Dazu steht die Operation *suspend()* zur Verfügung und diese Operation kann den Status dieser Instanz von *active* auf *suspended* verändern. Dabei muss die Instanzid eingegeben werden.

- *resume* - setzt die Prozessinstanz fort. Die Operation *resume()* kann eine pausierte Instanz fortführen. Im Gegensatz zu *suspend* kann *resume()* den Status dieser Instanz von *suspended* auf *active* zurücksetzten.

- *terminate* - terminiert die Prozessinstanz. Die Operation *terminate()* kann eine Instanz sofort beenden. Aber es ist unmöglich, irgendeinen *fault* oder *compensation handler* auszuführen.

- *fault* - kann eine Prozessinstanz abbrechen. Die Operation fault() ermöglicht, dass eine Instanz erfolglos beendet wird und dabei ein bestimmter Fehler ausgeworfen wird.

- *delete* - löscht alle oder einige vollendete Instanzen. Die Operation *delete()* benötigt einen Parameter, nämlich *filter* , was ein vorgegebener Name, ein bestimmter Status usw. sein könnte.

4. Verwandte Arbeiten

In diesem Kapitel werden drei relevante Arbeiten vorgestellt. Es existieren bereits wissenschaftliche Workflowsysteme und die Konzepte, die Wiederholung von Aktivitäten ermöglichen können.

4.1 E-BioFlow

Workflowsysteme sind schon erfolgreich für die Modellierung der Geschäftsprozesse und der Biowissenschaft im Einsatz. Das Modell der Workflows wird meistens in den folgenden zwei Bereichen verwendet. Bei Geschäftsprozessen ist das Workflowmodell am den Kontrollfluss orientiert, der die Reihenfolge der Aufgaben definiert. Aber bei Biowissenschaft ist das Workflowmodell am Datenfluss orientiert, der den Fluss der Informationen beschreibt. Die Workflowsysteme für die Biowissenschaften sollen noch weiter entwickelt werden, damit die Kontrollstrukturen besser modelliert werden können [WRV+08].

E-BioFlow ist dann ein Werkzeug des Workflows und wird im Bereich der Biowissenschaft eingesetzt. E-BioFlow wird von BioRange an der Universität Twente in den Niederlanden entwickelt. Es hilft den Biowissenschaftlern, die Experimente der Biowissenschaft durch die Verbindung der Webservices zu entwerfen. E-BioFlow ist ein Workflowsystem, das auf dem Kontrollfluss basiert. Die anderen Workflowsysteme im Bereich der Biowissenschaft sind dagegen am den Datenfluss orientiert [BioF]. Mehrere erweiterte Kontrollstrukturen, zum Beispiel die parallele Ausführung der Aufgaben, Iteration (loops) und auch die konditionale Verzweigung (if-then-else) können durch das Paradigma des Kontrollflusses modelliert werden [BioF].

Die von E-BioFlow erzeugten Daten sind deutlich im Workflowmodell dargestellt und können als die Datenquelle für die nächsten Schritte in den Experimenten benutzt werden. Solche Daten sind verfügbar als Eingabe für die neuen Aufgaben oder für die schon im Workflow existerenden Aufgaben [WOV09].

E-BioFlow wurde schon im Jahr 2009 zu einem Workflowsystem mit dem ad-hoc Editor weiterentwickelt. Es ermöglicht dem Entwerfer, den partiellen Workflow durchzuführen und den partiellen Workflow mit den Daten, die von dem schon ausgeführten Workflow erzeugt wurden, zu erweiteren [WOV09]. Die Vorteile vom ad-hoc-Editor im Vergleich zu den traditionellen Workflowsystemen werden in den folgenden Abschnitten in Anlehnung an [WOV09] vorgestellt.

4.1.1 Die Vorteile von E-BioFlow mit dem ad-hoc-Editor

- Die zu benutzenden Aufgaben sind oft unbekannt in der Designphase. Der ad-hoc Editor ermöglicht es dem Designer, die Aufgaben auszuprobieren.

- Der Designer des Workflows braucht nicht den kompletten Workflow im Voraus zu kennen, aber er kann den partiellen Workflow erweitern und ausführen.

- Im Fall einer kleinen Änderung im Workflow kann der ad-hoc Editor es ermöglichen, die entsprechenden Tasks zu wiederholen.

- Dazwischenliegende Ereigenisse können als die Informationsquellen benutzt werden, um die nächsten Schritte vom Workflow zu entscheiden.

- Der Designer muss keine Vermutungen zu den Daten anstellen, die von den Aufgaben produziert und gebraucht werden.

- Der Designer kann die Parameter einfach feinabstimmen und kann auch die Workflows durch die getrennte Ausführung der Aufgaben austesten.

4.1.2 Die sechs Perspektiven von E-BioFlow

Die Schnittstelle von E-BioFlow bietet sechs verschiedene Perspektiven: die Perspektive des Kontrolflusses, des Datenflusses, und die von Ressourcen, Engine von Workflows, Provenienzsysteme und der ad-hoc-Editor [WOV09].

Die Perspektive des Kontrollflusses (Control flow) fokussiert die Reihenfolge von Aufgaben. Sie bietet dem Designer des Workflows die Möglichkeit, die Reihenfolge der Ausführung der Aufgaben zu modellieren. Dabei können die Aufgaben sequenziell, parallel, iterativ und auch konditional durchgeführt werden.

Die Perspektive des Datenflusses (Data flow) wird benötigt, um die Überführung der Daten zwischen Aufgaben zu modellieren.

Die Perspektive der Ressourcen (Resource perspective) wird benutzt, um die Typen der Ressourcen zu definieren. Solche Ressourcen sind notwendig, um die Aufgaben auszuführen. Die tatsächlichen Ressourcen werden zur Ausführungszeit des Workflows bestimmt.

Die Engine des Workflows (Workflow engine) kann die Workflows ausführen. Sie ist verantwortlich für die Einplanung der Aufgaben, die Auffführung der Verbindungen und die Übergabe der Daten zwischen Aufgaben. Sie ist aufgebaut auf einer YAWL-Engine [VAD+04].

Das Provenienzsystem (Provenance system) erfasst alle Prozesse- und Daten-relevanten Informationen der Ausführung des Workflows.

Ad-hoc-Editor ist fähig, das ad-hoc-Design durchzuführen.

Alle Perspektiven werden in der Abbildung 4.1 gezeigt. Alle Perspektiven außer der Perspektive der Provenienz kommunizieren direkt mit dem Spezifikationscontroller (Specification Controller). Der Spezifikationscontroller verwaltet die noch offenen Workflows [WOV09]. Die Perspektiven senden die Anfragen an den Spezifikationscontroller im Fall von Änderungen im Workflowmodell, während der Benutzer das Diagramm des Workflows bearbeitet. Der Spezifikationskontroller setzt die Änderungen um und kündigt allen Perspektiven die Änderungen an. Die Änderungen senden und empfangen all diejenigen Perspektiven, die sich bei dem Spezifikationscontroller angemeldet haben. Der ad-hoc-Editor ist die Perspektive, die mit der Engine interagiert.

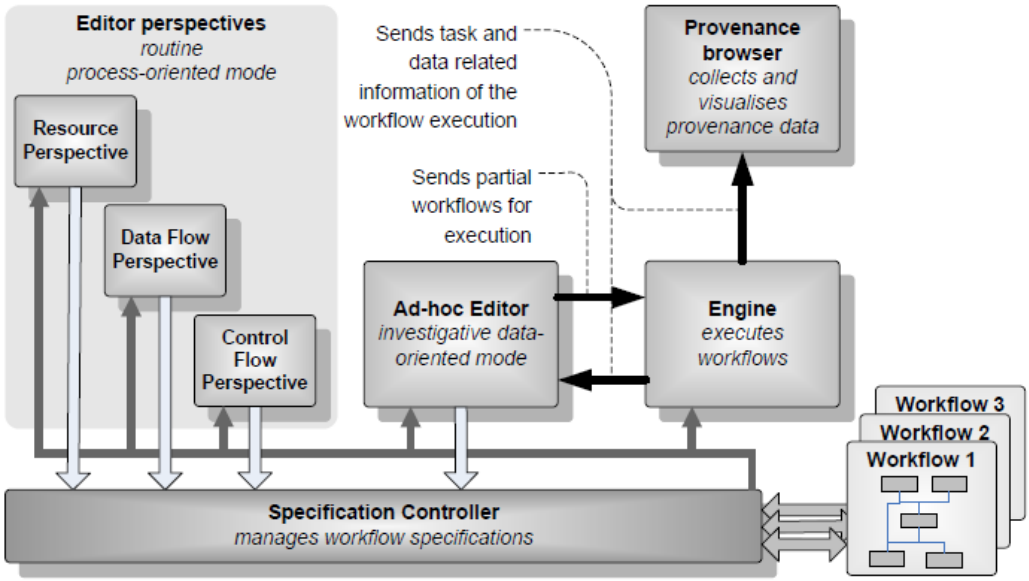

Abbildung 4.1 Alle Perspektiven von E-BioFlow mit ad-hoc-Editor [WOV09]

4.1.3 Ad-hoc Workflowdesign in E-BioFlow

Der Ad-hoc-Editor kombiniert die Eigenschaften von einem Editor, einer Engine und auch einem Provenanzbrowser [WOV09]. Die Modelle der Workflows können vom Ad-hoc-Editor und auch allen anderen Perspektiven gemeinsam benutzt werden. Wie in der Abbildung 4.1 gezeigt, kann der Ad-hoc-Editor durch den Controller der Spezifikation (Specification Controller) von jeder Änderung wissen und ebenfalls die Änderungen in dem Workflowmodell einsetzen, wenn ein Workflow im Editor modelliert wird. Der Ad-hoc-Editor wird in der folgenden Abbildung 4.2 gezeigt.

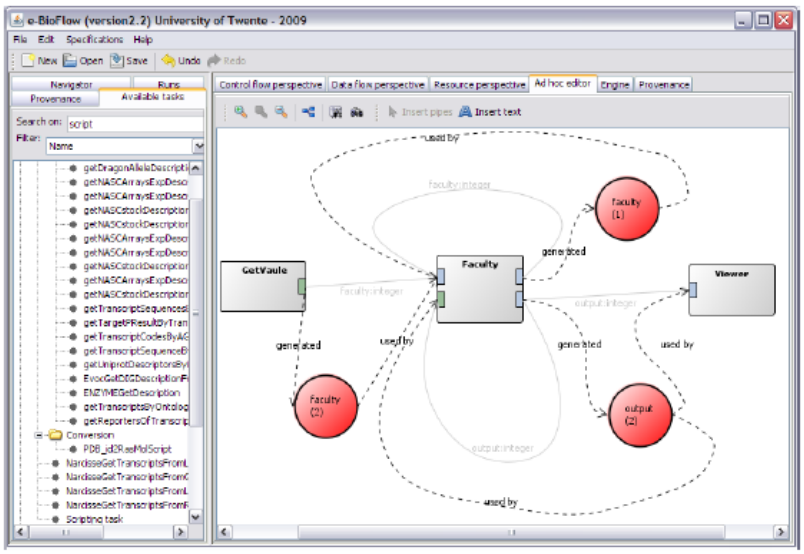

Abbildung 4.2 Der Screenshot des ad-hoc-Editors im E-BioFlow [WOV09]

Wie in der Abbildung 4.2 gezeigt, werden der Task als Rechteck und die Daten als Kreis im

Editor dargestellt. Die Daten haben zwei Ports, nämlich den Inputport und den Outputport. Die Daten werden vom Inputport benutzt (used by) und vom Outputport hergestellt (generated).

4.1.4 Ein im Ad-hoc-Editor entworfener Anwendungsfall

Im folgenden Abschnitt wird ein Anwendungsfall vorgestellt, der im Ad-hoc-Editor entwickelt wird. Dabei gibt es eine Operation für meine Arbeit relevant, die ähnlich wie die Iterate funktioniert.
Siehe bitte die folgende Abbildung 4.3.

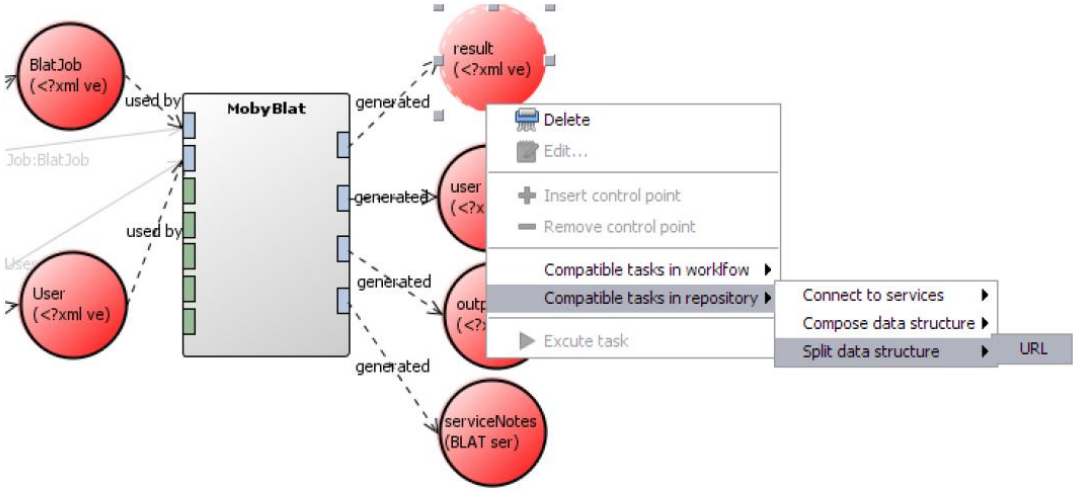

Abbildung 4.3 Der erste Task MobyBlat [WOV09]

Um eine biowissenschaftliche Frage zu analysieren, ist ein Webservice nämlich der Blat-Service, zu orchestrieren. Der Blat-Service kann in das Panel des Workflows gezogen werden. Der Webservice benötigt zwei Inputdaten. Der erste Input ist als User benannt, der für die Informationen über die Session verantwortlich ist. Der zweite Input ist als BlatJob benannt, der die Namen der Datenbank und der Sequenz anbietet. Die beiden Inputs sind XML-strukturiert. Aber der Editor kann dabei helfen, die komplizierten Datenstrukturen aufzubauen. Ein Composertask kann vom Benutzer im Editor hinzugefügt werden und der User kann als Inputdaten zu diesem Task angelegt werden bei rechtem Klick auf den Port. Der Benutzer hat den Composertask ausgewählt und auch dem Editor den Hinweis gegeben, den Composertask auszuführen. Der Editor lässt den Benutzer die Email-Adresse und das Passwort eingeben, um die komplexe Datenstruktur aufzubauen. Dann wird das Ergebnis des Composertasks im Editor gezeigt. Zwei Pfeile sind zu dem Workflowmodell neu hinzugefügt worden. Einer davon legt den Outputport des Composertasks im Datenitem (User) an und der andere legt das Datenitem (User) im Inputport des Tasks (MobyBlat) an. Der BlatJob Input kann auf die gleiche Weise wie der User-Input für den Task (MobyBlat) im Editor erstellt werden.

Jetzt sind alle notwendigen Inputdaten für unseren Task (MobyBlat) verfügbar und der User kann den Task gleich im Editor ausführen. Der Blat Service ergibt vier Ausgaben (siehe in Abbildung 4.3) und dies sind "result", "user", "output" und "serviceNotes". "result" ist die URL zum Blat-Bericht, "user" und "output" sind die Kopien von zwei Eingaben und "serviceNotes" bezeichnet ein Objekt des MOBY-S. MOBY-S sind die Webservices, die für

die Interoperabilität zwischen dem biowissenschaftlichen Host der Daten und den analytischen Services dient [MOB]. Der Benutzer muss das Ergebnis anhand der URL vom Blat herunterladen. Weil die URL ein Text im MOBY-S-XML-Format ist, muss der Benutzer durch einen Rechtsklick auf die Daten der URL einen Decomposertask für das MOBY-S Objekt anfordern (siehe die Abbildung 4.3). Nach der Ausführung des Decomposertasks ist die URL jetzt im Klartext.

Nach weiteren Designschritten gibt es im Editor jetzt die neuen Tasks. Im Abbildung 4.4 werden alle neuen Tasks gezeigt.

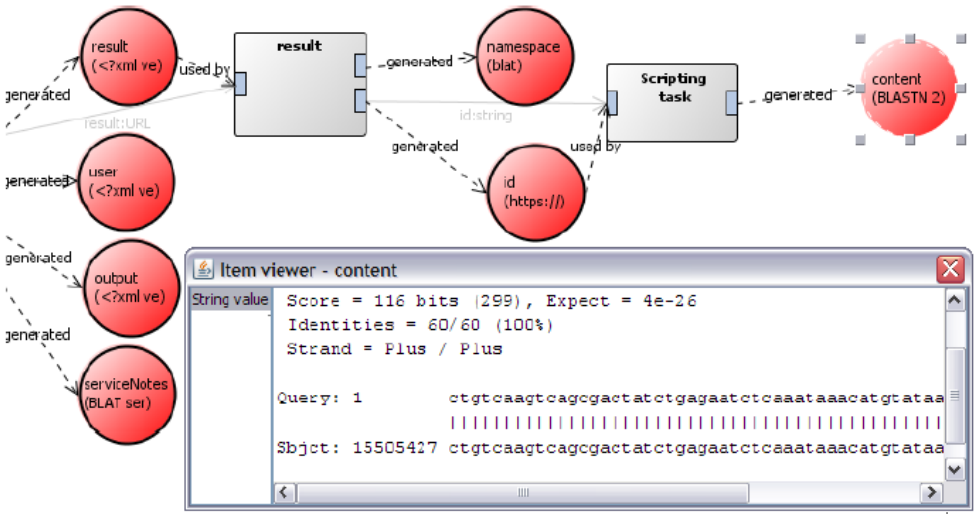

Abbildung 4.4 Die neuen Tasks "result" and "Scripting task" [WOV09]

Der neue Task "Scripting task" ermöglicht das Herunterladen des Inhalts über eine gesicherte Verbindung. Um den Task auszuführen, muss der Code vom Benutzer geschrieben werden. Der Benutzer hat sich für die Programmiersprache Perl entschieden. Danach führt der Benutzer den Task aus. Aber es kommt eine Fehlermeldung vom Programm, das der Benutzer geschrieben hat. Nach der Behebung des Fehlers kann dieser Task nochmal ausgeführt werden. Dieses Mal läuft der Task erfolgreich. Am Ende kann der Benutzer den kompletten Workflow nochmal laufen lassen.

4.2 Retry Scopes

BPEL ist die meistens beliebteste Sprache zur Prozessmodellierung für die Modellierung der Geschäftsprozesse und kommt ursprünglich aus der BPM- Domain. Das Hauptziel von BPEL ist, die Geschäftsprozesse als Orchestrierung der Services zu modellieren und auszuführen [EKU+10]. In diesem Fall wurden die Reaktionen auf die dynamischen Eigenschaften wie die Netzwerkverbindung während der Designphase von BPEL nicht genug berücksichtigt. Viele selten vorkommende Fehler sind im Prozessmodell nicht bedacht. Das hier vorgestellte Konzept bietet eine Annäherung an ein anpassungsfähiges und stärkeres Prozessmodell in der realen Welt. Die Aktivitäten können erneut angestoßen werden, wenn ein bestimmter Fehler auftritt. Das hier vorgestellte, neue definierte Modellelement ermöglicht und erweitert die <scope>-Aktivität, die auf Fehlersituationen in einer flexibleren Weise als die herkömmliche <scope>-Aktivität von BPEL reagieren kann.

4.2.1 Die zwei Szenarien

Um das Konzept besser zu verstehen, werden in diesem Abschnitt zwei Szenarien vorgestellt.

Beim ersten Szenario handelt es sich um einen Medikamententest in einem Prozess. Der Medikamententest besteht aus drei Testphasen. Der erste Schritt ist das Abholen der Ingredienzien. Der zweite Schritt wird die Mischung der Ingredienzien genannt und der letzte Schritt ist der Test der Mixtur. Die Syntax dieses Testprozesses kann in BPMN 1.2 repräsentiert werden (siehe die Abbildung 4.5) [EKU+10].

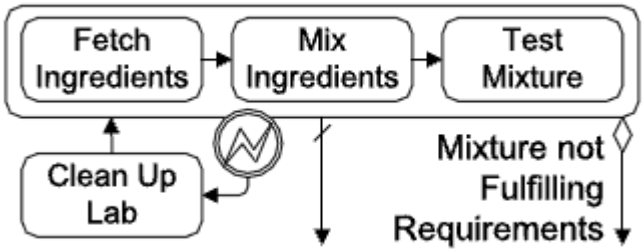

Abbildung 4.5 Der Medikamententest [EKU+10]

Der Prozess hat nach diesem Medikamententest zwei Alternativen fortzufahren. Eine Alternative wird ausgewählt, wenn die Mixtur die Anforderungen nicht erfüllen kann. Die andere davon wird ausgewählt, wenn der Test erfolgreich ist. Wie die Abbildung 4.5 gezeigt hat, sind die drei Testphasen gebündelt. Wenn ein Fehler während des Tests auftritt, muss das Labor sauber gemacht werden und es müssen auch alle Aktivitäten vom Test neu gestartet werden. Dieser Test kann als eine <scope>-Aktivität betrachtet werden. In diesem Prozess werden einigen Reparaturarbeiten gemacht, bevor diese <scope>-Aktivität neu versucht wird.

Das zweite Szenario unterscheidet sich von dem ersten. Ein Prozess kann dazu entworfen werden, einen Arbeiter einer Spedition zu unterstützen. In diesem Prozess werden alle Anweisungen der Navigation passend zur aktuellen Position des Liefernten berechnet. Wenn der Lieferant die richtige Route nicht verfolgt und sich dann mit dem Navigationsprozess in Verbindung setzt, kann der Navigationsservice abhängig von der aktuellen Position eine neue richtige Route berechnen. Dieser Prozess unterscheidet sich von dem ersten Prozess dadurch, dass der Prozess nicht den Anfangsort des Mitarbeiters berechnen muss, wenn ein Fehler auftritt. Die neue aktuelle Position kann als Input eingegeben werden und der Prozess kann dabei neu gestartet werden. Im Vergleich zu erstem Beispiel wiederholt sich die <scope>-Aktivität ohne irgendeine Reparaturarbeit.

4.2.2 Das Konzept von Retry/Rerun-Scopes

In den vorherigen zwei Abschnitten wurden die zwei verchiedenen Szenarien dargestellt und die grundlegenden Informationen über BPEL und Scopes eingeführt. Abhängig davon kann das Konzept jetzt erstellt werden. Zunächst sollen die Unterschiede zwischen den zwei Szenarien erklärt werden.

1. Aus dem ersten Szenario kann ein Verhalten als Retry definiert werden. Retry bedeutet, dass aufgrund eines auftretenden Fehlers einige Aktivitäten neu ausgeführt werden sollen, aber zuvor müssen einige Dinge "sauber" gemacht werden.

2. Aus dem zweiten Szenario kann ebenfalls ein Verhalten als Restart definiert werden. Restart hat fast die gleichen Eigenschaften wie Retry, aber es ist etwas einfacher aufgrund der Tatsache, dass die ausgeführten Aktivitäten nicht "sauber" gemacht werden müssen.

Das erste Szenario hat die Gemeinsamkeit mit der Operation von Reexecute gemein, dass einige Aufgaben gemacht werden müssen, bevor die Aktivität neu durchgeführt wird. Das zweite Szenario hat dann mit der Operation von Iterate gemein, dass keine Arbeit gemacht werden muss, bevor diese Aktivität wiederholt wird.

Das neue Konzept von Retry/Rerun-Scopes benutzt eine neue Aktivität nämlich <restart> innerhalb einer <catch>-Aktivität von einem Fault Handler. Das Attribut <times> der <restart>-Aktivität definiert die Häufigkeit der Wiederholung eines Fault Handlers im Fall eines bestimmten Fehlertyps: eine Endlosschleife kann dadurch auch vermieden werden. Insbesondere ist zu beachten, dass der Zähler nicht zunimmt, wenn die dazugehörende <scope>-Aktivität neu ausgeführt wird, sonder nur dann, wenn die <restart>-Aktivität neu ausgeführt wird. Der Zähler läuft bei jeder Ausführung der <restatt>-Aktivität mit der gleichen ID von Aktivitäten und auch der gleichen ID von Prozessen [EKU+10].

Die folgende Abbildung zeigt ein Beispiel, wie man mit dieser neuen vorgestellten <restart>-Aktivität das Verhalten von Retry darstellen kann.

```
<faultHandlers>
        <catch faultName="NoUserFound"?
                faultVariable="BPELVariableName">
                <sequence>
                        <compensate />
                        <wait />
                        <restart times="5" />
                </sequence>
        </catch>
</faultHandlers>
```

Listing 4.1 Fault Handler mit <restart>-Aktivität in RetryScope [EKU+10]

In der <catch>-Aktivität dieses Faulthandlers gibt es eine <sequence>-Aktivität, die auch die Aktivitäten von <compensate>, <wait> und <restart> enthält. <wait> dient dazu, dass die nachfolgende <restart>-Aktivität verzögert werden soll.

Für den RerunScope gibt es nur einen Unterschied zum RetryScope, wobei das Verhalten als Restart aus dem zweiten Szenario im RerunScope realisiert wird. In der <sequence>-Aktivität von dem Fault Handler im RerunScope wird nur die <compensate>-Aktivität nicht ausgeführt.

4.3 Dynamische Modifikation des Workflows

Wenn der Benutzer in manchen Situationen die Geschäftsprozesse ändern oder von einigen Stellen der gesamten Geschäftsprozesse abweichen möchte, können die Geschäftsprozesse nicht mehr wie vorher beschrieben ausgeführt werden. In diesem Abschnitt werden einige Fälle in Anlehnung an [LR00] vorgestellt, wo der Benutzer einen laufenden Workflow

modifizieren kann.

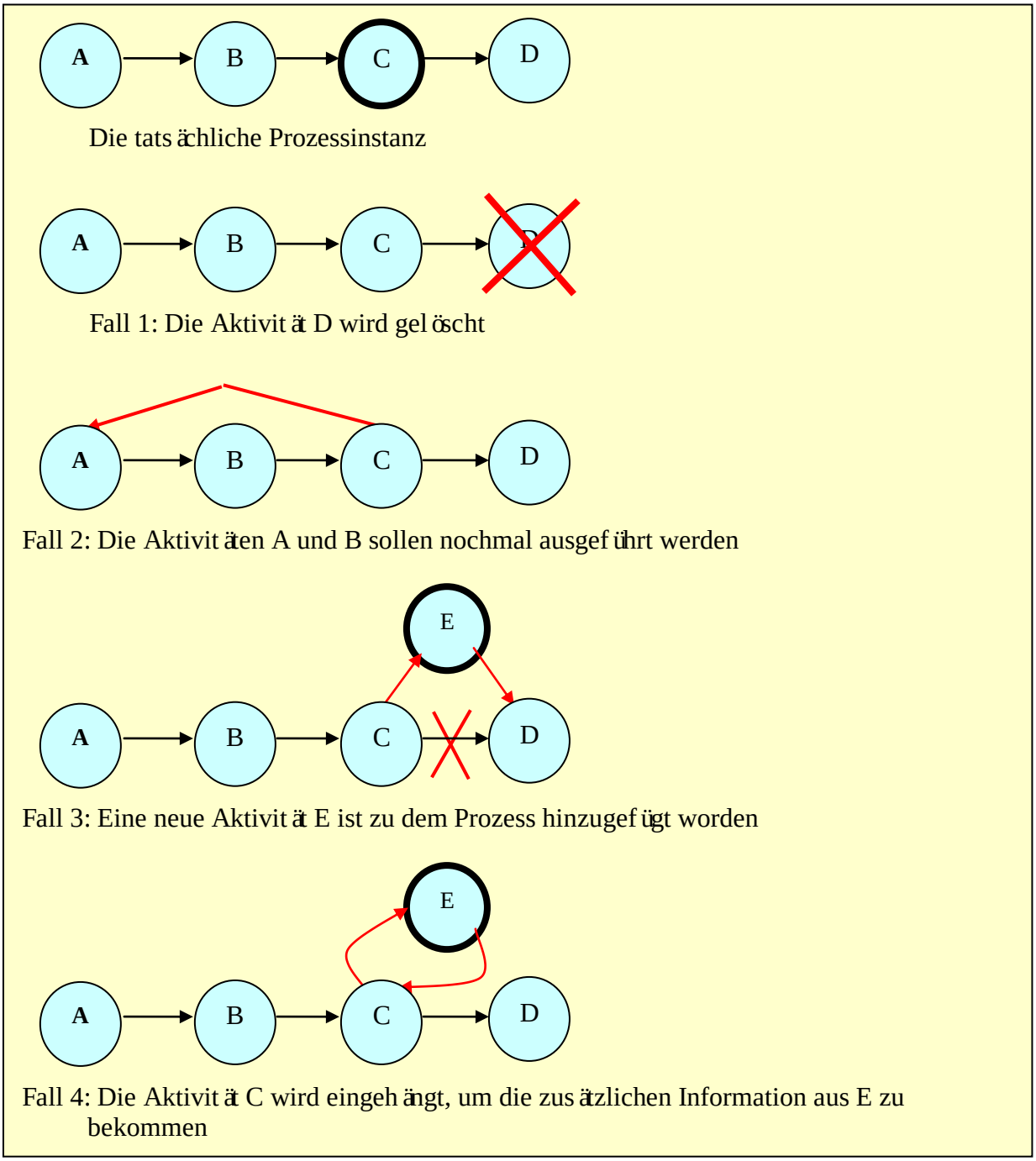

Abbildung 4.6 Modifikationen vom Workflow

Zu dem Zeitpunkt, zu dem die tatsächliche Prozessinstanz in der oben stehenden Abbildung gezeigt wird, bearbeitet der Benutzer gerade die Aktivität C. Im Fall 1 wird die Aktivität D aus dem gesamten Geschäftsprozess gelöscht, wenn diese Aktivität nicht nötig ist. Die Modifikation hier wird nach [LR00] "delete step" genannt. Der Fall 2 kann als eine Iterate-Modifikation betrachtet werden [LR00]. Während der Prozess läuft, kann der Benutzer bemerken, dass die Aktivität C die von ihm vorher erwünschten Ergebnisse nicht produzieren kann. Der Benutzer lässt deshalb um diesen Zeitpunkt die Aktivitäten A und B nochmal ausführen. Im Fall 3 wird die neue Aktivität E zwischen den Aktivitäten C und D zu den

Prozessen hinzugefügt. E kann von dem Benutzer oder anderen ausgeführt werden. Die Modifikation dieses Falls wird auch als "intermediate step" bezeichnet [LR00]. Im Fall 4 hängt der Benutzer die Aktivität C ein, weil er noch die zusätzlich benötigten Informationen aus der Aktivität E braucht. Die in diesem Fall relevante Modifikation wird nach [LR00] "inquiry" genannt.

5. Konzept für Iteration und Wiederholte Ausführung der Aktivitäten im WS-BPEL 2.0

Das Kapitel bezieht sich auf ein Konzept für Iteration und wiederholte Ausführung der Aktivitäten in WS-BPEL 2.0, die sich innerhalb von <sequence>-, <flow>-, <while>-, <repeatUntil>- und <forEach>-Aktivitäten befinden. Wiederholte Ausführung der Aktivitäten wird in diesem Kapitel Re-execution genannt.

Die Unterschiede zwischen dem Konzept dieser Arbeit und dem Konzept des Retry/Rerun-Scope aus dem Kapitel 4 sind im Folgenden beschrieben:

- In diesem Konzept wird keine neue Aktivität für WS-BPEL erzeugt. Bei Retry/Rerun wird eine neue Aktivität, nämlich die <restart>-Aktivität für WS-BPEL kreiert.

- Die Operationen Iteration und Re-execute werden nicht im Prozess modelliert, sondern von außen durch die ProcessInstanceMangement-API ausgeführt.

- Die beiden Operationen können an jeder beliebigen Stelle ausgeführt werden. Aber das Konzept von Retry/Rerun wird nur für die <scope>-Aktivität definiert.

Vorbedingungen für die Operationen Iteration und Re-execution:

Bevor die beiden Operationen durchgeführt werden können, muss der laufende Prozess zuerst angehalten werden (suspend).

Die Nachfolgeaktivitäten müssen dann auch ermittelt werden, weil die laufenden Nachfolgeaktivitäten beendet werden müssen.

In allen Abbildungen steht das rote Häkchen für die Aktivitäten, die schon erfolgreich ausgeführt wurden; ebenso zeigt der grüne Strich die Aktivitäten an, die zu diesem Zeitpunkt der Ausführung von der "iterate/reexecute"-Operation, noch aktiv sind. Der schwarze Pfeil zeigt die Aktivitäten, die von der "iterate/reexecute"-Operation nochmal durchgeführt werden sollen, an. Das rote Kreuz steht für die Aktivitäten, die nicht ausgeführt werden können, weil sie sich in einem toten Pfad befinden.

5.1 Die Unterschiede zwischen Iteration und Re-execution

Die Iteration soll nur als eine Schleife für die zu wiederholende Aktivität betrachtet werden. Es müssen keine zusätzlichen Aufgaben mehr erledigt werden, um die Aktivität nochmal auszuführen.

Bei der Re-execution dagegen müssen zusätzliche Aufgaben gemacht werden. Dabei müssen die Aktivitäten auf dem Pfad der zu wiederholenden Aktivität bis zur gerade laufenden Aktivität zuerst rückgängig gemacht werden. Das bedeutet, die Aktivität muss in dem gleichen Zustand sein (beziehungsweise in einem ähnlichen Zustand je nach

Kompensationslogik), wie zu dem Zeitpunkt, bevor sie ausgeführt wurde. Die zu wiederholende Aktivität soll die gleichen Werte für Partnerlinks und Variablen besitzen wie bei der vorherigen Ausführung und kann dann durch Re-execution nochmal ausgeführt werden. Für diese zusätzliche Aufgabe muss ein Mechanismus gefunden werden, um die Daten von Partnerlinks und Variablen zu speichern.

5.2 Iteration

In diesem Abschnitt werden einige Fälle für die Operation Iteration vorgestellt.

5.2.1 Iteration in Sequence

Zunächst wird das Konzept für die <sequence>-Aktivität vorgestellt. Für die <sequence>-Aktivität werden zwei Fälle im Folgenden zur Verfügung gestellt (siehe bitte die Abbildung 5.1 und 5.2).

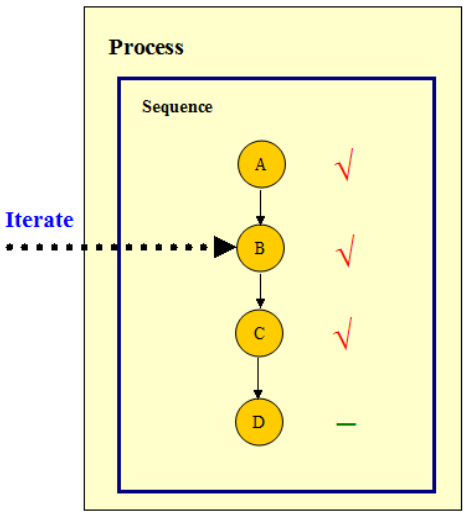

Abbildung 5.1 Iteration in Sequence, Fall 1

Die in diesem Beispiel gezeigte <sequence>-Aktivität ist eine Sequence mit 1-n beendeten und einer aktiven Aktivität. Die Iteration wird von einer der beendeten oder aktiven Aktivitäten ausgeführt. Wie die oben stehendene Abbildung zeigt, sind die Aktivitäten A, B und C schon fehlerfrei ausgeführt worden, D ist zurzeit noch aktiv und B soll nochmal durch der Iterate-Operation ausgeführt werden. Bevor die *Iterate* Operation ausgeführt werden kann, muss man den Prozess überprüfen, ob B oder Nachfolgeaktivitäten noch aktiv sind. Wenn dies nicht der Fall ist, kann B nochmal ausgeführt werden. Wenn B oder eine Nachfolgeaktivität noch aktiv ist, muss sie erst beendet werden bevor B erneut ausgeführt werden kann.

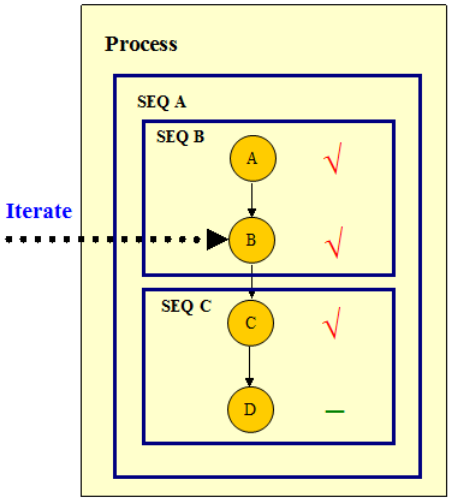

Abbildung 5.2 Iteration in Sequence, Fall 2

Die in diesem Fall beschriebene <sequence>-Akivität besteht aus mehreren <sequence>-Akivitäten und jede Sequence davon ist auch eine Sequence mit 1-n beendeten und einer aktiven Aktivität. Die Iterate-Operation wird von einer der beendeten oder aktiven Aktivitäten ausgeführt. In der Abbildung 5.2 besteht der Prozess aus insgesamt drei <sequence>-Aktivitäten, nämlich SEQ A, SEQ B und SEQ C. SEQ A besteht auch aus SEQ B und SEQ C. Die Aktivitäten A und B in SEQ B wurden schon erfolgreich ausgeführt und SEQ B wurde deshalb auch vollständig ausgeführt. SEQ C ist jetzt aktiv, weil das in C enthaltene D noch aktiv ist. Wenn eine Iteration von B aus gemacht werden soll, müssen wieder alle aktiven Aktivitäten im Pfad mit B beendet werden. Im Beispiel sind das C und SEQ C. B kann nicht ohne Weiteres wiederholt ausgeführt werden, da SEQ B bereits beendet ist. Deshalb muss SEQ B erneut ausgeführt werden. Innerhalb der Sequence dürfen dann nur die Aktivitäten ab dem Iterationspunkt ausgeführt werden (im Beispiel nur Aktivität B).

5.2.2 Iteration in Flow

Im Folgenden wird das Konzept für die <flow>-Aktivität in vier Fällen vorgestellt (siehe bitte die folgende Abbildungen 5.3, 5.4, 5.5 und auch 5.6).

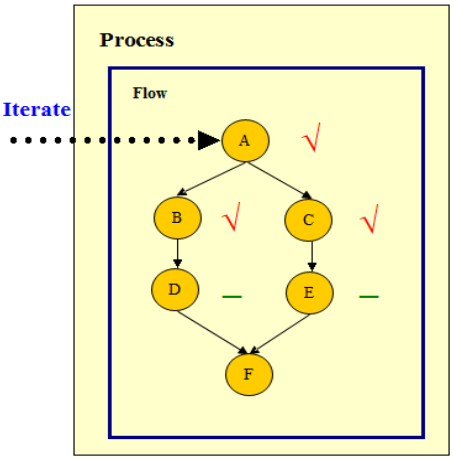

Abbildung 5.3 Iteration in Flow, Fall 1

In diesem Beispiel wird eine Flow-Aktivität von der Iterate-Operation nochmal ausgeführt. Dieser Prozess dieses Falls besteht aus einer <flow>-Aktivität, die die Aktivitäten A, B, C, D, E und F enthält. A, B und C wurden schon erfolgreich ausgeführt. D und E sind zurzeit noch aktiv. F ist inaktiv, weil F noch auf die Nachrichten aus D und E warten muss. Um die Iterate -Operation bei einer Aktivität zu ermöglichen, müssen erst die Aktivität selbst und/oder ihre Nachfolgeaktivitäten beendet werden (hier die zwei aktiven Aktivitäten D und E). Dann kann A durch Iterate wiederholt ausgeführt werden.

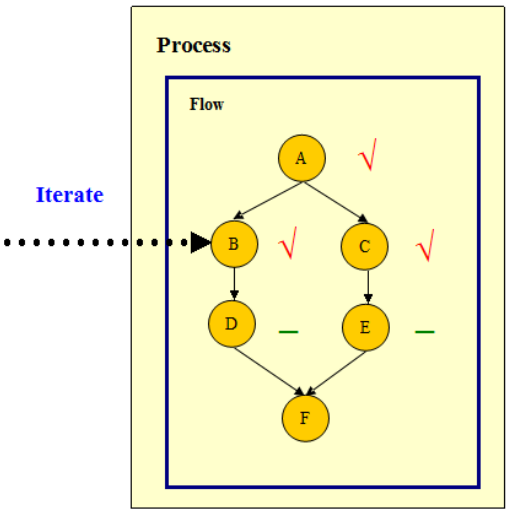

Abbildung 5.4 Iteration in Flow Fall 2

Bei diesem Beispiel handelt es sich um die Ausführung einer Iteration-Operation in parallelen Pfaden von einer <flow>-Aktivität. Dieser Prozess besteht aus einer <flow>-Aktivität, die die Aktivitäten A, B, C, D, E und F enthält. A, B und C wurden schon erfolgreich ausgeführt. D und E sind zurzeit noch aktiv. F ist inaktiv, weil F noch auf D und E warten muss. Um B neu auszuführen, muss die hinter B stehende aktive Aktivität D gefunden und dann beendet werden. Die Aktivitäten in dem anderen Zweig von dieser <flow>-Aktivität können ohne Einschränkung weiter laufen (im Beispiel Aktivität E). F beginnt zu laufen, sobald der Kontrollfluss in beiden Zweigen bei F ankommt.

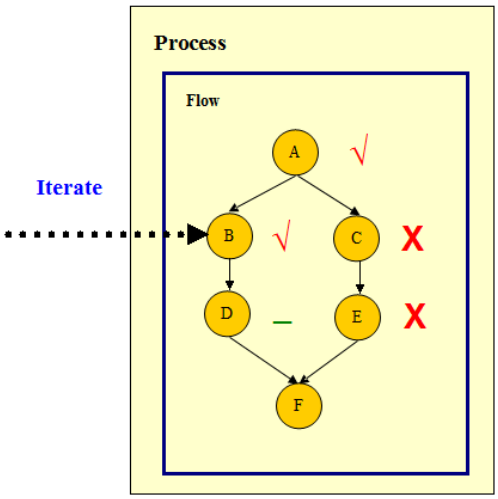

Abbildung 5.5 Iteration in Flow, Fall 3

In diesem Beispiel wird eine Flow-Aktivität von der Iterate-Operation nochmal ausgeführt. Der einzige Unterschied von Fall 3 in der Abbildung 5.5 zu Fall 2 in der Abbildung 5.4 ist, dass ein toter Pfad (Dead Path) der Transition Condition der ausgehenden Links von Aktivität A entstanden ist. Der Zweig zwischen C und F braucht wegen der Dead Path Elimination nicht besonders berücksichtigt zu werden. Es muss das gleiche wie im Fall 2 gemacht werden. (siehe die Abbildung 5.4)

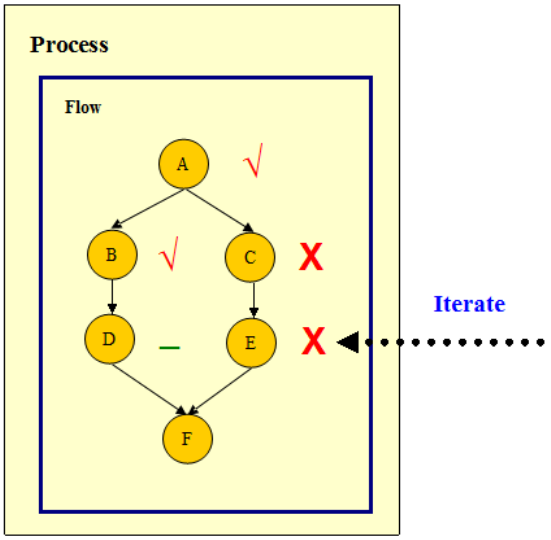

Abbildung 5.6 Iteration in Flow, Fall 4

Dieses Beispiel handelt von der Ausführung einer Iterate-Operation in toten Pfaden einer <flow>-Aktivität. Der Fall 4 beschreibt, wie eine Aktivität E von der Iterate-Operation nochmal ausgeführt werden kann. Die Iteration selbst ist von einer Aktivität in einem toten Pfad nicht erlaubt, da dadurch Ausführungshistorien von nicht zusammenhängenden Aktivitäten entstehen können. Allerdings kann die Iteration dann automatisch von einer sinnvollen Vorgängeraktivität durchgeführt werden. Zuerst müssen die zuvor schon evaluierten Pfade (Links) rückwärts bis zu der Aktivität, die schon erfolgreich durchgeführt wurde, durchlaufen werden. Im Beispiel kann die Aktivität A dadurch gefunden werden. Von dieser Aktivität aus sucht man alle aktiven Nachfolger und beendet sie (hier: Aktivität D). Es wird jetzt die Aktivität erneut ausgeführt, die durch das rückwärts gerichtete Durchlaufen des Pfades gefunden wurde.

5.2.3 Iteration in While, RepeatUntil und forEach

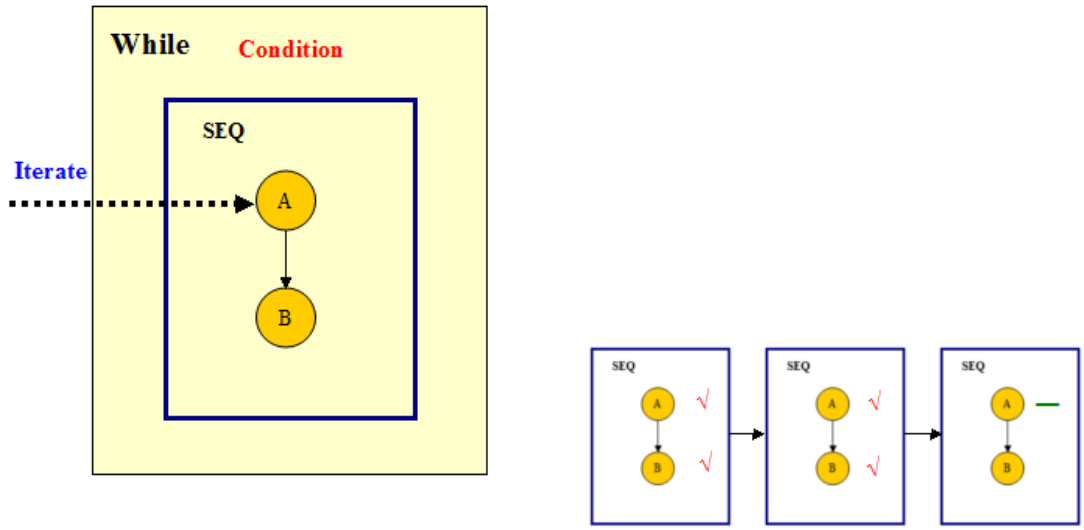

Abbildung 5.7 Iteration in While

In diesem Abschnitt wird die Iterate-Operation innerhalb einer While-Aktivität vorgestellt. Die <while>-Aktivität enthält eine <squence>-Aktivität und deshalb funktionert die Iterate-Operation in While genauso wie in der Sequence-Aktivität. Es muss aber darauf geachtet werden, dass es zu Endlosschleifen kommen kann, wenn Variablen so durch die Iterate-Operation verändert werden, dass die While-Bedingung niemals false werden kann.

Die Iterate-Operation in der <repeatUntil>-Aktivität arbeitet fast auf die gleiche Weise wie <while>. Es muss aber richtig aufgepasst werden, dass die Bedingung von <repeatUntil> durch mehrmalige Ausführung der "rerun"-Operation verletzt werden kann, wenn zum Beispiel ein Zähler als die Bedingung für die <repeatUntil>-Aktivität dient.

Für die <forEach>-Aktivität müssen zuerst die folgenden zwei Fälle unterschieden werden. Wenn der Wert des Attributs "parallel" "no" ist, ist diese <forEach>-Aktivität eine serielle <forEach>-Aktivität und kann die Iterate-Operation wie in <while> von einer bestimmten Aktivität ausgeführt werden, weil sich die serielle <forEach>-Aktivität im Grunde nicht von While und RepeatUntil unterscheidet. Allerdings kann die Aktivität selbst das Hochzählen des Zählers übernehmen. Deshalb muss man nicht auf die ForEach-Bedingung achten wie bei While und RepeatUntil. Wenn dieser Attributwert "yes" ist, ist diese <forEach>-Aktivität ein paralleles <forEach>. Die enthaltene <scope>-Aktivität wird gleichzeitig ausgeführt. Es werden mehrere Instanzen des Scopes gleichzeitig gestartet. Für eine Iteration muss die richtige Scope-Instanz gefunden werden. Dazu muss die Iterate/Re-execute-Operation um einen entsprechenden Parameter erweitert werden.

5.3 Re-execution

Die Unterschiede zwischen Iteration und Re-execution wurden schon in dem Abschnitt 5.1 beschrieben. Hier wird nur die Re-execution von einer Aktivität innhalb von einer <sequence>-Aktivität, die schon im Abschnitt 5.2.1 als das erste Beispiel benutzt wurde,

erläutert. Die Grundkonzepte von Re-execution der Aktivität sind gleich wie Iterate in Sequence, Flow, While usw. Zusätzlich ist nur zu berücksichtigen, wie die Aktivitäten auf dem Pfad der zu wiederholenden Aktivität bis zur gerade laufenden Aktivität rückgängig gemacht werden. Um dies zu ermöglichen, müssen die Partnerlinks und die Variablen dieser Aktivität abgespeichert werden und in manchen Fällen muss die Kompensierung auch ausgeführt werden. Weil die Re-execution an jeder beliebigen Stelle eines Prozesses durchgeführt werden kann, müssen die Werte von den Partnerlinks und den Variablen von jeder Aktivität vor ihrer Ausführung gespeichert werden. Wenn eine Re-exection durchgeführt wird, müssen die Werte von den beiden wieder abgeholt und zu dieser zu wiederholenden Aktivität übergeben werden. Dafür können die Snapshots verwendet werden, um sie zu speichern. In den folgenden Abbildungen stellt der Block in lila die Snapshots dar, die vor der Ausführung von Aktivitäten schon abgespeichert werden sollen. Wir nehmen hier drei Beispiele für Re-execute.

5.3.1 Einfache Re-execution für die Aktivität ohne Kompensierung

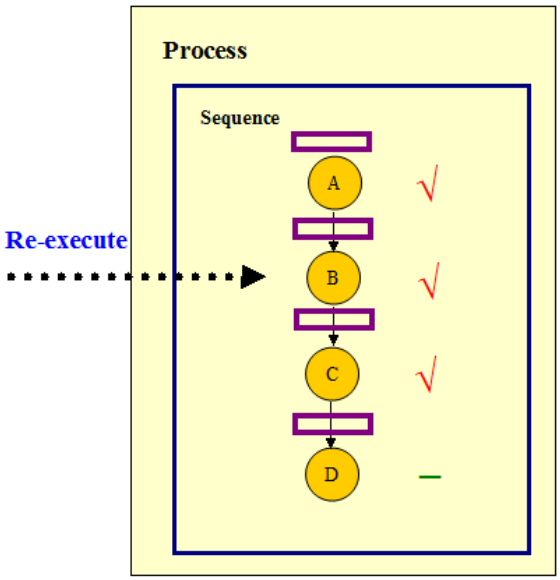

Abbildung 5.8 Re-execution in Sequence ohne Kompensierung

In diesem Beispiel gibt es nur die vier Aktivitäten A, B, C und D. Sie befinden sich alle innerhalb einer <sequence>-Aktivität. Die zusätzlichen Aufgaben für Re-execute im Vergleich zu Iterate sind nur die Snaptshots für alle A, B, C und D zu erzeugen. Wenn B nochmals durch Re-execute auszuführen ist, werden die Werte von den Partnerlinks und den Variablen aus dem Snapshot abgeholt und wieder an B übergeben, die schon vor der ersten Ausführung von B abgespeichert wurden.

5.3.2 Re-execution für die Aktivität mit Kompensierung

In diesem Abschnitt wird Re-execute für die Aktivität mit Kompensierung vorgestellt. Die Kompensierung muss für bereits ausgeführte Aktivitäten gemacht werden, die durch das Re-execute erneut ausgeführt werden sollen. Die Kompensierung kann allerdings nur bei einer erfolgreich ausgeführten <scope>-Aktivität und <invoke>-Aktivität gemacht werden. Das bedeutet, wenn auf dem Pfad der zu wiederholenden Aktivität und den gerade aktiven

Aktivitäten schon beendete Scopes und Invokes liegen, dann müssen die Scopes und Invokes kompensiert werden. Hier werden zwei Beispiele betrachtet. Bei einem davon handelt es sich um Re-execute für die Aktivitäten in der <scope>-Aktivität und das andere beschreibt über die Re-execute speziell für die <invoke>-Aktivität.

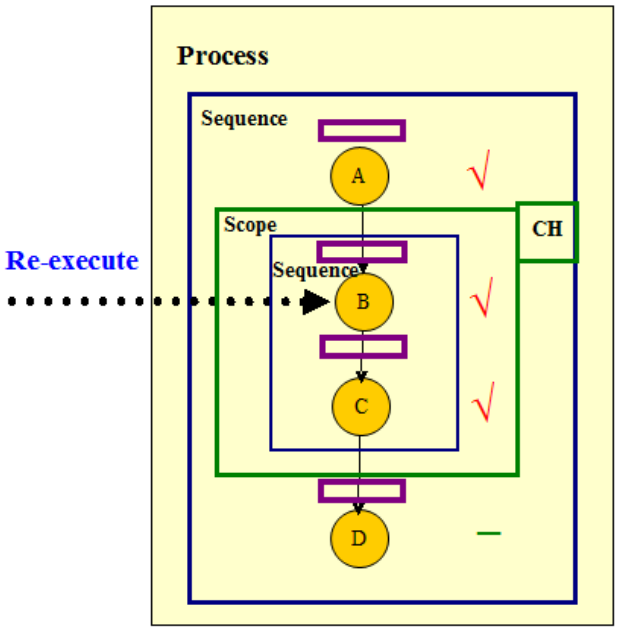

Abbildung 5.9 Re-execution in Sequence mit Kompensierung Fall 1

Die Abbildung 5.9 zeigt die Aktivitäten A, B, C und D. B und C befinden sich in einer <sequence>-Aktivität und diese Sequence existiert auch in einer <scope>-Aktivität mit einem vordefinierten Compensation Handler. B muss durch Re-execute erneut ausgeführt werden. Dabei müssen die Snapshots wie im Abschnitt 5.3.1 wieder für jede Aktivität A; B; C und D erzeugt werden. Wenn die Operation Re-execute ausgeführt wird, wird der Compensation Handler sofort aufgerufen werden, um das schon beendeten B und C zu kompensieren. Dann werden die Werte für Parnterlinks und Variablen aus dem zu B gehörenden Snapshot wieder an die Aktivität B übergeben. Jetzt läuft B nochmals. Sollte eine <scope>-Aktivität keinen Compensation Handler besitzen, wird der implizite Compensation Handler aufgerufen.

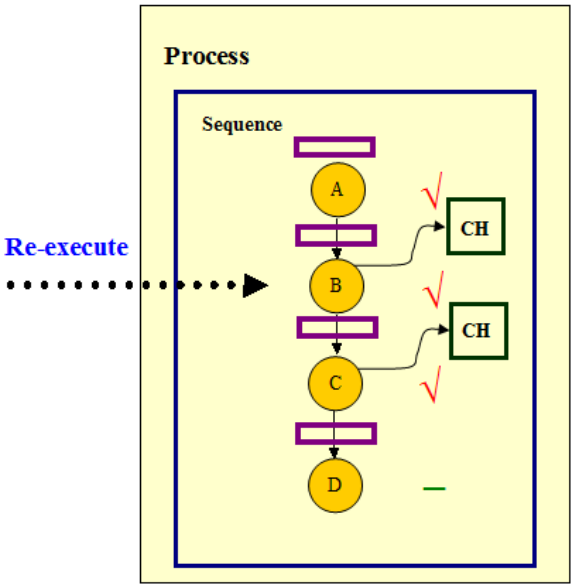

Abbildung 5.10 Re-execution in Sequence mit Kompensierung Fall 2

Die Abbildung 5.10 zeigt die Aktivitäten A, B, C und D. B und D sind jeweils die <invoke>-Aktivitäten, die auch eigene Compensation Handler haben. Während dieser Prozess läuft, müssen auch die Snapshots für jede Aktivitäten gemacht werden. Wenn die Operation Re-execute aufgerufen wird, soll C zuerst und dann B kompensiert werden. Das heißt die Kompensierung wird in umgekehrter Ausführungsreihenfolge durchgeführt. Jetzt werden die Werte für Partnerlinks und Variablen aus dem Snapshot wieder geladen. Jetzt kann die B endlich nochmals ausgeführt werden. In einigen Fällen besitzt die <invoke>-Aktivität auch möglicherweise keinen Compensation Handler: in diesem Fall läuft die Re-execute Operation auf die gleiche Weise wie es der Abschnitt 5.3.1 gezeigt hat.

6. Realisierung eines Prototyps

Im vorhergehenden Kapitel wurde ein Konzept für die Iteration und wiederholte Ausführung der Aktivitäten vorgestellt. In diesem Kapitel werden zunächst die Anforderungen an die Realisierung (Abschnitt 6.1) beschriebn. Anschließend wird die Entwicklungsumgebung (Abschnitt 6.2) eingeführt. Zuletzt wird die prototypische Umsetzung des Konzepts aus dem Kapitel 5 im Abschnitt 6.3 erläutert. Der Prototyp hat die Einschränkung, dass die Aktivitäten nur innerhalb einer <sequence>-Aktivität durch Iterate und Re-execute nochmal ausgeführt werden können, wie der erste Fall in dem Abschnitt 5.2.1 gezeigt hat. Für die Aktivitäten in anderen Fällen können die beiden Operationen von Apache ODE noch nicht unterstützt werden.

6.1 Anforderungen an die Realisierung

Die Anforderungen werden in diesem Abschnitt beschrieben, die im Rahmen von den praktischen Umsetzung der Aufgabendefinition im Kapitel 1 erfüllt werden sollen. Hier werden die Anforderungen im Einzelnen beziehungsweise in den zwei Operationen "Iterate" und "Reexecute" dargestellt.

Die Operation "Iterate" funktioniert wie eine Schleife. Sie greift auf den auf der BPEL-Engine Apache ODE laufenden Prozess zu und führt dann eine bestimmte Aktivität nochmal aus. Bevor diese Operation Iterate ausgeführt werden kann, muss der Prozess zuerst durch die Operation "Suspend" ausgesetzt werden. Die Operation "Suspend" ist schon der Apache ODE für den Benutzer angeboten worden. Zwei Parameter sind nötig für die Operation Iterate.
Einer davon ist die Prozessinstanz-ID, die zu jeder Prozessinstanz eindeutig zuordenbar ist, und der andere ist ein XPath-Ausdruck. Mittels des XPath-Ausdrucks kann die neue auszuführende Aktivität in der Apache ODE gefunden werden. Der Benutzer gibt die zwei Parameter in eine SOAP-Nachricht ein und sendet die Nachricht zu den Prozess. Die BPEL-Engine bearbeitet diese Nachricht, sendet dem Benutzer eine Nachricht als Response zurück und führt die angegebene Aktivität neu aus.

Die Operation "Reexecute" unterscheidet sich von der Operation "Iterate". Dabei müssen die Aktivitäten auf dem Pfad der zu wiederholenden Aktivität bis zur gerade laufenden Aktivität zuerst rückgängig gemacht werden. Das bedeutet, die Aktivität muss in dem gleichen Zustand sein (beziehungsweise. in einem ähnlichen Zustand je nach Kompensationslogik), wie zu dem Zeitpunkt, bevor sie ausgeführt wurde.

6.2 Die Entwicklungsumgebung

Die Apache ODE in Version 1.3.4 wird in dieser Arbeit erweitert. Als eine Web-Container-Umgebung wird Apache Tomcat in Version 6.0.26 verwendet. SoapUI in Version 3.5.1 wird als die Benutzerschnittstelle von Webservices benutzt. Der Code der Arbeit wird in Eclipse 3.5.2 mit Java 6 geschrieben. Apache ActiveMQ als eine nachrichtenorientierte Middleware (MOM) in Version 5.3.0 und Maven in Version 2.2.1 als ein Build-Management-Werkzeug werden während der Entwicklungszeit verwendet. SimTechODE Auditing wird zur Überwachung des BPEL-Prozesses benutzt werden. Dadurch können die ProzesseinstanzID, der Ablauf des Prozesses und auch Informationen wie der Status, die Ereignisse über alle im Prozess enthaltenen Aktivitäten in GUI präsentiert werden.

6.3 Erweiterung der Apache ODE

In diesem Abschnitt wird die Erweiterung der Apache ODE in folgenden drei Teilen vorgestellt.

6.3.1 XPathParser

Zuerst wird XPath in diesem Abschnitt kurz vorgestellt. Die XML Path Language (XPath) ist eine vom W3C entwickelte Abfragesprache, mit der die Teile eines XML-Dokumentes adressiert werden können [WXP].

Wie der Abschnitt 3.4 gezeigt hat, werden die übergebenen BPEL-Prozesse zuerst vom Compiler überprüft und zum ODE-Objekt-Modell kompiliert. Deshalb gibt ein passendes Objekt, das für jede Aktivität für jeden Prozess und für jeden Handler des ODE-Compilers erzeugt wird. Um die beiden Operationen Iterate und Reexecute zu implementieren, muss zuerst die Aktivität, die nochmal ausgeführt werden muss, beziehungsweise. das zu dieser Aktivität passende Objekte im ODE-Objekte-Modell durch zwei Parameter *xpath* und *Prozessinstanz-ID* gefunden werden. Dafür wird eine neue Klasse nämlich *XPathParser.java* geschrieben. Das Klassendiagramm von dieser Klasse wird in der folgenden Abbildung 6.1 gezeigt.

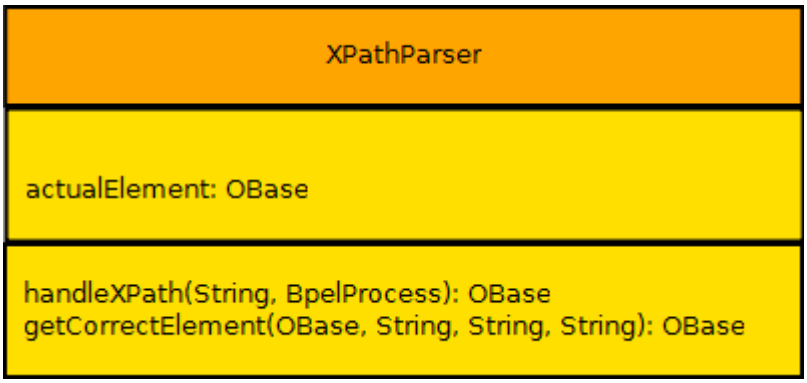

Abbildung 6.1 Klassendiagramm von XpathParser.java

Der Hauptteil in dieser Klasse ist die Methode *handleXPath*, die auch eine Methode *getCorrectElement* enthält. Die Methode *getCorrectElement* kann die richtige OActivity mittels der Prioraktivität *priorActivity*, der jetzt zu bearbeitenden Aktivität *activity* und des Suffixes dieser Aktivität *activitySuffix* finden. Die Methode *getCorrectElement* wird immer rekursiv aufgerufen, bis die Zielaktivität in Form des ODE-Objekte-Modells gefunden werden kann. In Listing 6.1 wird ein Ausschnitt der Klasse XpathParser.java gezeigt.

```
public class XPathParser {
      static OBase actualElement;
      public  OBase handleXPath(String xpath, BpelProcess process){
      String[] values = xpath.split("/");
      String priorActivity = null;
      actualElement = process.getOProcess();
      for (String value: values){
       int number = value.indexOf("[");
            String activity = value;
            String activitySuffix = null;
            if(number != -1){
            activity = value.substring(0, number);
            activitySuffix=value.substring(number+1,value.length()-1);
            }
            if (activity.compareTo("") != 0){
                  if(activity.compareTo("process")!= 0){
                        try{

      actualElement=getCorrectElement(actualElement,priorActivity,
                        activity, activitySuffix);
                        }
                        catch(Exception e){
                              e.printStackTrace();
                        }
                  }
                  priorActivity = new String(activity);
         }
     }
         return actualElement;
  }
```

Listing 6.1 Der Ausschnitt aus dem Klassendiagramm von XpathParser.java

Der eingegebene Xpath-Ausdruck wird zuerst durch das Zeichen / aufgeteilt. Jeder Teil davon wird dann durch die Methode *getCorrectElement* hintereinander analysiert. Um die Klasse besser zu verstehen, wird hier ein Beispiel eines Xpath-Ausdrucks benutzt. Der Xpath-Ausdruck ist "/process/sequence[1]/assign[1]". Als Xpath-Ausdruck ist der erste abgetrennte Teil dann "*process*". Der Anfangswert von *actualElement* ist dann *OProcess*. Weil der Wert von *activity* jetzt "*process*" ist, kann die Methode *getCorrectElement* nicht aufgerufen werden. Der Wert von *priorActivity* ist jetzt "*process*" und es folgt der nächste zu analysierende Teil des Xpath-Ausdrucks, beziehungsweise ist "sequence" der Wert von *activity*. Der Wert von activitySuffix ist 1, das heißt, die erste <sequence>-Aktivität im Prozess soll gefunden werden. Jetzt sind die erforderlichen Parameter für die Methode *getCorrectElement* vollständig. Weil in der Apache ODE zu jedem *OProcess* ein *OSope* als Processscope hinzugefügt wird, kann das Objekt im ODE-Objekte-Modell in diesem *OSope* gefunden werden (Siehe Listing 6.2). Das Ergebnis ist, dass OSequence als *actualElement* gefunden wird.

```
else if (priorActivity.compareTo("process") == 0){
                if(activity.compareTo("faultHandlers")==0){
               result = ((OProcess)element).processScope.faultHandler;
                } else if(activity.compareTo("eventHandlers")==0){
               result = ((OProcess)element).processScope.eventHandler;
                } else {
                result = ((OProcess)element).processScope.activity;
                }
     }
```

Listing 6.2 Suche nach der Kindaktivität von OProcess in der Methode *getCorrectElement*

Der Wert von *priorActivity* ist auf "sequence" gesetzt. Jetzt wird der nächste Teil "assign" im Xpath-Ausdruck gesucht. Die nötigen Parametern von *activity* und a*ctivitySuffix* für die Methode *getCorrectElement* sind "assign" und 1. Das folgende Listing 6.3 zeigt, wie das richtige Objekt mit den oben beschriebenen Parametern zu finden ist. Ein *OAssign* mit richtigem Suffix wird jetzt als das Ergebnis gefunden.

```
public OBase getCorrectElement(OBase element, String priorActivity,
String activity, String activitySuffix){
        OBase result = element;
           int activityNumber = 1;
           if (activitySuffix!=null&& activitySuffix.compareTo("")!=0){
                activityNumber = new Integer(activitySuffix);
           }
           int i = 0;
           if (priorActivity.compareTo("sequence")==0){
           for(OActivity childActivity :((OSequence)element).sequence){
                      if(activity.compareTo("assign")==0){
                            if (childActivity instanceof OAssign){
                                  i++;
                                  if(i == activityNumber){
                                        result = childActivity;
                                        break;
                                  }
                            }
```

Listing 6.3 Suche nach bestimmtem OAssign in Sequence in der Methode *getCorrectElement*

6.3.2 Iterate

In diesem Abschnitt wird die Implementierung von der Operation *Iterate* beschrieben. Es gab einige Überlegungen, wie man mittels der Iterate-Operation die bestimmten Aktivitäten erneut ausführen kann. Es muss überlegt werden, ob das BpelRuntimeContextImpl-Objekt, das schon von dieser Prozessinstanz erzeugt wurde, noch einmal erzeugt werden muss. Wie eine Kindaktivität innerhalb einer <sequence>-Aktivität neu ausgeführt wird und wie eine neue <sequence>-Aktivität zur Laufzeit hinzugefügt wird, muss auch bedacht werden. Was muss mit den Aktivtäten getan werden, die durch *suspend*-Operationen nicht mehr ausführbar sind, aber sich noch in der *ExecutionQueue* befinden? Der Ablauf der Erweiterung der Apache ODE um die *Iterate*-Operation ist in Abbildung 6.2 durch ein UML-Sequenzdiagramm

graphisch dargestellt. Mittels des Sequenzdiagramms können die Interaktionen, die den Nachrichten- und Datenaustausch zwischen mehreren Kommunikationspartnern umfassen, besser ausgedrückt werden.

Dieses Sequenzdiagramm zeigt, dass der Wissenschaftler zuerst eine der ODE Nachricht zur erneuten Ausführung einer Aktivität schickt und gleich auch eine Antwort von der ODE bekommt. Diese Nachricht wird dann von der ODE verarbeitet, dabei kann die richtige Aktivität mittels der in der Nachricht existierenden Parametern durch die Methode *handleXPath* in der Klasse *XPathParser* gefunden werden. Ein Job kann jetzt für Iterate erstellt werden und wird dann in vielen Arbeitschritten erledigt. Die ausführlichen Details über die Implementierung von Iterate werden in den folgenden Abschnitten beschrieben.

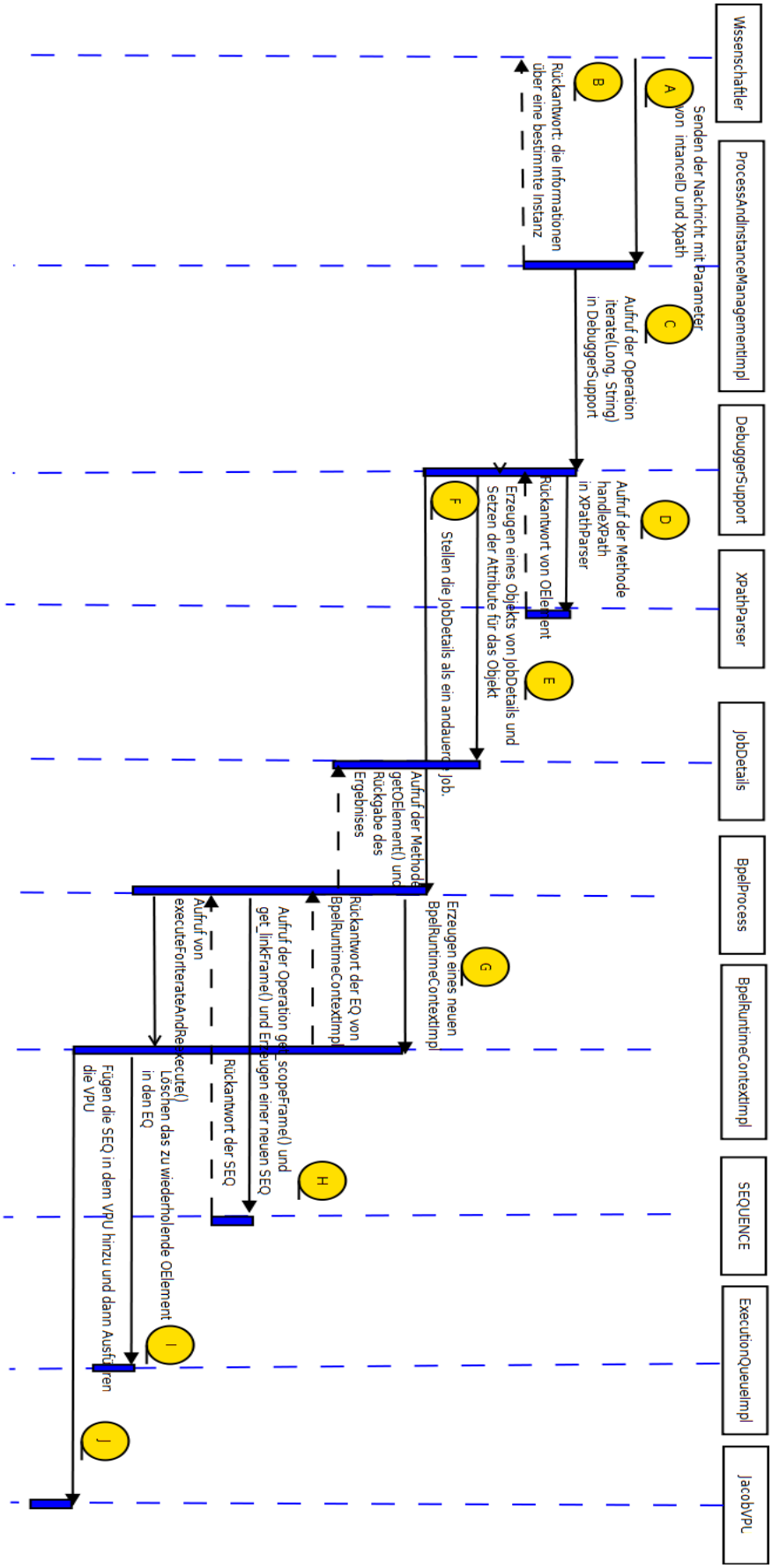

Abbildung 6.2 Sequenzdiagramm für Iterate-Operation

ProcessAndInstanceManagementImpl

Wie der Abschnitt 3.7 vorgestellt hat, hat die BPEL Management API in der Apache ODE zwei Teile. Einer davon ist das *Process Management* Interface, das die Funktionalität anbietet, um die zu den Prozessen gehörenden Instanzen zu verwalten (siehe die Abbildung 6.3). Der andere ist das *Instance Management* Interface. Das File pmapi.wsdl bietet einen Web Service, das es den in den beiden Interfaces definierten Operationen ermöglicht, auf die Prozesse und Instanzen in der ODE-Engine zuzugreifen. *ProcessAndInstanceManagementImpl* implementiert eine API, die Operationen zur Verfügung stellt, mit denen der Prozess und die Ausführung von diesem Prozess von außen verfolgt und beeinflusst werden können [Ste08]. Die von dieser Klasse zur Verfügung gestellten Funktionen sind beispielweise *suspend*, *resume* und *terminate*. Für die Implementierung im Bereich der BPEL Management API sind drei Aufgaben zu erledigen.

- Die Operation *Iterate* muss im InstanceManagement.java-Interface hinzugefügt werden.

- Die Klasse ProcessAndInstanceManagementImpl.java, die InstanceManagement implementiert, muss auch um die Operation Iterate erweitert werden.

```
public InstanceInfoDocument iterate(final Long iid, String xpath)
    throws ManagementException {
    getDebugger(iid).iterate(iid, xpath);
    return getInstanceInfo(iid);
    }
```

Listing 6.4 Die Iterate-Operation in ProcessAndInstanceManagementImpl.java

- Alle abstrakten Definitionen und konkreten Beschreibungen von *Iterate*, z.B.<message>, <portType>, <operation> und <binding>, müssen im File pmapi.wsdl eingetragen werden.

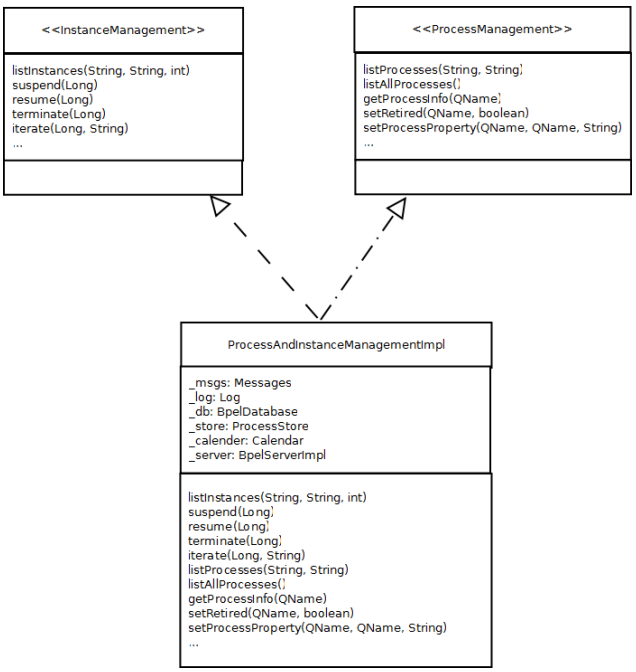

Abbildung 6.3 Klassendiagramm für die Verwaltung von Prozessen und Instanzen

Nach der dritten Aufgabe kann die *iterate* Operation im InstanceManagementSOAP12Binding wie in Abbildung 6.4 gezeigt aufgerufen werden.

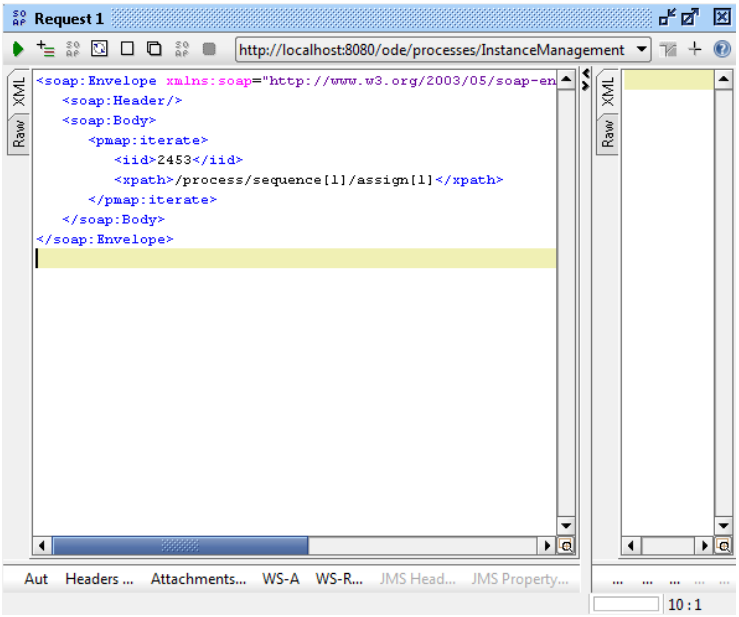

Abbildung 6.4 Request von Iterate in SoapUI

DebuggerSupport

Zuerst müssen der neue JobType beziehungsweise ITERATE in Scheduler.java, ein neues Attribut, nämlich *oelement*, und die getter- und setter-Methode für *oelement* in der Klasse *JobDetails.java* hinzugefügt werden. Alles, was die Iterate-Operation in dem Prozess tut, wird innerhalb einer Transaktion der Datenbank ausgeführt. Durch eine Verbindung mit der Datenbank kann ein Objekt des Typs ProcessInstanceDAO bekannt sein. Um die Operation *Iterate* auszuführen, muss zuerst entschieden werden, ob der Prozess zu dem Zeitpunkt schon durch die Operation *Suspend* pausiert wurde. Wenn der Zustand der Prozessinstanz *suspended* ist, wird durch den Parameter *Xpath* die richtige Aktivität im Form von OBase mittels des Aufrufs von der Operation *handleXPath()* in der *XPathParser* Klasse gefunden werden. Dann wird ein neues Objekt der *JobDetails*-Klasse erzeugt und es sollen auch Attribute wie JobType, IntanceId, ProcessId und OElement gesetzt werden. Wichtig ist, das neue erzeugte *JobDetails*-Objekt als einen andauernden Job in dem Ablaufplan der BPEL-Engine anzusetzen. Der Job wird dann gleich von der ODE-Runtime durchgesetzt werden, indem der Wert von *Date* auf *null* gesetzt ist (siehe Listing 6.5)

```
OBase element= new XPathParser().handleXPath(xpath, _process);
JobDetails we = new JobDetails();
we.setType(JobType.ITERATE);
we.setInstanceId(iid);
we.setProcessId(instance.getProcess().getProcessId());
we.setOElement(element);
_process._engine._contexts.scheduler.schedulePersistedJob(we,null);
```

Listing 6.5 Ausschnitt aus der Klasse DebuggerSupport.java

BpelProcess

In der Klasse BpelProcess wird die Iterate-Operation weiterbearbeitet. Die in dieser Klasse existerende Methode handleJobDetails wird dafür zur Verfügung gestellt. Eine Instanz von der Klasse JobDetails ist als einziger Parameter für diese Methode gesetzt. Das bedeutet, das gefundene Objekt vom Typ OBase, was wir im vorherigen Abschnitt im Attribut OElement gesetzt haben, kann von dieser Methode handleJobDetails weiter benutzt werden. Abhängig von den unterschiedlichen Jobtypen werden die in dem Ablaufplan vorkommenden Jobs individuell behandelt werden (siehe bitte Listing 6.6).

```
public void handleJobDetails(JobDetails jobData) {
 JobDetails we = jobData;
  ....
  switch (we.getType()) {
      case TIMER:
       ...
      case RESUME:
       ...
      case ITERATE:
       BpelRuntimContextImpl processInstance5=
             createRuntimeContextForIterate(procInstance,null);
       ...
 }
```

Listing 6.6 Ausschnitt aus der Klasse BpelProcess.java

Eine neue Instanz der Klasse *BPELRuntimeContextImpl* wird zuerst erzeugt; dies ist eine Implementierung des Interfaces *BPELRuntimeContext*. Eine neue Instanz der Klasse *JacobVPU* und eine Instanz der Klasse *ExecutionQueueImpl* werden direkt im Konstruktor dieser Klasse angelegt. Alle ausführbaren Aktivitäten werden einer Warteschlange, die *ExecutionQueue* genannt wird, hinzugefügt. Jedes Objekt, das sich in dieser Warteschlange beziehungsweise *ExecutionQueue* befindet, repräsentiert eine Aktivität. Die Ausführung einer Aktivität bedeutet, dass die run-Methode dieser Aktivität in Form von *ACTIVITY* ausgeführt wird. Wenn diese Aktivität ausgeführt wird, wird das zu dieser Aktivität gehörende Objekt aus der *ExecutionQueue* entfernt und dann ausgeführt.

```
public SEQUENCE (ActivityInfo self, ScopeFrame scopeFrame, LinkFrame
linkFrame, QName processname, Long pid){
     this(self, scopeFrame, linkFrame, ((OSequence) self.o).sequence,
          CompensationHandler.emptySet(), true, processname, pid);
    }
public SEQUENCE(ActivityInfo self, ScopeFrame scopeFrame, LinkFrame
linkFrame,List<OActivity>      remaining,      Set<CompensationHandler>
compensations, Boolean firstT, QName processname, Long pid) {
     super(self, scopeFrame, linkFrame, processname, pid);
     _remaining = remaining;
     _compensations = compensations;
     _firstTime = firstT;
     process_name = processname;
     process_ID = pid;
    }
```

Listing 6.7 Der Ausschnitt der Klasse SEQUNECE

Weil wir zur Zeit nur die Aktivitäten innerhalb von einer <sequence>-Aktivität durch die Iterate-Operation nochmal ausführen lassen und die Instanz von der Klasse *SEQUENCE* noch nicht fertig ausgeführt ist, kann diese Instanz über das Attribut *_index* der *ExecutionQueue* gefunden werden. Eine neue Instanz von SEQUENCE soll für die Operation *Iterate* erzeugt werden. In Listing 6.7 steht *_remaining* für die Menge der Kindaktivitäten, die sich innerhalb dieser <sequence>-Aktivität befinden. Die erneut auszuführende Aktivität gehört auch zu den Kindaktivitäten. In *_index* gibt es mehrere Objekte für OSequence. Eines davon ist das Objekt für OSequence, was am Anfang der ExecutionQueue hinzugefügt wurde und in dem noch keine Kindaktivitäten ausgeführt wurden. Das andere ist auch das Objekt für OSequence, aber dieses Objekt enthält weiniger Kindaktivitäten, weil einige davon schon fertig ausgeführt und aus der Menge der Kindaktivitäten entfernt wurden. Diese zwei Objekte von OSequence können basierend auf ihrer Länge unterschieden werden. Jetzt müssen die erneut auszuführende Aktivität und auch ihre Nachfolgeaktivitäten wieder der Menge der Kinderaktivitäten hinzugeführt werden. Die anderen Werte von Parametern, die von dem Konstruktor der Klasse SEQUENCE benötig werden, können auch bekannt sein. Jetzt kann die neue Instanz der Klasse SEQUENCE erzeugt werden (siehe Listing 6.8).

```
sq = (SEQUENCE)it1.next();
scopeframe = sq.get_scopeFrame();
linkframe = sq.get_linkFrame();
List<OActivity>     activities = ((OSequence)sq._self.o).sequence;
int a = activities.indexOf(element);
List<OActivity> as = new ArrayList<OActivity>();
//to find all the activities after the element, which should be
iterated.and i will put
//all these activities in a new SEQUENCE()as the _remaining
for (int i = a; i< activities.size();i++){
     as.add(activities.get(i));
 }
SEQUENCE sequence;
QName process_name;
Long processID;
process_name = we.getProcessId();
processID = we.getInstanceId();

sequence = new SEQUENCE(sq._self, scopeframe, linkframe, as,
CompensationHandler.emptySet(), false, process_name, processID);
```

Listing 6.8 Erzeugen einer neuen Instanz der Klasse SEQUENCE

SEQUENCE und ACTIVITY

Jetzt soll die Klasse SEQUENCE näher betrachtet werden. Jede Klasse von Aktivitäten in der ODE-Runtime erweitert die Klasse ACTIVITY. ACTIVITY, wie der Abschnitt 3.5 vorgestellt hat, erweitert die Klasse BPELJacobRunnable, die ebenfalls auch die Klasse JacobRunnable erweitert. Wir haben nur eine neue Instanz von BPELRuntimeContextImpl erzeugt, aber diese BPEL-Instanz wird noch nicht gestartet. Das bedeutet, dass die Instanz von JacobVPU auch nicht läuft. Deshalb werden einige Änderungen in der Klasse ACTIVITY gemacht. Listing 6.9 zeigt den originalen Konstruktor dieser Klasse und Listing 6.10 stellt den neuen geänderten Konstruktor dieser Klasse dar. Wenn in Listing 6.9 die Methode getBpelRuntimeCntext() aufgerufen wird, muss eine aktive Instanz von *JacobThread*

gefunden werden. Aber zum aktuellen Zeitpunkt läuft die Instanz von JacobVPU nicht und deshalb kann keine aktive Instanz von *JacobThread* gefunden werden.

```
public ACTIVITY(ActivityInfo self, ScopeFrame scopeFrame,
     LinkFrame linkFrame) {

 process_name = getBpelRuntimeContext().getBpelProcess().getPID();
 process_ID = getBpelRuntimeContext().getPid();
 assert self != null;
 assert scopeFrame != null;
 assert linkFrame != null;
  _self = self;
 _scopeFrame = scopeFrame;
 _linkFrame = linkFrame;
 _terminatedActivity = false;
 getFrames();
 }
```

Listing 6.9 Der originale Konstruktor der Klasse ACTIVITY

In dem neuen Konstruktor der Klasse ACTIVITY sind Prozessname und ProzessinstanzId auf die bekannten Parametern gesetzt.

```
public ACTIVITY(ActivityInfo self, ScopeFrame scopeFrame,
     LinkFrame linkFrame, QName processname, Long pid) {
 assert self != null;
 assert scopeFrame != null;
 assert linkFrame != null;

 process_name = processname;
 process_ID= pid;
 _self = self;
 _scopeFrame = scopeFrame;
 _linkFrame = linkFrame;
 _terminatedActivity = false;
 getFrames();
}
```

Listing 6.10 Der geänderte Konstruktor der Klasse ACTIVITY

BpelRuntimeContextImpl

Des Weiteren wird der Konstruktor der Klasse *SEQUENCE* auch geändert. Jetzt wird die neue Instanz der Klasse *SEQUENCE* schon erfolgreich erzeugt. Jetzt kann die neu kreierte Instanz von *BpelRuntimeContext* endlich ausgeführt werden. Weil die neue <sequence>-Aktivität in den *JacobVPU* angefügt wird, muss auch eine neue Methode zu der Ausführung von *BpelRuntimeContext* speziell für die *Iterate*-Operation entwickelt werden. Hier wird sie *executeForIterateAndReexecute()* genannt. Diese Methode braucht zwei Parametern. Einer davon ist die auszuführende Aktivität in Form von *OActivity*. Der andere ist die neu erzeugte Instanz der Klasse *SEQUENCE*.

```
public void executeForIterateAndReexecute(OActivity element,
  SEQUENCE sq){
 IncomingMessageHandlerincMess=IncomingMessageHandler.getInstance();
  boolean canReduce = true;
  ExecutionQueueImpl eql;
  Set<Continuation> reactions = new HashSet<Continuation>();
   ...
}
```

Listing 6.11 Die für Iterate neu erstellte Methode *executeForIterateAndReexecute()*

Alle in der ODE-Runtime auszuführenden Aktivitäten werden als ein Objekt der Klasse *Continuation*, die ebenfalls auch die Klasse *ExecutionQueueObject* erweitert, in *_reactions* gespeichert. Dabei ist *_reactions* ein Attribut von ExecutionQueue. Wichtig ist hier, die zu wiederholende Aktivität aus der ExecutionQueue zu entfernen, das heißt, sie aus *_reactions* zu entfernen. Nach der Entfernung kann die neue Instanz der Klasse SEQUENCE endlich der JacobVPU hinzugefügt werden. Weil die Instanzen vom *OActivity* in ODE-Runtime auf der unterschiedliche Weise repräsentiert werden, muss eine Methode ergedacht werden, um das richtige passende Objekt der Klasse *Continuation* für die zu wiederholende Aktivität in *_reactions* zu finden. Beispielsweise wird das Objekt der Klasse *OAssign* als *{OAssign : Assign2, joinCondition=null}*, das Objekt der Klasse *OWait* als *{OWait#41-wait}* und das Objekt der Klasse *OThrow* als *{OThrow#43-Throw}* dargestellt. Das folgende Listing zeigt, wie die Aktivität aus der ExecutionQueue zuerst gefunden, daraus entfernt und die als Parameter mitgebrachte Instanz der SEQUENCE wieder im JacobVPU eingesetzt wird.

```
 eql=(ExecutionQueueImpl) _vpu._executionQueue;
Continuation continuation = null;
Iterator <Continuation> it = reactions.iterator();
String elementname = element.toString();
//find the key name of this element. By different OActivity the
 //names are also
//different, for example {OAssign : Assign2, joinCondition=null}
 // and {OWait#41-wait}and by {OThrow#43-Throw }
int inof = elementname.indexOf("#");
int inof2 = elementname.indexOf((char)32);
String name;
String activityname;
if(inof == -1){
      name = elementname.substring(2,inof2);
      activityname = name.toUpperCase();
}
else{
name = elementname.substring(2, inof);
activityname = name.toUpperCase();
}
//to delete the activity from the _reactions, which should be
// iterated.
while(it.hasNext()){
Continuation     tmp1 = (Continuation)it.next();
if(!tmp1.getClosure().getRunner()){
      if(tmp1.getClosure().toString().contains(activityname)){
            continuation = tmp1;
            break;
      }
}
}
reactions.remove(continuation);
//put the new "SEQUENCE" into the JacobVPU
 _vpu.inject(sq);
```

Listing 6.12 Der Ausschnitt aus der Methode *executeForIterateAndReexecute()*

Schließlich wird der JacobVPU ausgeführt und die Aktivität kann erfolgreich noch einmal ausgeführt werden.

6.3.3 Re-execute

Im Kapitel 5 wurden das Konzept für die Operation Re-execute und auch die Unterschiede zwischen den beiden Operationen Iterate und Re-execute schon vorgestellt. In diesem Abschnitt werden nur die zusätzlichen Implementierungsaufgaben von Re-execute im Vergleich zu Iterate beshrieben. Im folgenden Sequenzdiagramm für Re-execute werden die Unterschiede in rot dargestellt(siehe die Abbildung 6.5).

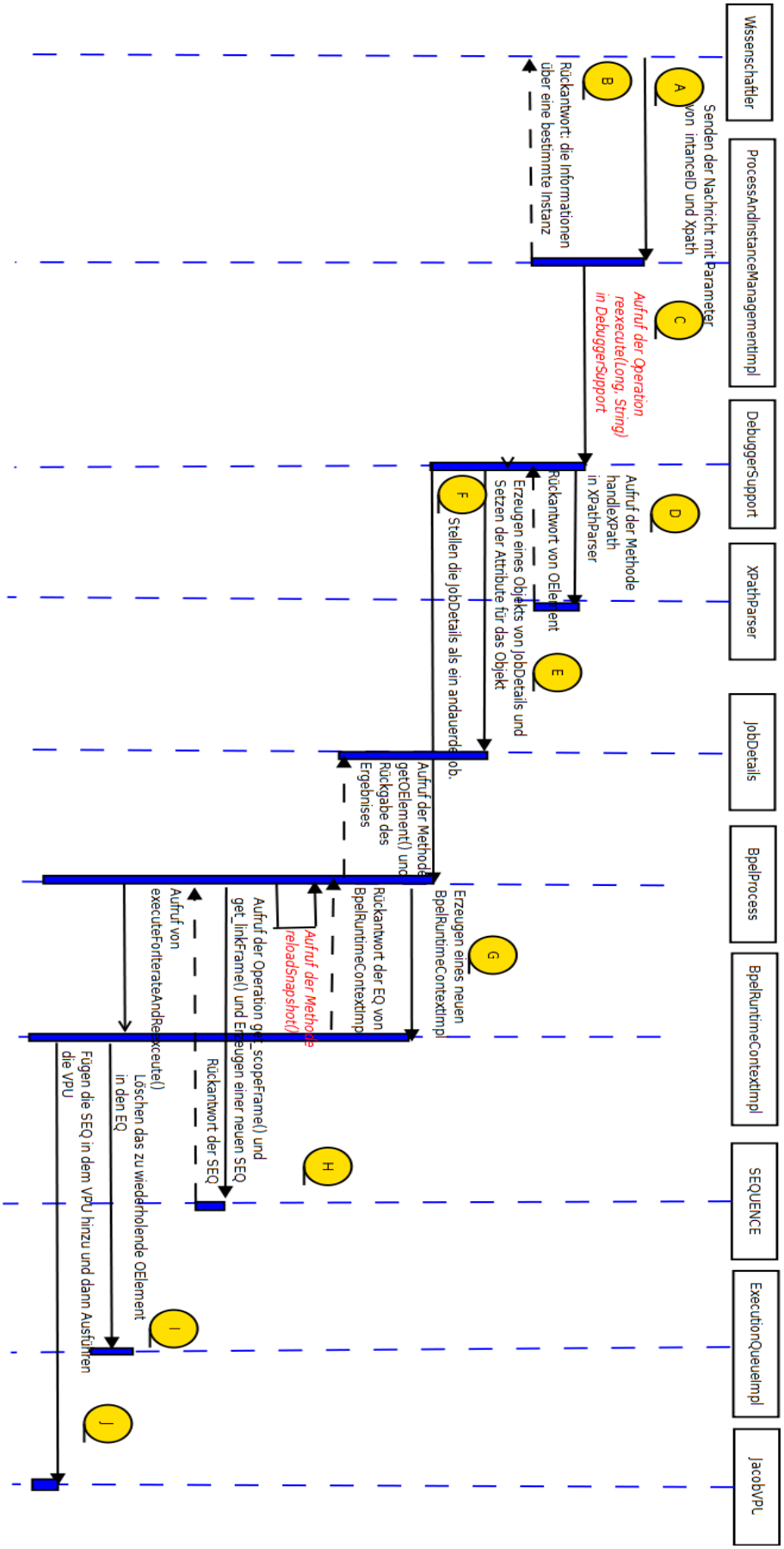

Abbildung 6.5 Das Sequenzdiagramm für die Operation Re-execute

Für Re-execute sind zwei neue Methoden nämlich *reexecute()* in der Klasse *DebuggerSupport.java* und *reloadSnapshot()* in der Klasse *BpelProcess.java*, hinzugefügt worden. Der einzige Unterschied zwischen *reexecute()* und *iterate()* in DebuggerSupport.java liegt in den Namen der Methode. Die andere neue Methode *reloadSnapshot()* wird später vorgestellt. Außer den in Abbildung 6.6 repräsentierten Unterschieden beziehungsweise den zwei neuen Methoden gibt es noch zusätzliche Aufgaben, um die Operation Re-execute zu realisieren. Der Abschnitt 3.3 hat schon kurz vorgestellt, dass die Engine-Runtime der Apache ODE die Data Access Objects (DAO) als die Schnittstelle für die Persistenzebene enthält. Die Persistenzebene ist normalerweise eine relationale Datenbank. Die durch DAOs dauerhaft gespeicherten Informationen enthalten die Daten über die Prozessinstanzen, Werte von den Variablen, usw. Die Apache ODE bietet zwei verschiedene Implementierungen von DAOs an. Eine Implementierung benutzt Hibernate, ein Open-Source-Persistenz- und ORM (Object-Relational Mapping)-Framework für Java [WHIB]. Die andere Implementierung basiert auf Apache OpenJPA, welches die Java Persistence API(JPA) implementiert, um die Java Objects dauerhaft abzuspeichern [Son08]. In dieser Arbeit wird nur die Implementierung der Apache OpenJPA betrachtet.

DAOs

In dem Paket *bpel-dao* bietet die APIs im Sourcecode die Ebene der DAOs an. Diese APIs sind beispielsweise *ProcessInstanceDAO*, *ScopeDAO*, *PartnerlinkDAO* und auch *XmlDataDAO*. *ScopeDAO* repräsentiert die Instanz von einem BPEL Scope. *PartnerlinkDAO* stellt die Referenzen der Endpunkte (EPR) von einer speziellen Partnerlinkrole dar. *XmlDataDAO* repräsentiert die XML-Daten, die zur Speicherung von BPEL-Variablen dienen. In *ProcessInstanceDAO* kann ein *ScopeDAO* erzeugt werden und in *ScopeDAO* kann ein *PartnerlinkDAO* erstellt werden. In dem Paket bpel-jpa gibt es die Implementierung von OpenJPA für die DAOs. Die folgende Abbildung 6.6 zeigt ein Klassendiagramm von DAOs und auch ihren jeweiligen Implementierungsklassen.

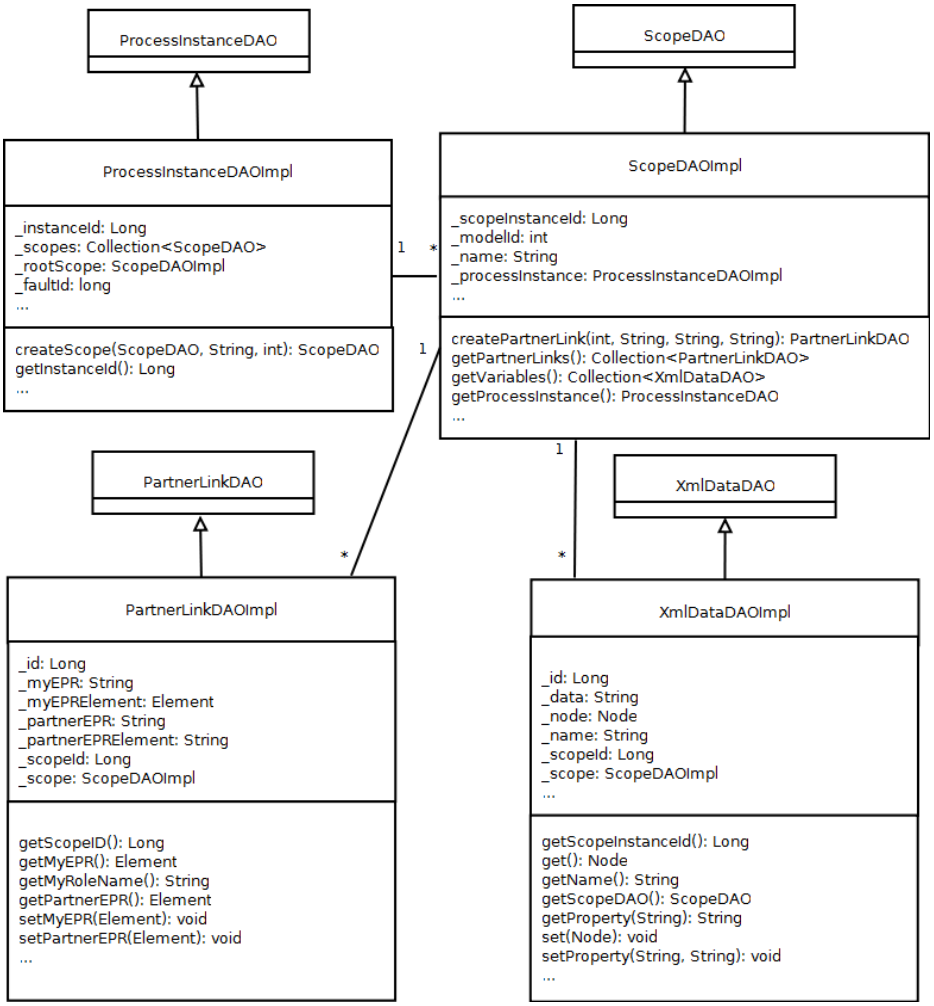

Abbildung 6.6 Klassendiagramm von vier Klassen in DAO

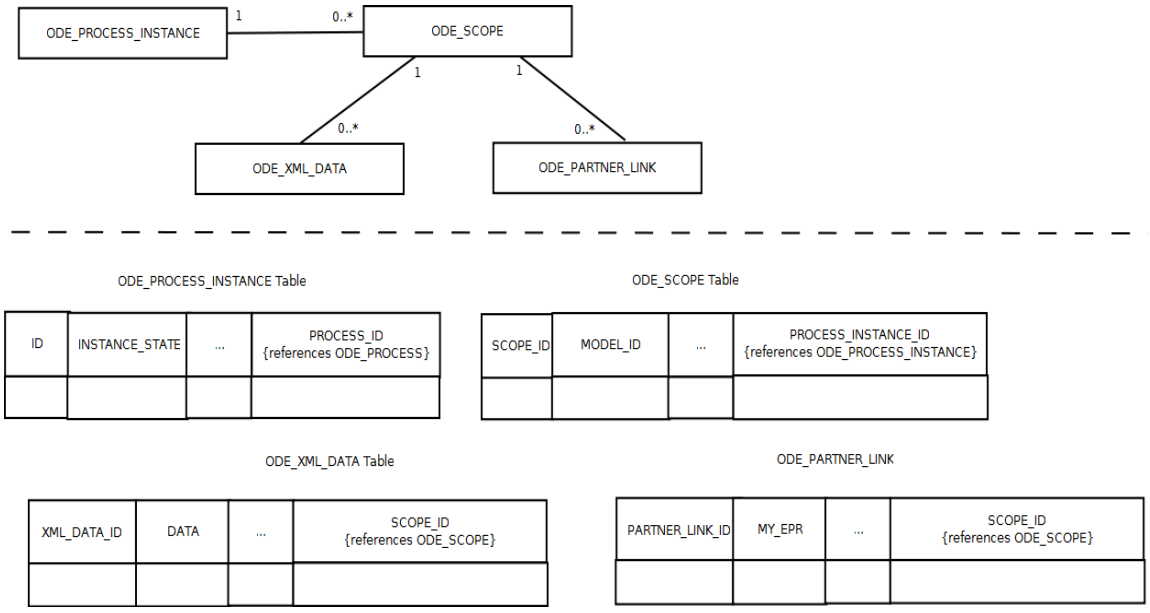

Abbildung 6.7 Datenmodell von vier Klassen in DAO

Die Abbildung 6.7 stellt das Datenmodell von den Tabellen dar, die durch die vier Implementierungsklassen der DAOs erzeugt werden. Eine Prozessinstanz kann mehrere Scopeinstanzen haben und eine Scopeinstanz kann auch mehrere Variablen und Partnerlinks besitzen. Das heißt, dass die Beziehung zwischen Prozessinstanz und Scopeinstanz *one-to-many* ist. Gleichermaßen ist die Beziehung zwischen Scopeinstanz und Variablen und die Beziehung zwischen Scopeinstanz und Partnerlink auch *one-to-many*.

Wie der Abschnitt 5.3 über das Konzept von Re-execute vorgestellt hat, müssen auch einige DAOs für die Snapshots erzeugt werden, um die Werte von Partnerlinks und Variablen für jede Aktivität abzuspeichern. Wenn eine Aktivität durch die Operation Re-execute nochmals ausgeführt wird, sollen die schon gespeicherten Daten dieser Aktivität aus den Snapshots wieder geladen werden. Das bedeutet, dass die Daten von dieser zu wiederholenden Aktivität in den DAOs von PartnerLinkDAO, XmlDataDAO, und ScopeDAO durch die Daten dieser Aktivität aus Snapshots überschrieben werden müssen. Die Snapshots können in gewisser Hinsicht als Zwischenspeicher betrachtet werden. Die ODE bietet noch keine Funktionalität, Snapshots zu speichern, deshalb muss sie in dieser Arbeit entsprechend nachgerüstet werden. Die folgende Abbildung 6.8 zeigt das Klassendiagramm von DAOs für Snapshots.

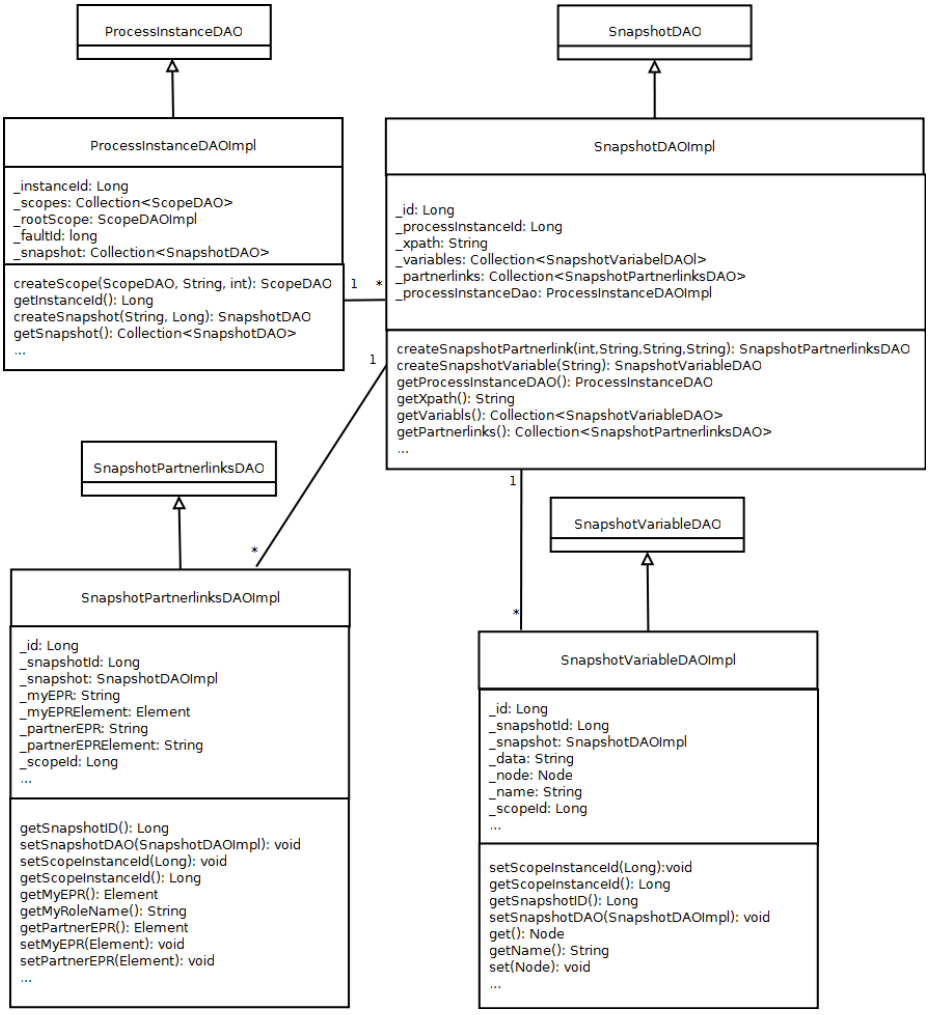

Abbildung 6.8 Das Klassendiagramm für Snapshots

Wie das Klassendiagramm von *ProcessInstanceDAOImpl* gezeigt hat, werden zwei neue Methoden, nämlich *createSnapshot* und *getSnapshotDAO* und auch das neue Attribut, nämlich *_snapshot* eingebaut (siehe Listing 6.13). Mit den neuen Methoden kann ein SnapshotDAO für jede Aktivität erzeugt werden, weil es ein ProcessinstanceDAO für jede Aktivität gibt. Deshalb können auch die SnapshotPartnerlinksDAO und SnapshotVariableDAO erstellt werden. Wie das SnapshotDAO genau erzeugt werden soll, wird im folgenden Abschnitt erklärt. Zu jedem Snapshot wird auch der Xpath dieser Aktivität abgespeichert.

```
Public SnapshotDAO createSnapshot(String xpath, Long
      processinstanceId){
   SnapshotDAOImpl snap = new SnapshotDAOImpl(processinstanceId,
   xpath, this);
  _snapshot.add(snap);
  // Must persist the snapshotDAO to generate a snapshot ID
  getEM().persist(snap);
  return snap;
  }
//to read the SnapshotDAO collection from the ProcessInstanceDAO
public Collection <SnapshotDAO> getSnapshotDAO(){
    return _snapshot;
  }
```

Listing 6.13 Die neuen Methoden in ProcessInstanceDAOImpl.java

Die folgende Abbildung 6.9 stellt das Datenmodell von SnapshotDAOs dar.

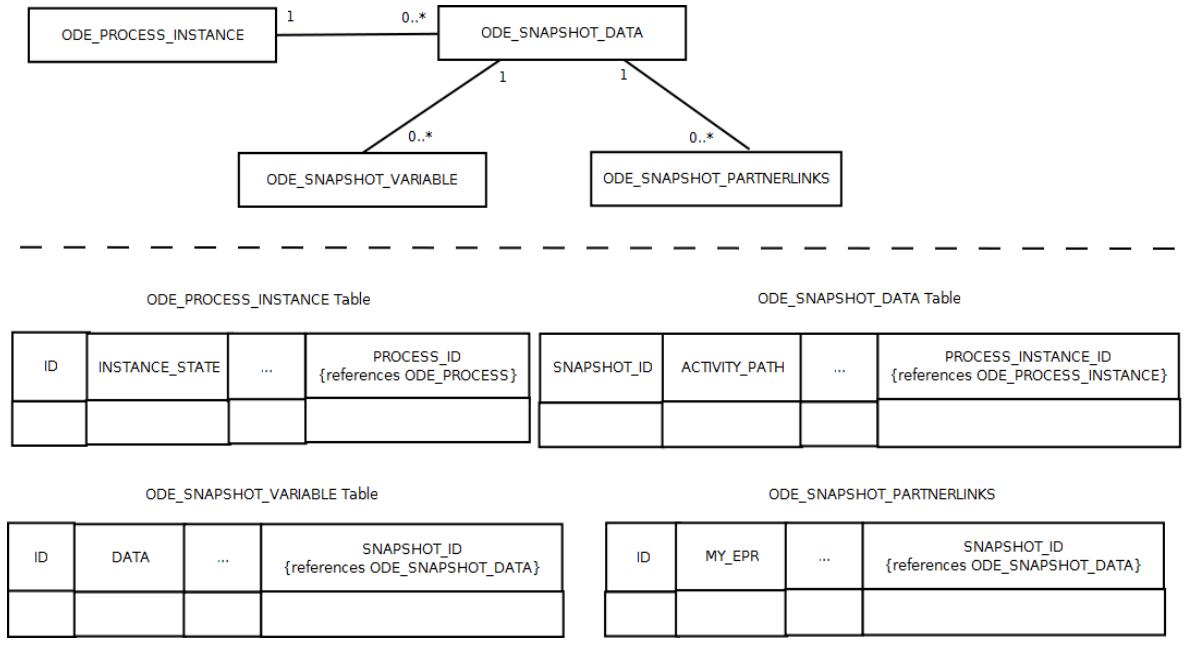

Abbildung 6.9 Datenmodell für neue SnapshotDAOs

Eine Prozessinstanz hat mehrere *SnapshotDAO* und jeder *SnapshotDAO* kann auch mehrere *SnapshotPartnerlinksDAO* und *SnapshotVariableDAO* enthalten. Aber durch den FK (Fremd Key) kann die Beziehung von *one-to-many* eindeutig dargestellt werden.

ACTIVITY

Um die Snapshots für jede Aktivität abspeichern zu können, muss eine Methode in *ACTIVITY.java* geschrieben werden. Weil alle in der ODE-Runtime laufenden Aktivitäten diese Klasse erweitern, können die Snapshots für jede Aktivität dann durch den Aufruf dieser Methode erzeugt werden. Listing 6.14 zeigt die neue Methode *storeSnapshot()*.

```
public SnapshotDAO storeSnapshot(){
      String xpath;
      ProcessInstanceDAO pi;
      xpath = this._self.o.getXpath();
      pi = getBpelRuntimeContext().getProcessInstanceDao();
      //create a snapshot for this processInstanceDAO now!
      SnapshotDAO snapshotdao;
      snapshotdao = pi.createSnapshot(xpath, pi.getInstanceId());
      //create a collection for the SnapshotPartnerlinks
      Collection<PartnerLinkDAO> partnerlinks ;
        //create a collection for the SnapshotVariables
         Collection<XmlDataDAO> variables;
         SnapshotPartnerlinksDAO sp ;
         SnapshotVariableDAO var;
       for (ScopeDAO scope : pi.getScopes()){
          partnerlinks = scope.getPartnerLinks();
          Long scopeinstanceID;
          scopeinstanceID =scope.getScopeInstanceId();
          variables = scope.getVariables();
//put the content of the SnapshotPartnerlinks and SnapshotVariables
//into the SnapshotDAO
             for (PartnerLinkDAO partnerlink : partnerlinks){
                sp=
snapshotdao.createSnapshotPartnerlink(partnerlink.getPartnerLinkModelId
(),        partnerlink.getPartnerLinkName(),partnerlink.getMyRoleName(),
partnerlink.getPartnerRoleName());
            sp.setMyEPR(partnerlink.getMyEPR2());
            ..........
              }
              for(XmlDataDAO variable : variables){
               var=

snapshotdao.createSnapshotVariable(variable.getName());
                 var.set(variable.get());
            ..........
              }
        }
        return snapshotdao;
       }
```

Listing 6.14 Die Methode *storeSnapshot()* in ACTIVITY.java

Zuerst wird eine Instanz von *ProcessInstanceDAO* aus dem BpelRuntimeContext ermittelt und dann wird eine Instanz von *SnapshotDAO* durch den Aufruf der Methode *createSnapshot()* erzeugt. Dann werden die Instanzen von SnapshotPartnerlinksDAO und SnapshotVariableDAO erstellt. Alle Partnerlinks und Variablen inklusive auch ihren Attributen, die durch die Methoden *getPartnerLinks()* und *getVariables()* in den PartnerLinkDAO und XmlDataDAO gespeichert sind, sollen in SnapshotPartnerlinksDAO und SnapshotVariableDAO noch einmal gespeichert werden.

Weil die Methode storeSnapshot() vor der Ausführung jeder Aktivität aufgerufen werden muss, muss diese Methode in der run()-Methode vor dem Event *Activity_Ready* in allen Aktivitätenklassen in der ODE-Runtime eingefügt werden.

BpelProcess

Jetzt sind alle notwendigen Daten von Partnerlinks und Variablen für die Operation Re-execute in SnapshotDAO, SnapshotPartnerlinksDAO und SnapshotVariableDAO gespeichert. In der Methode handleJobDetails wird die Operation *Reexecute* auch wie *Iterate* behandelt. Der einzige Unterschied ist, dass eine zusätzliche Methode, nämlich *reloadSnapshot()* aufgerufen werden muss, bevor eine neue Instanz von SEQUENCE erzeugt wird. Diese Methode dient dazu, dass die Daten über Partnerlinks und Variablen aus SnapshotPartnerlinksDAOs und SnapshotVariablesDAO wieder in den PartnerlinksDAO und XmlDataDAO geladen werden können. Das heißt, dass die aktuellen Werte von Partnerlinks und Variablen von dieser zu wiederholenden Aktivität, die schon ausgeführt wurde, durch die Werte, die vor der letzten Ausführung dieser Aktivität von storeSnapshot() abgespeichert wurden, überschrieben werden. Die Methode *reloadSnapshot()* wird auch in *BpelProcess.java* eingefügt (siehe Listing 6.15).

```
public void reloadSnapshot(Long processInstanceId, String xpath){
     ProcessInstanceDAO pi_dao;
     Collection<SnapshotPartnerlinksDAO> spl;
     Collection<SnapshotVariableDAO> svar;
     //get the current ProcessInstanceDAO with the given
    //ProcessInstanceID
     pi_dao = this.getProcessDAO().getInstance(processInstanceId);
     // get the data of the SnapshotDAO from the ProcessInstanceDAO
     Collection<SnapshotDAO> s = pi_dao.getSnapshotDAO();
           Collection<ScopeDAO> scopes = pi_dao.getScopes();

           Collection<PartnerLinkDAO> partnerlinks;
           Collection<XmlDataDAO> variables;

           for(SnapshotDAO snapshot : s){
     //only get the snapshot, which hat the given xpath!

           if(snapshot.getXpath().equals(xpath)){
                      spl = snapshot.getPartnerLinks();
                      svar = snapshot.getVariables();

            for(ScopeDAO scope : scopes){
                           ..........
                for(SnapshotPartnerlinksDAO snapshotpl : spl){
                    Long scope_id_sp =snapshotpl.getScopeInstanceId();
                       for (PartnerLinkDAO ps : partnerlinks){
                           ..........
                                 }
                  }
                for (SnapshotVariableDAO snapshotvar : svar){
                  Long scope_id_var = snapshotvar.getScopeInstanceId();
                           for (XmlDataDAO variable :variables ){
                           ..........
```

Listing 6.15 Die Methode *reloadSnapshot()* in BpelProcess.java

Listing 6.14 zeigt, dass die Methode *reloadSnapshot()* zwei Parameter braucht. Sie sind ProzessinstanceId und xpath, um die bestimmten *ProcessInstanceDAO*s und die bestimmten *SnapshotDAOs* für die bestimmte Aktivität zu finden. Durch xpath können nur die Daten in *SnapshotDAOs* für die Aktivität, bei der die erneute Ausführung starten wird, gefunden werden. Anhand von *ProcessInstanceDAO* kann das dazu gehörende *ScopeDAO* produziert werden und mittels *ScopeDAO* können die Daten von *PartnerlinkDAO* und *XmlDataDAO* ermittelt werden. Anschließend werden die Werte von Partnerlinks und Variablen durch die Werte aus *SnapshotPartnerlinksDAO* und *SnapshotVariableDAO* überschrieben. Nach dem Aufruf von der Methode *reloadSnapshot()* wird dann eine neue Instanz von SEQUENCE erzeugt und die Prozessinstanz der Methode *executeForIterateAndReexecute()* wird ausgeführt. In dieser Methode *executeForIterateAndReexecute()* wird der JacobVPU ausgeführt und die Aktivität kann erfolgreich noch einmal durch Re-execute ausgeführt werden.

7. Zusammenfassung und Ausblick

Im Rahmen dieser Diplomarbeit wurden die Möglichkeiten untersucht, mit denen Workflow-Maschinen an die Anforderungen von Wissenschaftlern angepasst werden können. Solche Möglichkeiten können den Wissenschaftlern bei Entwurf, Ausführung, Überwachung und Analyse von Experimenten helfen. Dazu wurde ein Konzept entwickelt, um Aktivitäten in Workflows von den Wissenschaftlern erneut ausführen zu können. Für die Wiederholung von Aktivitäten gibt es 2 Szenarien. Erstens ist dies die einfache Iteration wie in einer Schleife. Zweitens ist es dies das erneute Ausführen der Aktivitäten, als hätte es die Ausführung davor nicht gegeben. Das Konzept besteht deshalb aus den zwei Operationen Iterate und Re-execute, die diese beiden Szenarien abdecken. Das Konzept zu den zwei Operationen wurde für die Sprache WS-BPEL in verschiedenen Fällen erstellt.

Auf Basis der oben beschriebenen Erkenntnisse wurde die Apache ODE prototypisch um die Operationen Iterate und Re-execute erweitert. Die Operation Iterate ermöglicht es den Wissenschaftlern, die Aktivitäten innerhalb einer <sequence>-Aktivität an beliebigen Stellen erneut auszuführen. Die Operation Re-execute ermöglicht es den Wissenschaftlern, die Aktivitäten innerhalb von einer <sequence>-Aktivität an beliebigen Stelle mit den Werten von Partnerlinks und Variablen, die bei ihrer letzten Ausführung gültig waren, noch einmal auszuführen.

Das im Rahmen dieser Diplomarbeit entwickelte Konzept von den zwei Operationen kann als Basis zu der Entwicklung einer Workflow-Maschine für die Sprache WS-BPEL, die ähnliche Funktionalität anbietet, dienen. Der in dieser Arbeit entwickelte Prototyp kann zu der Weiterentwicklung der Apache ODE benutzt werden. Weil in dieser Arbeit nur die Sequence-Aktivität zur Implementierung benutzt wurde, sind zukünftige Arbeiten möglich, die Aktivitäten innerhalb der Flow-Aktivität, While-, forEach- und auch repeatUntil-Aktivitäten durch Iterate und Re-execute erneut ausführen. Für das Re-execute soll in der Zukuft umgesetzt werden, die zu wiederholenden Aktivitäten vor der erneuten Ausführung erst zu kompensieren, wie im Kapitel 5 beschrieben wurde.

Literaturverzeichnis

[BioF]E-BioFlow.
URLhttp://janus.cs.utwente.nl:8000/twiki/bin/view/BioRange/BioRangeSoftware#e_BioFlow.

[Bur05]H. Burkhart: Webtechnologien. 2005.
URL http://fgb.informatik.unibas.ch/lectures/archive/SS2005/CS211%20webtech/L11_f.pdf.

[EKU+10]H. Eberle; O. Kopp; T. Unger; F. Leymann: Retry Scopes to Enable Robust Workflow Execution in Pervasive Environments. In: Proceedings of the 2nd Workshop on Monitoring, Adaptation and Beyond (MONA+), 2009.

[Dos05]W. Dostal: Service-orientierte Architekturen mit Web Services.
Spektrum Akademischer Verlag, 2005.

[HW05]T. Holzherr, M. Wodischek: Java RMI und SOAP. 2005. URL https://www.ba-horb.de/fileadmin/media/it/studienprojekte/projektarbeiten/se_seminar_it2003/se_seminar_it2003_02.pdf.

[IBM]IBM. URL
http://publib.boulder.ibm.com/infocenter/cicsts/v4r1/index.jsp?topic=/com.ibm.cics.ts.webservices.doc/concepts/soap/dfhws_messagepath.html.

[Int06]S. Intas: Konzept für die Einbindung von Webservices in Ereignisgesteuerte Prozessketten, 2006.
URL http://www.se.uni-hannover.de/documents/studthesis/MSc/Sebastian_Intas-Konzept_fuer_die_Einbindung_von_Webservices_in%20Ereignisgesteuerte_Prozessketten.pdf.

[LR00]F. Leymann, D. Roller: Production Workflow - Concepts and Techniques. Prentice Hall, 2000.

[Man07]K. Manhart: Web Services implementieren mit WSDL, 2007.
URL http://www.tecchannel.de/webtechnik/soa/464653/web_services_implementieren_mit_wsdl/index2.html.

[Mas07]D. Masak: SOA? Serviceorientierung in Business und Software. Springer Verlag, 2007. URL
http://books.google.de/books?id=0JNivfep8DMC&printsec=frontcover&dq=SOA+Tempel#v=onepage&q&f=true.

[Mel10]I. Melzer et al.: Service-orientierte Architekturen mit Web Services. Konzepte – Standards – Praxis. 2010 4nd Edition, 2010. URL
http://books.google.de/books?id=e3qnVPngoUoC&pg=PA13&dq=SOA+Tempel&hl=de&ei=FWdcTYqsApDz4gbL1o2bDA&sa=X&oi=book_result&ct=result&resnum=2&ved=0CEEQ6AEwAQ#v=onepage&q=SOA%20Tempel&f=false.

[MOB]MOBY. URL http://biomoby.open-bio.org/index.php/what-is-moby/.

[OASIS]OASIS. URL http://docs.oasis-open.org/wsbpel/2.0/CS01/wsbpel-v2.0-CS01.html.

[ODE]Apache ODE. URL http://ode.apache.org/.

[ODE1] Apache ODE. URL http://ode.apache.org/ws-bpel-20-specification-compliance.html

[ODE2] Apache ODE. URL http://ode.apache.org/architectural-overview.html

[RNS96] Research Note SPA-401-068, 12 April 1996, "'Service Oriented' Architectures, Part 1" und SSA Research Note SPA-401-069, 12 April, 1996.

[Sal07]A. Salnikow: Ermittlung von Testabdeckungsmetriken in BPEL-Kompositionen, 2007. URL http://www.se.uni-hannover.de/documents/studthesis/MSc/Alex_Salnikow-Ermittlung_von_Testabdeckungsmetriken_in_BPEL.pdf.

[Sch07]B. Schurr: Analyse von XPath-Ausdrücken in BPEL Prozessbeschreibungen. DIP-2687, 2007.

[Sch11]T. Schliemann: Unterstützung des "Model-as-you-go"-Ansatzes durch Modell-Versionierung und Instanzmigration., DIP-3121, 2011.

[SK10]M.Sonntag; D. Karastoyanova: Next Generation Interactive Scientific Experimenting Based On The Workflow Technology. In: Alhajj, R.S. (Hrsg); Leung, V.C.M. (Hrsg); Saif, M. (Hrsg); Thring, R. (Hrsg): Proceedings of the 21st IASTED International Conference on Modelling and Simulation (MS 2010), 2010.

[SOA06]Reference Model for Service Oriented Architecture 1.0, Commit Specification1, 2 August, 2006.

[Son08]M. Sonntag: Conceptual Design and Implementation of a BPEL^{light} Workflow Engine With Support for Message Exchange Patterns., DIP-2822. 2008.

[Ste08]T. Steinmetz: Ein Event-Modell für WS-BPEL 2.0 und dessen Realisierung in Apache ODE ., DIP-2729, 2008.

[Tap04]Carlos C. Tapang: Web Services Description Language (WSDL) in Überblick, 2004. URL http://www.microsoft.com/germany/msdn/library/xmlwebservices/WebServices DescriptionLanguageWSDLImUeberblick.mspx?mfr=true.

[Ulm]T. Uml: Einführung in SOAP,URL http://www.devtrain.de/news.aspx?artnr=422.

[VAD+04]W. Van der Aalst, L. Aldred, M, Dumas, A. Ter Hofstede: Design and implementation of the YAWL system. In G. Goos, et al, 16th International Conference on Advanced Information Systems Engineering. Springer Verlag. 2004.

[Vik08]B. Vikum: Serviceorientierte Architekturen - SOA 2008. URL http://swt.cs.tu-berlin.de/lehre/sepr/ss08/referate/SOA_Ausarbeitung.pdf.

[W301]W3. URL http://www.w3.org/TR/ws-arch/.

[W3S]W3schools. URL http://www.w3schools.com/soap/soap_header.asp.

[WCL]Wikipedia. URL http://de.wikipedia.org/wiki/Closure.

[WCL+05]S. Weerawarana; F. Curbera; F. Leymann; T. Storey; D. F. Ferguson: Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More. Prentice Hall PTR, 2005.

[WHIB]Wikipedia. URL http://de.wikipedia.org/wiki/Hibernate_(Framework).

[WODE]Wikipedia. URL http://en.wikipedia.org/wiki/Apache_ODE.

[WOV09]I. Wssink, M. Ooms, P. Van der Vet: Designing workflows on the fly using e-BioFlow, 2009. URL http://www.springerlink.com/content/1660n17463872507/.

[WRV+08]I. Wassink, H. Rauwerda, P. Van der Vet, T. Breit, A. Nijholt: BioFlow: Different Perspectives on Scientific Workflows, 2008.
URL http://www.springerlink.com/content/j38vk6222j140k72/.

[WSOA]Wikipedia.
URL http://de.wikipedia.org/wiki/Dienstorientierte_Architektur#cite_ref-0.

[WSOAP]Wikipedia.
URL http://de.wikipedia.org/wiki/SOAP#Aufbau_von_SOAP-Nachrichten.

[WXP]Wikipedia. URL http://de.wikipedia.org/wiki/XPath.

[Yu10] J.Yu: Exploring ODE part 2, URL http://jeff.familyyu.net/2010/01/exploring-apache-ode-source-code-part.html.

Alle URLs wurden zuletzt am 08.03.2011 geprüft.

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

(Bo Ning)