Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D – 70569 Stuttgart

Diplomarbeit Nr. 3121

Unterstützung des „Model-as-you-go“-Ansatzes durch Modell-Versionierung und Instanzmigration

Tina Schliemann

Studiengang:	Softwaretechnik
Prüfer:	Jun.-Prof. Dr.-Ing. Dimka Karastoyanova
Betreuer:	Dipl.-Inf. Mirko Sonntag
begonnen am:	30.08.2010
beendet am:	01.03.2011
CR-Klassifikation:	H.4.1 Workflow-Management
	H.3.5 Web-based Services

Inhaltsverzeichnis

Abkürzungsverzeichnis

AJAX – Asynchronous JavaScript and XML

Apache ODE – Apache Orchestration Director Engine

BPEL – Business Process Execution Language

BPMN – Business Process Modeling Notation

DAO – ODE Data Access Objects

GUI – Graphical User Interface

IL – ODE Integration Layer

JACOB – ODE's Java Concurrent Objects

NMR – Normalized Message Router

RPC – Remote Procedure Call

SGML – Standard Generalized Markup Language

SMX4 – Apache ServiceMix 4.0

SOA – Service Oriented Architecture

VPU – Virtual Processing Unit

W3C – World Wide Web Consortium

WSDL – Web Service Description Language

XML – Extensible Markup Language

XSD – XML Schema Definition

1 Einleitung

Seit kurzem gibt es Bestrebungen, die konventionelle Workflow-Technologie in der Wissenschaft einzusetzen. Für die Lösung komplexer Probleme in der Medizin oder anderen Wissenschaften sind Simulationstechnologien essentiell wichtig geworden. Die Erwartungen der Wissenschaftler an die Workflow-Technologie sind in den letzten Jahren durch neue Entwicklungen in der Hardware sowie der Modellierungs- und Simulationstechnik stetig gestiegen. Diese Anforderungen zu erfüllen, ist Aufgabe des SimTech-Clusters, einem Forschungsprojekt der Universität Stuttgart, in dessen Rahmen auch diese Diplomarbeit stattfindet.

Mit Hilfe der Workflow-Technologie sollen Wissenschaftler, um den Programmieraufwand gering zu halten, ihre Simulationen und Experimente graphisch modellieren können. Diese graphischen Modelle sollen danach ausgeführt werden. Durch diese Neuerungen sollen die Wissenschaftler mehr Konzentration auf ihr eigentliches Forschungsgebiet lenken können. Sehr von Vorteil sind dabei die Automatisierung und Robustheit der Software.

In den letzten Jahren wurde die Anwendung von BPEL für wissenschaftliche Workflows untersucht [1][2][3]. Die Sprache BPEL bietet Wissenschaftlern einige entscheidende Vorteile. Durch die Fehlerbehandlung können auftretende Probleme abgefangen werden und Anwendungen miteinander über Web Service verbunden werden. Alle Aktivitäten werden als einzelne Transaktionen abgearbeitet. Es existieren mehrere BPEL-Engines, die eine persistente Speicherung von Prozessinstanzen ermöglichen und sich durch ihre Robustheit auszeichnen.

1.1 Motivation

BPEL konzentriert sich hauptsächlich auf die Erstellung von geschäftlichen Workflows. Im Gegensatz zu einem wissenschaftlichen Workflow kennt man bei einem geschäftlichen Workflow die Prozesslogik meist bereits zur Designtime. Bei wissenschaftlichen Workflows ist das anders. Hier werden häufig zur Laufzeit die Workflows noch geändert bzw. sogar erst während der Laufzeit entwickelt. Durch das experimentelle Vorgehen des Wissenschaftlers verschmelzen die Phasen zur Erstellung und Ausführung von Workflows.

Heutige Workflowmaschinen bieten die Möglichkeit, mehrere Versionen eines Workflows parallel zur Verfügung zu stellen. Standardmäßig ist nur eine der Versionen aktiv. Eine neue Workflow-Instanz läuft dann in der Regel nach der aktuellsten Version des Workflows. Durch die in der Wissenschaft üblichen Änderungen zur Laufzeit reicht dies nicht aus, um die Anforderungen eines Einsatzes in einem wissenschaftlichen Umfeld zu erfüllen. Es muss möglich sein, auch ältere Modell-Versionen zu instanziieren, um zum Beispiel ältere Experimente erneut ausführen zu können.

Eine weitere Eigenschaft der traditionellen Workflow-Technologie ist, dass Workflow-Instanzen nach der Ausführung ihrer letzten Aktivität automatisch beendet sind. Um der explorativen Workflow-Entwicklung von Wissenschaftlern gerecht zu werden, ist es erforderlich, abgelaufene Workflow-Instanzen im „Suspended“-Zustand zu halten. Dadurch, dass die Instanz am Leben erhalten wird, können Wissenschaftler das Experiment noch beeinflussen und beispielsweise weitere Aktivitäten einfügen oder Teile des Experiments wiederholen.

1.2 Ziele der Arbeit

Ziel der Arbeit ist es, ein Konzept für BPEL zu entwickeln, das es Wissenschaftlern erlaubt, weitere Logik in laufende Prozessinstanzen einzufügen und dadurch ihre Experimente fortführen zu können. Dadurch wird die explorative Entwicklung von Workflows ermöglicht. Folgende Aufgaben gilt es dabei zu lösen:

- Eine Instanz soll nach erfolgreicher Beendigung automatisch am Leben erhalten werden, um ein späteres Hinzufügen von weiterer Logik zu ermöglichen.
- Es soll möglich sein, eine neue Version eines Prozessmodells zu deployen und dabei sowohl die neue als auch die alte Prozessmodell-Version aktiv (d.h. instanziierbar) zu halten.
- Eine oder mehrere laufende Instanzen der alten Modellversion sollen auf die neue Modellversion migriert werden können. Zur Vereinfachung dieser komplexen Aufgabe wird in dieser Arbeit davon ausgegangen, dass die betrachtete(n) Instanz(en) migriert werden kann/können. Das heißt die Modelländerungen betreffen nur das zukünftige Verhalten der Instanzen. Das Prüfen der Migrierbarkeit von Instanzen ist bereits in vorherigen Arbeiten behandelt worden [4].
- Es wird ein Mechanismus benötigt, um laufende Prozessinstanzen zu beenden.
- Um die Anwendbarkeit des Konzeptes zu zeigen, wird es prototypisch für eine bereits vorhandene BPEL Workflow Engine implementiert.

1.3 Aufbau der Arbeit

Kapitel 1 Im weiteren Verlauf dieses Kapitels wird die Arbeit zu anderen wissenschaftlichen Arbeiten abgegrenzt.

Kapitel 2 – Grundlagen Dieses Kapitel beschäftigt sich mit den für das Verständnis der Diplomarbeit benötigten Technologien.

Kapitel 3 – Apache ODE Hier werden der grundlegende Aufbau sowie die benötigten Komponenten der Apache ODE beschrieben.

Kapitel 4 – Konzeption einer Deploy New Version-Strategie In diesem Kapitel wird zuerst der aktuelle Stand der Wissenschaft beschrieben. Anschließend wird auf den gewünschten Funktionsumfang der Deploy New Version-Funktionalität eingegangen, sowie auf die möglichen Änderungen an den einzelnen BPEL-Aktivitäten.

Kapitel 5 – Prototypische Umsetzung der Strategie Hier wird die prototypische Implementierung der Deploy New Version-Funktion an der Apache ODE beschrieben.

Kapitel 6 – Anwendungsbeispiel Dieses Kapitel beschreibt ein Anwendungsbeispiel des Prototyps.

Kapitel 7 – Zusammenfassung und Ausblick Abgeschlossen wird diese Diplomarbeit mit einer Zusammenfassung und einem Ausblick auf offene Fragenstellungen.

1.4 Definitionen

In dieser Arbeit werden die Begriffe Prozess, Instanz und Modell folgendermaßen verwendet:

Prozessmodell / Workflowmodell / Modell bezeichnet das undeployte aber deploybare BPEL-Prozessmodell im Sinne des Deployment Bundles.

Prozess / Workflow bezeichnet das auf der Apache ODE deployte Prozessmodell.

Instanz eine Ausführung des Prozesses.

1.5 Verwandte Arbeiten

In diesem Kapitel werden Arbeiten, die sich mit ähnlichen Problemstellungen wie der Deploy New Version-Funktionalität beschäftigen, vorgestellt.

1.5.1 Migrating WS-BPEL Process Instances

Andreas Fritzler behandelt in seiner Diplomarbeit Migrating WS-BPEL Process Instances [5] einen Ansatz zur Migration einer Prozessinstanz von einer Workflow-Maschine zu einer anderen Workflow-Maschine. Die Workflow-Maschine, auf die migriert wird, ist in diesem Fall die Apache ODE. Dazu wird die Instanz in den Zustand SUSPENDED überführt und die Instanzdaten in ein Zwischenformat gespeichert. Dieses Zwischenformat ist ein XML-Format. Beim Import der Instanzdaten gibt es zwei mögliche Szenarien:

- Die Prozess-ID ist dieselbe ID wie auf der alten ODE Instanz
- Die Prozess-ID ist eine andere ID wie auf der alten ODE Instanz.

Wenn die Prozess-ID dieselbe ID ist, wird das Zwischenformat mit den Instanzdaten importiert und über eine *recreateInstance()*-Methode als Instanz abgespeichert. Der Ausführungszustand der Instanz beinhaltet alle Daten, die zum Fortführen der Instanz benötigt werden. Die Instanz kann jetzt wieder gestartet werden. Wenn die Prozess-ID eine andere ID ist, müssen in den Instanzdaten, die im Zwischenformat vorliegen, zuerst jede Prozess-ID der alten ODE Instanz durch die korrekte Prozess-ID der neuen ODE Instanz ersetzt werden. Daraufhin wird die Instanz wie im anderen Szenario beschrieben importiert.

1.5.2 Enforcement auf laufenden BPEL-Prozessen

M.Kern beschreibt in seiner Diplomarbeit Enforcement[1] auf laufenden BPEL-Prozessen [6] einen Ansatz zur Modifikation von laufenden Prozessinstanzen. Der Ansatz der eventbasierten Instanzmodifikation beruht auf Events, die jede ausgeführte Aktivität auslöst. Die Modifikation wird erst unmittelbar vor der Ausführung der vorhergegangenen Aktivität, die das entsprechende Event ausgelöst hat, vorgenommen.

Um eine neue Aktivität einzufügen wird vom Workflow-Administrator über ein Web Service die neue Aktivität im Eventhandler registriert. Registriert werden die Daten über die Position der neuen Aktivität im Prozessverlauf, sowie die Definition der Aktivität. Die Modifikation wird ausgeführt sobald ein Event auftritt, für das die Modifikation registriert wurde. Daraufhin wird die zusätzlich eingefügte Aktivität ausgeführt. Das Entfernen von Aktivitäten innerhalb einer Instanz wird realisiert

[1] Enforcement: Englisch für Durchführung oder Erzwingung

durch das Ersetzen einer Aktivität durch eine leere Aktivität oder das Überspringen einer Aktivität. Mit diesem Ansatz ist es möglich, einzelne Instanzen oder alle Instanzen „on-the-fly“ zu ändern. Die Modifikation findet entweder in der Execution Queue oder im Event Handler der Workflow-Maschine statt.

2 Grundlagen

In diesem Kapitel werden die Grundlagen, die zum Verständnis dieser Arbeit notwendig sind, erläutert. Da es sich bei dieser Arbeit um eine Modifikation einer bestehenden Workflowmaschine handelt, werden als erstes Workflow und Workflowmaschinen im Allgemeinen erklärt. Die Modifikation der Workflowmaschine soll das Deployen einer neuen Version eines Prozessmodells ermöglichen. Deshalb werden als nächstes die für ein Prozessmodell benötigten Web Service-Technologien erklärt.

2.1 Serviceorientierte Architektur

Gernot Starke und Stefan Tilkov definieren SOA [7] folgendermaßen:

„Eine serviceorientierte Architektur (SOA) ist eine unternehmensweite IT-Architektur, deren zentrales Konstruktionsprinzip lose gekoppelte Services (Dienste) sind. Services realisieren Geschäftsfunktionen, die sie über eine implementierungsunabhängige Schnittstelle kapseln. Zu jeder Schnittstelle gibt es einen Servicevertrag, der die funktionalen und nichtfunktionalen Merkmale (Metadaten) der Schnittstelle beschreibt. Die Nutzung (und Wiederverwendung) von Services geschieht über (entfernte) Aufrufe (»Remote Invocation«).“

Dienste, über die Funktionalitäten bereitgestellt werden, sind der grundlegende Bestandteil einer serviceorientierten Architektur. Die wohl bekannteste Darstellung ist das SOA-Dreieck, das die Grundprinzipien von SOA darstellt.

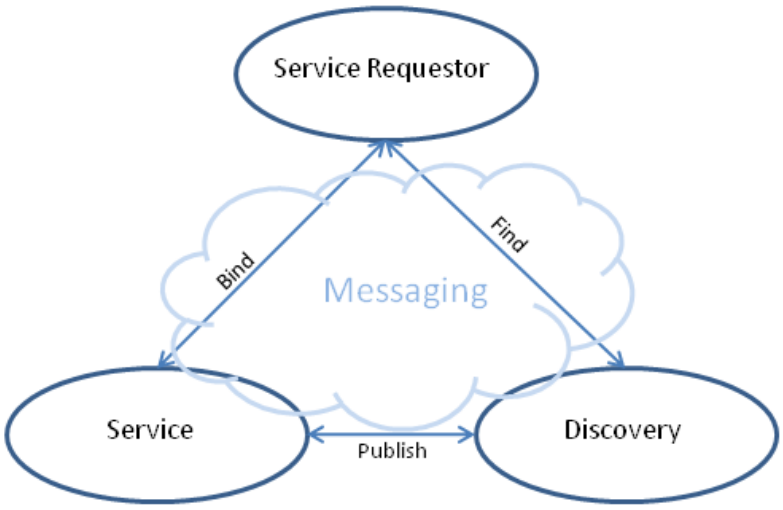


Abbildung 1: SOA-Dreieck angelehnt an [8].

Die *Discovery Facility* stellt einen Suchmechanismus für die Dienste zur Verfügung und stellt die Metadaten der Dienste bereit. Der *Service Requestor*, der einen Dienst benutzen möchte, stellt seine Suchanfrage an die *Discovery Facility* und bekommt die Metadaten eines passenden Dienstes zurückgeliefert. Mit Hilfe dieser Daten kann der *Service Requestor* den Dienst aufrufen. Wenn dies zur Laufzeit passiert, wird es als *dynamic binding* bezeichnet.

2.2 Workflows und Workflow-Maschinen

In diesem Kapitel werden Workflows und Workflowmaschinen grundlegend erläutert, ohne auf Techniken, die zur Umsetzung benötigt werden, einzugehen.

2.2.1 Workflows

Ein Workflow entsteht aus einem Prozess- oder Geschäftsmodell aus der realen Welt, indem das Modell auf einem Rechner ausführbar gemacht wird[2]. Ein Workflowmodell kann dabei ein Teil eines größeren Prozessmodells sein oder aber das gesamte Prozessmodell abbilden. Einzelne Aktivitäten bilden dabei die Grundlage eines Workflows. Immer stehen diese Aktivitäten in einer Abhängigkeit zueinander. Eine Aktivität kann dabei entweder eine atomare Aktivität sein oder einen untergeordneten Prozess aufrufen. Der Anfang und das Ende eines Workflows sind definiert, der Ablauf ist organisiert.

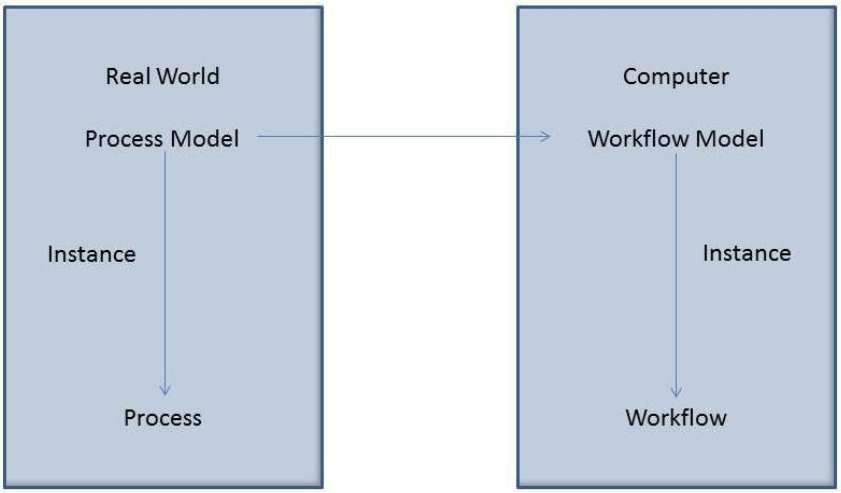


Abbildung 2: Prozesse und Workflows. Angelehnt an [9]

Prozess- und Workflowmodelle haben drei voneinander unabhängige Dimensionen [9]. Die erste Dimension stellt die Prozesslogik dar. Sie wird auch als „*what*"-Dimension bezeichnet und beschreibt, welche Aktivitäten in welcher Reihenfolge ausgeführt werden müssen. Die Aktivitäten können entweder sequentiell, also nacheinander, oder parallel ausgeführt werden**.** Als zweite Dimension gibt es die „*who*"-Dimension, auch Organisations-Dimension genannt. Diese Dimension beschreibt den Aufbau eines Unternehmens, Abteilungen, Rollen und Menschen. Diese Informationen werden gebraucht, um festzulegen, wer eine bestimmte Aktivität ausführen soll. Dieses *wer* kann dabei eine einzelne Person aber auch eine Gruppe von Personen sein, die alle die Fähigkeit haben, diese Aktivität zu bearbeiten. Falls die Aktivität keine Interaktion mit einem Menschen erforderlich macht, wird sie vom Workflowsystem weiterverarbeitet. Als „*with*"-Dimension wird die dritte Dimension bezeichnet, die IT (Information Technology)-Dimension. Sie legt fest, welche Techniken zur Ausführung der Aktivitäten benötigt werden.

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

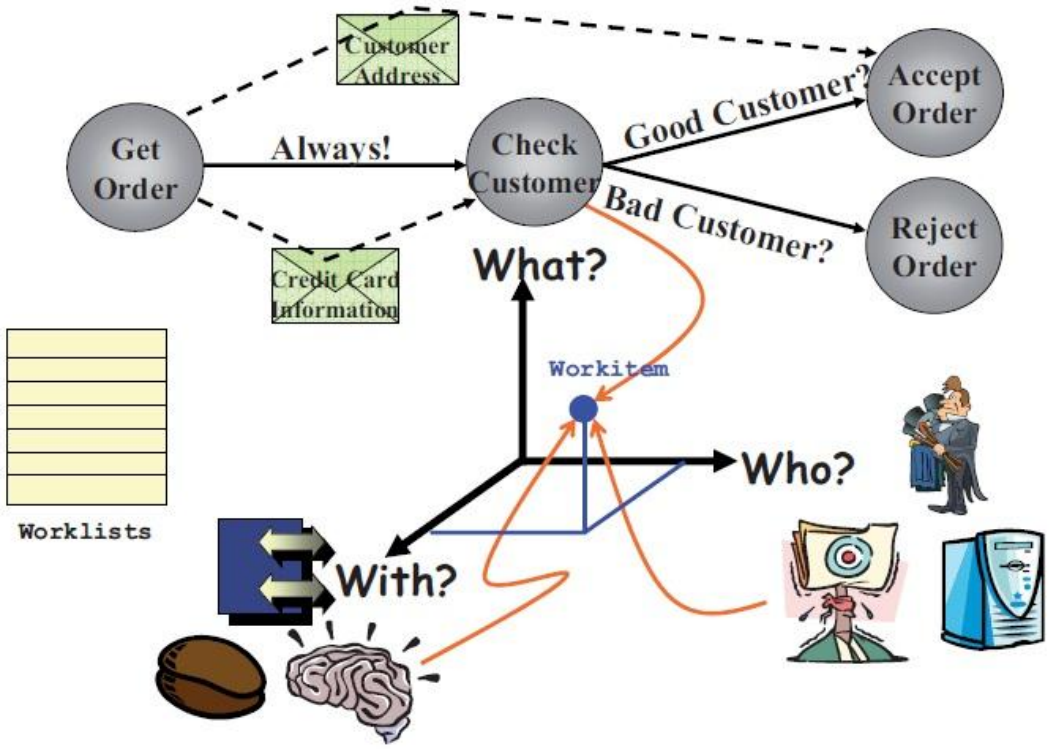


Abbildung 3: Dimensionen eines Workflows [9].

2.2.2 Workflow-Maschinen

M. Böhm, S. Jablonski, und W. Schulze definieren ein Workflow-Management-System [10] folgendermaßen:

„Workflow-Management-Systeme haben eine möglichst vollständige Rechnerunterstützung der Ablauforganisation von Unternehmen zum Ziel. Es ist offensichtlich, dass sie im Kern dem Management von Arbeitsvorgängen (Arbeitsabläufen) dienen. Unter Management ist auch die Verantwortung für die Steuerung eines Systems - insbesondere für die Steuerung seiner Prozesse - zu verstehen. Daher ist als deutsche Übersetzung für Workflow-Management-System die Bezeichnung Vorgangssteuerungssystem gebräuchlich.“

Workflow-Management-Systeme haben viele verschiedene Aufgaben:

- das Bereitstellen von benötigten Daten und Tools,
- das Verwalten von Daten,
- das Steuern von Aufgabenbearbeitung und Kontrollflüssen,
- das Aufrufen von Applikationsprogrammen sowie
- das Benachrichtigen der Benutzer über anstehende Aufgaben.

In Abbildung 4 sind die Hauptkomponenten eines Workflow-Management-Systems und ihre Beziehung zueinander abgebildet. Für diese Arbeit ist die Workflow-Maschine, im Bild als Workflow-Engine bezeichnet, von zentraler Bedeutung.

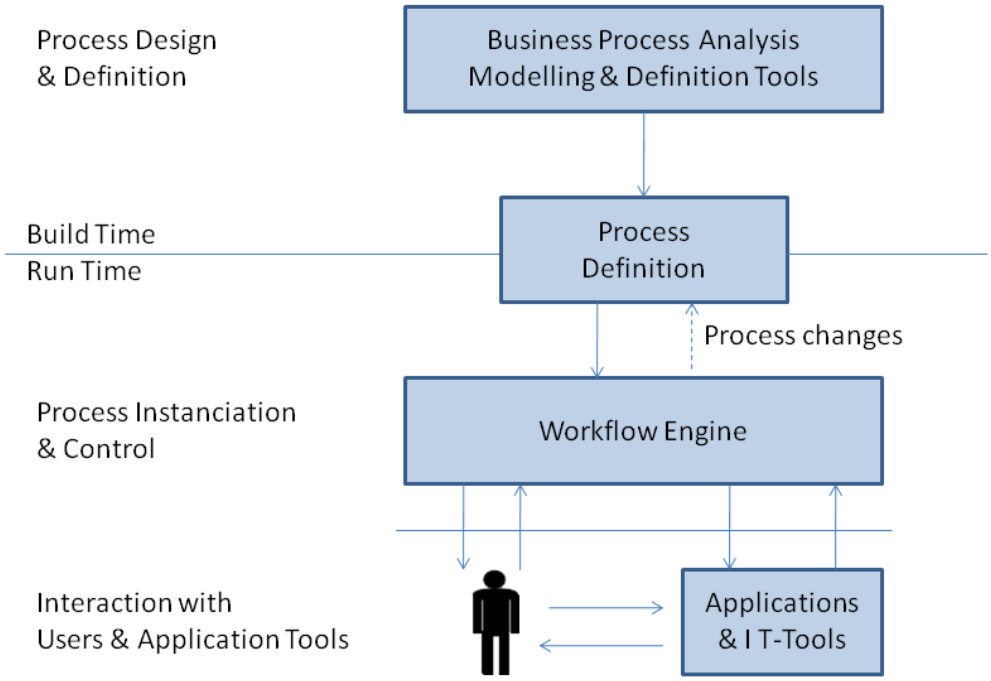


Abbildung 4: Charakteristik eines Workflow-Management-Systems. Angelehnt an [11].

Eine Workflow-Maschine kann die modellierten Prozessmodelle ausführbar machen und eine Prozessinstanz davon erzeugen. Diese Instanz wird von der Workflow-Maschine gesteuert und verwaltet. Alle relevanten Daten werden von der Workflow-Maschine verwaltet und weiterverarbeitet. Beispielsweise werden Workitems erzeugt und den passenden Workflow-Teilnehmern zugewiesen. Ein Workitem ist die Darstellung einer Aufgabe. Diese Teilnehmer können in Organisationseinheiten oder Rollen gruppiert werden.

2.3 Verwendete Web Technologien

In diesem Kapitel werden die Technologien, die zur Umsetzung von Workflows benötigt werden, erläutert.

2.3.1 Web Service

Das World Wide Web Consortium (W3C) definiert einen Web Service 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)." [3]

Web Services sollen eine standardisierte Interoperabilität zwischen verschiedenen Systemen, die auf unterschiedlichen Rechnern betrieben werden, ermöglichen. Dabei sollen Informationen in möglichst oft wiederverwendbaren und voneinander unabhängigen Diensten gebündelt werden. Dem Benutzer

[3] http://www.w3.org/TR/ws-arch - W3C WS-Architektur

soll die Implementierung der Dienste möglichst verborgen bleiben. Der Anbieter stellt die Dienste über eine Schnittstelle bereit. Dabei werden an die Umgebung unterschiedliche Anforderungen gestellt, die in [12] und [13] wie folgt beschrieben sind.

- Geheimhaltung der Anwendungslogik, nur die Schnittstellenbeschreibungen werden veröffentlicht
- Wiederverwendbarkeit von Diensten
- Lose Kopplung
- Orchestrierung von Diensten
- Alle zur Nutzung der Dienste benötigten Informationen sind in einem formalen Vertrag zusammengefasst. Solche Informationen sind beispielsweise die Spezifikationen der Schnittstellen, einzelne Methoden und Protokoll- und Adressierungsinformationen.

2.3.2 XML

Die Extensible Markup Language (XML) ist ein vom W3C definierter Standard zur strukturierten Darstellung von Daten. Bei der Entwicklung von XML waren die wichtigsten Ziele[4]:

1. XML soll überall im Internet benutzbar sein
2. XML soll eine Vielfalt von Applikationen unterstützen
3. XML soll kompatibel zu SGML (Standard Generalized Markup Language) sein
4. Es soll einfach sein, Programme zu schreiben, die XML-Dateien verarbeiten
5. Die Anzahl optionaler Features soll auf ein Minimum beschränkt werden, idealerweise bei null liegen
6. XML-Dokumente sollen möglichst leserlich und klar strukturiert sein
7. Das XML-Design soll sich schnell erstellen lassen
8. Das XML-Design soll formal und präzise sein
9. XML-Dokumente sollen einfach zu erstellen sein
10. Kürze im XML Markup ist von geringer Bedeutung

XML wird vor allem im Internet für den plattformunabhängigen Austausch von Daten eingesetzt. Ein XML-Dokument ist vom Menschen lesbar, da es aus Textzeichen besteht und per Definition keine Binärdaten enthält.

Der Aufbau eines XML-Dokumentes stellt eine Baumstruktur dar. Es besitzt genau ein Wurzelelement. In diesem Element werden die globalen *Namespaces* definiert, die sicherstellen, dass keine Doppeldeutigkeiten bei Überschneidungen mit anderen XML-Daten entstehen können. Alle Elemente des Baumes beginnen mit einem Start-Tag *<active>* und enden mit einem End-Tag *</active>.* Elemente ohne Kind-Elemente können auch in sich geschlossen werden *<service name="wns:Hello Service" />.* Ein Kind-Element muss geschlossen werden, bevor ein übergeordnetes Element geschlossen werden kann oder ein Geschwisterelement geöffnet werden kann. Ein Element kann Attribute, die zusätzliche Informationen bereitstellen und Verarbeitungsanweisungen beinhalten. Des Weiteren kann über *<!-- Kommentartext-->* ein Kommentar in das XML-Dokument eingefügt werden. XML-Dokumente können in drei Dokumentarten unterteilt werden:

[4] http://www.w3.org/TR/2008/REC-xml-20081126/

- dokumentzentriert: hauptsächlich für den menschlichen Gebrauch erstellte Dokumente
- datenzentriert: hauptsächlich zur maschinellen Verarbeitung erstellte Dokumente und
- semistrukturiert: eine Mischung von datenzentriert und dokumentzentriert.

XML-Dokumente können von Parsern ausgelesen, interpretiert und modifiziert werden.

In Listing 1 ist der beispielhafte Aufbau eines XML-Dokumentes zu sehen.

```
<deploy xmlns=
xmlns:pns="http://ode/bpel/unit-test"
xmlns:wns="http://ode/bpel/unit-test.wsdl">
        <process name="pns:HelloWorld2">
                <active>true</active>
                <provide partnerLink="helloPartnerLink">
                        <service name="wns:HelloService" port="HelloPort"/>
                </provide>
        </process>
</deploy>
```

Listing 1: Aufbau eines XML-Dokumentes

2.3.3 Web Service Description Language

Die *Web Service Description Language* (WSDL) [5]ist eine vom World Wide Web Consortium entwickelte Sprache, um Web Services zu beschreiben. Der neueste Standard ist der W3C WSDL 2.0. Diese Arbeit beruht jedoch auf dem WSDL 1.1 Standard, der hier auch vorgestellt wird.

WSDL beschreibt die verwendeten Nachrichten und Datentypen, die zum Aufruf eines Web Services benötigt werden. Des Weiteren wird die Schnittstelle der Operation beschrieben. Eine Web Service-Beschreibung in WSDL besteht aus zwei Teilen, dem abstrakten und dem konkreten Teil. Der abstrakte Teil beschreibt die Funktionalität des Web Services. Da in diesem Teil keine sprach- oder maschinenspezifischen Elemente vorkommen, kann dieser wiederverwendet werden. Der konkrete Teil definiert, wo der Web Service zur Verfügung steht und wie auf ihn zugegriffen werden kann.

In WSDL sind dafür folgende Konzepte spezifiziert. Diese Konzepte und ihr Zusammenspiel werden später erläutert.

- Port Type

```
<wsdl:definitions . . . . >
        <wsdl:portType name="nmtoken">
                <wsdl:operation name="nmtoken" . . . . /> *
        </wsdl:portType>
</wsdl:definitions>
```

Listing 2: WSDL-Port Type

[5] http://www.w3.org/TR/wsdl/

- Port

```
<wsdl:definitions . . . . >
        <wsdl:service . . . . > *
                <wsdl:port name="nmtoken" binding="qname"> *
                        < - - extensibility element (1) - - >
                </wsdl:port>
        </wsdl:service>
</wsdl:definitions>
```

Listing 3: Port

- Operation

```
<definitions .... >
        <binding .... >
                <operation .... >
                        <input>
                            <soap:body parts="nmtokens"? use="literal|encoded"?
                            encodingStyle="uri-list"? namespace="uri"?>
                        </input>
                        <output>
                            <soap:body parts="nmtokens"? use="literal|encoded"?
                            encodingStyle="uri-list"? namespace="uri"?>
                        </output>
                </operation>
        </binding>
</definitions>
```

Listing 4: WSDL-Operation Binding

- Message

```
<definitions .... >
        <message name="nmtoken"> *
                <part name="nmtoken" element="qname"? type="qname"?/> *
        </message>
</definitions>
```

Listing 5: WSDL-Message

- Service

```
<wsdl:definitions . . . . >
        <wsdl:service  name="nmtoken"> *
                <wsdl:port . . . ./> *
        </wsdl:service
</wsdl:definitions>
```

Listing 6: WSDL-Service

- Binding

```
<definitions .... >
        <binding .... >
                <soap:binding transport="uri"? style="rpc|document"?>?
        </binding>
</definitions>
```

Listing 7: WSDL-Binding

- Type

```
<definitions .... >
        <types>
                <xsd:schema . . . .  / > *
        </types>
</definitions>
```

Listing 8: WSDL-Type

Das Zusammenspiel der Konzepte wird in Abbildung 5 erläutert.

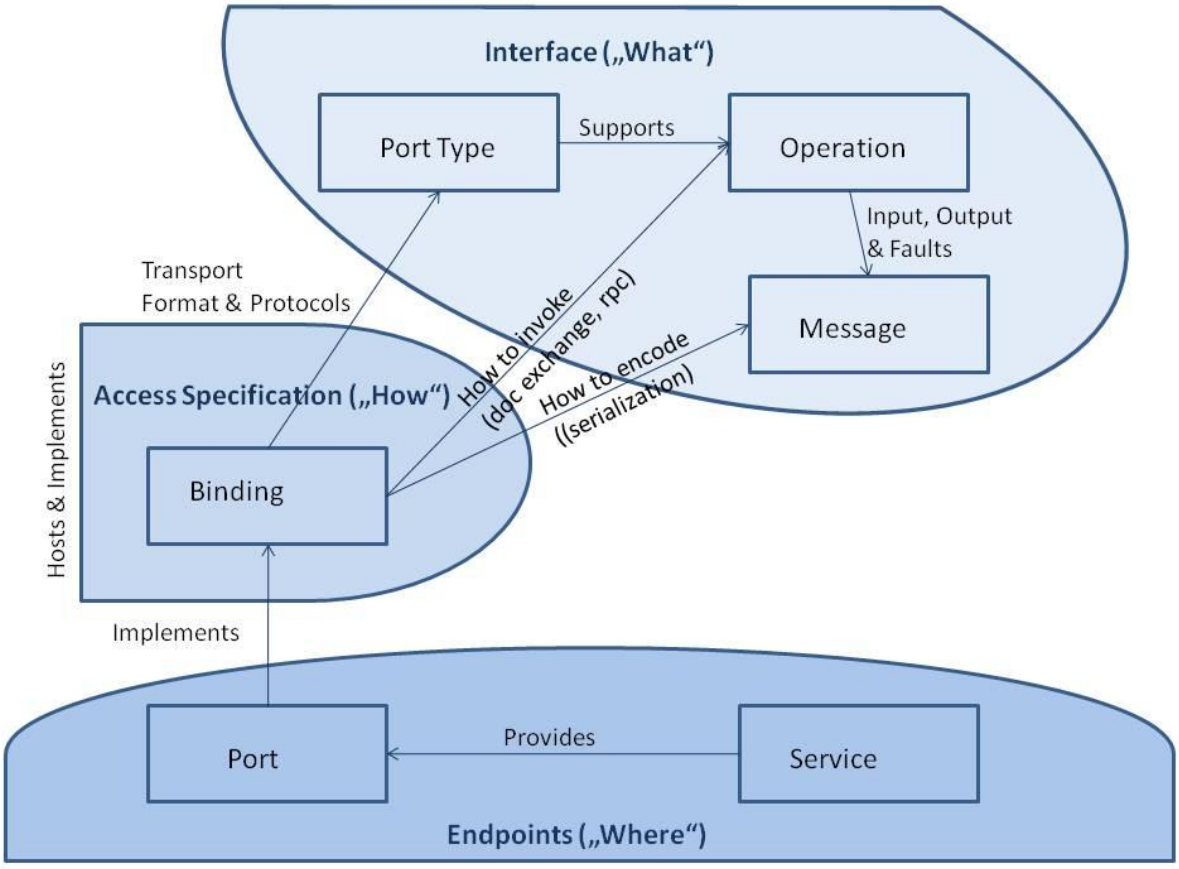


Abbildung 5: Aufbau einer WSDL-Datei. Angelehnt an [8].

Ein *Port Type* besteht aus mehreren abstrakten *Operations* und definiert die vom Web Service zur Verfügung gestellte Funktionalität. Ein *Port Type* hat innerhalb des WSDL-Dokuments einen eindeutigen Namen, der über das *name*-Attribut zugewiesen wird. Mit Hilfe von WSDL können vier Message Exchange Pattern realisiert werden.

- *One-way:* Der *Endpoint* empfängt eine Nachricht.
- *Request-response*: Der *Endpoint* empfängt eine Nachricht und versendet eine dazugehörige Nachricht.
- *Solicit-response:* Der *Endpoint* versendet eine Nachricht und empfängt eine dazugehörige Nachricht.
- *Notification:* Der *Endpoint* versendet eine Nachricht.

Eine O*peration* hat Ein- und Ausgänge. Diese werden durch abstrakte Nachrichten definiert, die eine als XML Schema angegebene abstrakte Datenstruktur beschreiben. Diese Struktur definiert die Nachrichten, die für die Kommunikation mit dem Web Service erwartet werden.
Das *Binding* konkretisiert die abstrakten Konzepte. Es definiert das Nachrichtenformat und das Protokoll, wie auf den Web Service über den Port Type zugegriffen werden soll. Es kann mehrere *Bindings* für einen einzelnen *Port Type* innerhalb eines Dokumentes geben, wobei das *Binding* durch einen eindeutigen Namen definiert ist. Der zu dem *Binding* gehörende *Port Type* wird über das *type*-Attribut zugewiesen. Durch Kombination einer Netzwerkadresse und eines *Bindings* wird ein *Port* definiert. Ähnliche *Ports* werden in einem *Service*-Element zusammengefasst. Das *Type*-Element gruppiert Definitionen von Datentypen, die für den Nachrichtenaustausch relevant sind. WSDL bevorzugt XSD, um ein Maximum an Plattformneutralität und Kompatibilität zu gewährleisten.
Die Implementierung des durch WSDL beschriebenen Web Services ist von der Programmiersprache unabhängig.

2.3.4 BPEL

Die *Business Process Execution Language* (BPEL) [14] hat sich als Standard zur Beschreibung von Geschäftsprozessen durchgesetzt. BPEL ist eine XML-basierte Sprache, die von IBM, BEA Systems und Microsoft entwickelt wurde. Ein Geschäftsprozess ist eine Komposition von Web Services, dessen Geschäftslogik durch XML beschreiben wird. Nach außen wird der Geschäftsprozess wieder als Dienst angeboten, dessen Schnittstelle durch WSDL beschreiben ist.
Die Sprache BPEL ermöglicht die Modellierung von komplexen Kontrollflüssen und die Möglichkeit, mit anderen *Web Services* zu kommunizieren.
Aktivitäten sind die grundlegenden BPEL-Konstrukte. Diese können in zwei Kategorien aufgeteilt werden. Zum einen die *Basic Activities*, die die atomaren Aktivitäten darstellen, zum anderen die *Structured Activities*, die aus *Basic* und *Structured Activities* bestehen und die Modellierung von komplexen Prozessen zulassen.

Basic Acitivities:

- *assign*: Zuweisen eines Variablenwertes
- *invoke*: Aufruf eines Web Services
- *receive*: Warten auf eine Nachricht
- *reply*: Antwort an einen Web Service versenden
- *throw*: Fehler wird signalisiert

- *rethrow*: Fault wird von *fault- handler* an *scope* weitergegeben
- *wait*: Eine bestimmte Zeitspanne oder bis zu einem Zeitpunkt warten
- *empty*: Leere Aktivität
- *exit*: Beendet eine Instanz sofort
- *compensate:* Ruft *compensation-handler* aller *scopes* auf
- *compensateScope:* Ruft *compensation-handler* eines bestimmten *scopes* auf
- *validate:* Validiert XML-Messages
- *extensionActivity:* Erweiterung von BPEL um eine neue Aktivität

Um die Programmlogik zu definieren, existieren folgende *Structured Activities*:

- *sequence*: sequentielle Abarbeitung von Aktivitäten
- *flow*: parallele Ausführung von Aktivitäten
- *while*: Ausführen von Aktivitäten, solange eine boolesche Bedingung erfüllt ist
- *if*: Ausführen einer Aktivität, wenn Bedingung erfüllt ist
- *pick*: Ausführen einer Aktivität durch ein Ereignis
- *scope*: Bündelung von Aktivitäten. Diesem Bündel kann beispielsweise ein *fault- handler*, ein *compensation-handler,* ein *termination-handler* oder ein *event- handler* zugewiesen werden
- *repeatUntil*: Ausführen einer Aktivität bis eine Bedingung erfüllt ist
- *forEach*: Mehrfaches Ausführen derselben Aktivität mit verschiedenen Daten

Der *fault-handler*, *compensation-handler*, und *termination-handler* sind Konzepte zur Transaktionssteuerung in Prozessen. Jeder *fault-handler* enthält eine Anweisung in Form einer Aktivität. Während der Designtime werden für jede *scope*-Aktivität und den *process catch* und *catch-all-fault-handler* definiert. Wenn ein Fehler bei der Verarbeitung der Aktivitäten auftritt, wird ein *fault* geworfen, der von dem von der Workflow-Maschine aktivierten *fault-handler* abgefangen wird. Alle laufenden Aktivitäten innerhalb des *scope* oder *process* werden beendet, das *fault-handling* beginnt. Alle nicht behandelten *faults* werden in den übergeordneten *scope* weitergegeben, für die behandelten *fault* werden die jeweilig definierten Aktivitäten ausgeführt. Ein *catch-all-fault-handler* fängt im Gegensatz zum normalen *fault-handler* alle *faults* ab und verarbeitet sie weiter. *Scopes* sind verschachtelt und der Wurzelknoten stellt immer das *process* Element dar.
Der *compensation-handler* wird nach erfolgreicher Ausführung eines *scopes* aktiviert. Dort werden Informationen zum Undo des *scopes* gespeichert. Ziel ist es, eine Möglichkeit zu schaffen, die Instanz in den Zustand, die sie vor Ausführung dieses *scopes* gehabt hat, zurückzusetzen. Ein *compensation-handler* kann nur durch den übergeordneten *scope* aufgerufen werden. Ist kein *compensation-handler* definiert, wird ein impliziter aufgerufen. Der implizite *compensation handler* ruft das Kompensieren aller innerhalb des *scopes* installierter *compensation-handler* auf.
Wenn ein *scope* terminiert, wird das sogenannte *termination-handling* gestartet, entweder das implizite oder das definierte *termination-handling.* Aus diesem *termination-handler* wird dann beispielsweise das *compensate* aufgerufen.
Zusätzlich zu diesen *handlern* gibt es noch den *event-handler.* Dieser reagiert durch Ausführung bestimmter Aktivitäten auf definierte *application-messages* oder Timeouts.
BPEL ermöglicht die Modellierung von ausführbaren und abstrakten Prozessen. Im Gegensatz zu abstrakten Prozessen, die der Beschreibung des Verhaltens von Prozessen dienen, können ausführbare Prozesse auf einer Workflow-Maschine deployed werden. Abstrakte Prozesse können

eine Sicht auf einen ausführbaren Prozess oder ein Template für das Entwickeln von Prozessen darstellen. Deswegen werden sie auch als *Behavioral Interface* bezeichnet. In Abbildung 6 ist der Zusammenhang zwischen einem abstrakten und einem ausführbaren Prozess grafisch dargestellt.

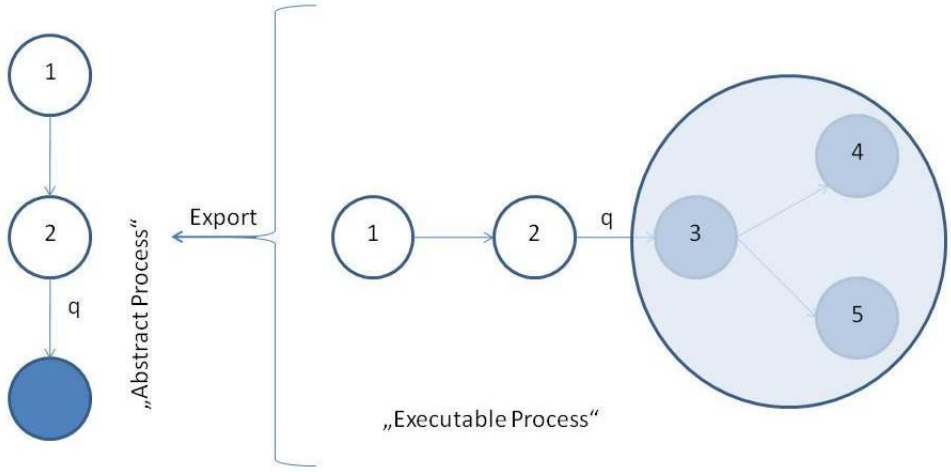


Abbildung 6: Zusammenhang abstrakter und ausführbarer Prozess. Angelehnt an [9].

Im Vergleich zu höheren Programmiersprachen bietet BPEL einen reduzierten Sprachumfang an. BPEL ist auf das prozessorientierte Komponieren von Web Services, was auch als „Programmieren im Großen“ bekannt ist, optimiert. Im Anhang in Abbildung 33 ist die graphische Modellierung eines BPEL-Prozesses dargestellt. Abbildung 34 stellt den Code des BPEL-Prozesses dar. Dieser zeigt die enge Verknüpfung zwischen WSDL und BPEL. Aufzurufende Web Services werden über WSDL-Konstrukte spezifiziert, ebenso werden WSDL-Messages verwendet.

2.3.5 SOAP

SOAP stand ursprünglich für Simple Object Access Protocol und ist eine vom W3C definierte Nachrichtenarchitektur für den strukturierten Austausch von Daten in einem Netzwerk. Die Apache ODE unterstützt derzeit nur Version 1.1. Seit Version 1.2 wird SOAP nicht mehr als Akronym gebraucht. SOAP kann unabhängig von dem darunterliegenden Protokoll eingesetzt werden, wobei die Struktur als XML-Infoset definiert wird. In der aktuellen Spezifikation 1.2 wird ein Framework durch folgende Punkte spezifiziert[6]:

- *SOAP Processing Model:* Ein Verarbeitungsmodell, das Regeln zum Abarbeiten der SOAP-Nachrichten definiert.
- *SOAP Extensibility Model*: Ein Erweiterungsmodell, das die Konzepte und Funktionen der SOAP-Module definiert.
- *SOAP Protocol Binding Framework*: Ein Framework für die Protokollbindung , das das Versenden der SOAP-Nachrichten über das darunterliegende Protokoll zwischen den Knoten (*Nodes*) definiert.
- *SOAP Message Construct*: Gibt den Aufbau und die Struktur von SOAP-Nachrichten an.

Das äußerste Element einer SOAP-Nachricht ist der *Envelope*. Darin enthalten sind maximal ein *Header*-Element und genau ein *Body*-Element. Ein *Header*- Element besteht aus beliebig vielen *Headern*, ein *Body*-Element kann beliebig viele Kind-Elemente haben. Die SOAP-Spezifikation gibt vor,

[6] http://www.w3.org/TR/soap/

wie die Elemente verarbeitet werden, nicht aber den Inhalt der Elemente, der durch die Anwendung bestimmt wird. Das SOAP-*Header*-Element beinhaltet Daten, um eine SOAP-Nachricht auf eine dezentrale und modulare Weise zu erweitern. Es dient dazu, Informationen, die nichts mit dem eigentlichen Payload der Anwendung zu tun haben, zu transportieren. Der SOAP-*Body* beinhaltet den eigentlichen Inhalt der SOAP-Nachricht, der vom Sender zum endgültigen Empfänger (*Ultimate SOAP Receiver*) übermittelt werden soll.
Es gibt zwei verschiedene Typen von SOAP-Nachrichten. Die *Document-Style* und die *RPC-Style (Remote Procedure Call*) Nachrichten. Abbildung 7 ist die Struktur einer *RPC-Style* SOAP-Nachricht zu sehen.

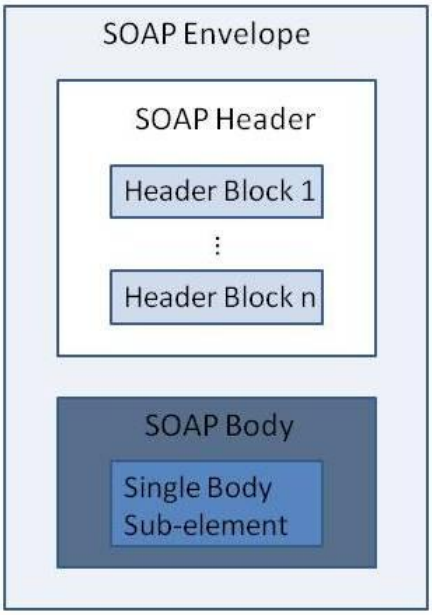


Abbildung 7: Struktur einer RPC-Style SOAP-Nachricht. Angelehnt an [8].

RPC-Style Nachrichten bilden einen entfernten Methodenaufruf in einer SOAP-Nachricht ab. Dazu werden alle notwendigen Informationen in die Nachricht kodiert. Der Name der aufzurufenden Methode steht im Wurzelelement des *Bodys*. Da in diesem Fall nur genau ein *Body*-Element existiert, wird die Interpretation der Nachricht durch die SOAP-Spezifikation vorgegeben. Der Empfänger generiert eine Antwort-Nachricht, die die Rückgabewerte des Methodenaufrufs enthält.
Document-Style Nachrichten enthalten keine Informationen, wie sie zu interpretieren sind. Die Anwendung definiert im Voraus die Semantik der Nachricht. Eine SOAP-Nachricht kann dann mehrere *Body*-Elemente besitzen und der Empfänger muss nicht zwangsläufig eine *Response*-Nachricht generieren. Das *SOAP Processing Model* geht davon aus, dass die Zustellung der Nachrichten von einem Sender zum eigentlichen Empfänger, dem *Ultimate SOAP Receiver,* nicht direkt erfolgt sondern über mehrere Zwischenknoten, die *SOAP Intermediaries*. Es wird beschrieben, wie die Empfänger eine SOAP-Nachricht verarbeiten sollen. Im SOAP-*Header* stehen die Anweisungen für den Empfänger. Über ein *Role*-Attribut wird der *Header* an den entsprechenden Knoten auf dem Pfad zum *Ultimate Receiver* adressiert. Im *Role*-Attribut wird definiert, welche *Header*-Elemente der Empfänger verarbeiten soll. *Header*-Elemente können von den *Intermediariers* verändert, gelöscht oder neu hinzugefügt werden. Des Weiteren dürfen *Intermediaries* Änderungen am SOAP-*Body* vornehmen. Dadurch lassen sich *Quality-of-Service*-Eigenschaften, die nicht durch das darunterliegende Protokoll gegeben sind, realisieren. Ein Beispiel hierfür ist die Verschlüsselung einer

SOAP-Nachricht. Der SOAP-*Body* ist immer an den *Ultimate Receiver* bestimmt. Dieses Verarbeitungsmodell ist in der Abbildung 8 dargestellt.

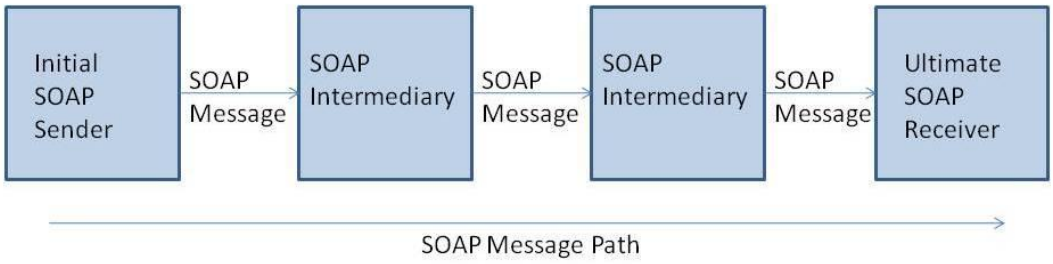


Abbildung 8: SOAP Verarbeitungsmodell

3 Apache ODE

3.1 Grundlagen der Apache ODE

Die Apache ODE (Orchestration Director Engine)[7] wird von der Apache Software Foundation als Top-Level-Projekt entwickelt. Sie ist ein Open Source Workflow-Management-System für BPEL-Prozessmodelle, lizenziert unter der Apache License Version 2.0. Implementiert wird die Apache ODE in Java und basiert auf dem Java Development Kit (JDK) 5.0. Die aktuelle stabile Version ist 1.3.4, auf der auch diese Arbeit beruht.

Die Apache ODE kommuniziert mit Web Services, sendet und empfängt Nachrichten, verarbeitet Daten und Fehler nach den Beschreibungen im BPEL-Prozess. Sie unterstützt lang- und kurzlebige Prozessinstanzen und orchestriert Web Services.

Es gibt drei unterschiedliche Umgebungen, in denen die Apache ODE deployed werden kann:

- Als Web Service in Axis 2. Dafür wird die ODE als WAR gepackt und kann in jedem Application Server deployed und aufgerufen werden.
- Als JBI Servicegruppe. Dafür wird die ODE als ZIP gepackt und kann in einem JBI Container deployed und über NMR aufgerufen werden.
- Als OSGi Bundle in SMX4

Folgende Standards werden in der aktuellen stabilen Version unterstützt:

- WS-BPEL 2.0, bis auf wenige Abweichungen. Ziel ist es die Abwärts-Kompatibilität zu BPEL4WS 1.1 zu erhalten
- WSDL 1.1 und teilweise WSDL 2.0
- SOAP 1.1
- XPath 2.0

Die Apache ODE kann auf jedem Betriebssystem, das Java 5 unterstützt, ausgeführt werden. Dazu wird nur eine der oben genannten Umgebungen benötigt, da momentan keine Standalone-Version der ODE verfügbar ist. DAOs (data access objects) bilden die Grundlage zur Kommunikation mit den Datenbanken der ODE. Es werden zwei DAO-Implementierungen angeboten, OpenJPA und Hibernate. ODE unterstützt die meisten relationalen Datenbanken. Standardmäßig mitgeliefert werden Datenbank-Schemas für Derby und MySQL sowie eine bereits konfigurierte Derby-Datenbank.

Bisher sind keine Instanzmodifikationen möglich. Beim Deployen einer aktualisierten Version eines Prozesses mit demselben Namen, werden alle Instanzen auf der alten Version des Prozesses gelöscht. In der Praxis bedeutet das, dass Prozessmodelle unter einem anderen Namen veröffentlicht werden müssen, wenn Instanzen, die nicht gelöscht werden sollen, auf ihnen aktiv sind und Aktualisierungen am Prozessmodell nötig sind.

[7] http://www.ode.apache.org/

3.1.1 Abweichungen vom WS-BPEL 2.0 Standard

Bis auf wenige Abweichungen wird der WS-BPEL 2.0 Standard unterstützt. Diese Abweichungen betreffen folgende Aktivitäten:

- *<receive>*: kein Support der *<fromPart>*-Syntax, dafür wird das *variable* Attribut genutzt. Des Weiteren können im *variable* Attribut nur Nachrichten-Variablen referenziert werden, obwohl die Spezifikation auch Element-Variablen erlaubt.
  Mehrere Start-Aktivitäten werden nicht unterstützt, ebenso wenig die Anordnungsrichtlinien der Spezifikation, die ODE ist hier deutlich toleranter als die Spezifikation. *conflictingRequest* wird wie *conflictingReceive* behandelt. Es wird immer, wenn *conflictingRequest* auftritt, *conflictingReceive* geworfen. Ein existierendes *validate*-Attribut wird ignoriert.
- *<reply>:* Einschränkungen wie *<receive>.*
- *<invoke>: <toPart>* und *<fromPart>* werden nicht unterstützt. Die Attribute *inputVariable* und *outputVariable* müssen auf eine Nachrichten-basierte (message-typed) Variable referenzieren.
- *<assign>*: Das Validieren von Variablen wird nicht unterstützt, ebenso wenig Zuweisungen innerhalb der Variablendeklaration. Die ODE verwendet derzeit das *expressionLanguage* statt des *queryLanguage* Attributs, um die verwendeten Sprachen innerhalb einer Anweisung *festzulegen.*
- *<pick>:* Einschränkungen wie *<receive>.*
- *<compensate>:* Entspricht der *<compensateScope>*-Aktivität.
- *<validate>:* Bisher nicht implementiert. Wenn *validate* in einem Prozessmodell vorkommt, wird ein Kompilationsfehler geworfen.

3.2 Architektur der Apache ODE

Bei der Entwicklung der Apache ODE [15] waren die Hauptziele, eine zuverlässige, kompakte und aus mehreren eingebetteten Komponenten bestehende Workflow-Maschine zu entwickeln, die langlebige BPEL-Prozesse ausführen kann. Der Fokus bestand darin, kleine losgekoppelte Module zu entwickeln, die zu einer voll funktionsfähigen Workflow-Maschine gruppiert werden können. In Abbildung 9 ist die Architektur der ODE graphisch dargestellt. Es wird das Zusammenspiel der Komponenten erläutert.

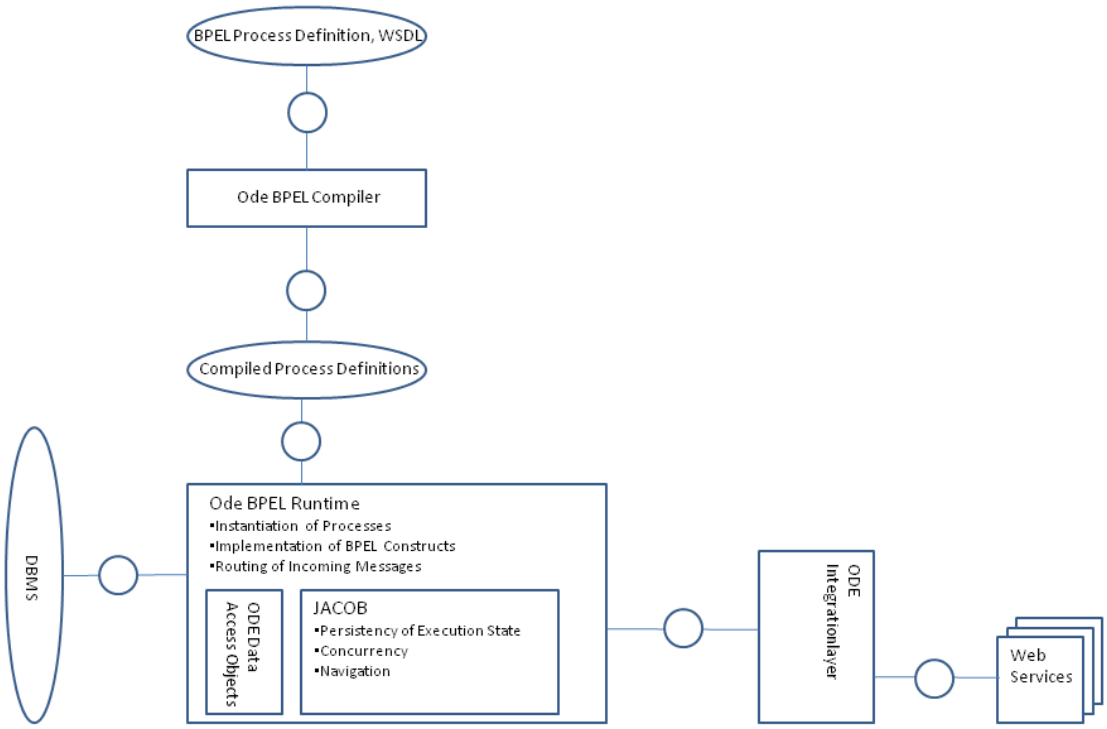


Abbildung 9: ODE Architektur. Angelehnt an [15].

3.3 Komponenten

Die Hauptkomponenten der Apache ODE Architektur sind ODE BPEL Compiler, ODE BPEL Engine Runtime, ODE Data Access Objects (DAOs) und der ODE Integration Layer (IL). Zusammenfassend kann die Architektur folgendermaßen beschrieben werden [15]:

“The compiler converts BPEL documents into a form executable by the run-time, which executes them in a reliable fashion by relying on a persistent store accessible via the DAOs; the run-time executes in the context of an Integration Layer which connects the engine to the broader execution environment (i.e. the "world").”

3.3.1 ODE BPEL Compiler

Der ODE BPEL Compiler kompiliert die einzelnen BPEL-Artefakte, das BPEL-Prozess-Dokument, WSDL-Dokumente und XML Schemas in einen ausführbaren Prozess. Bei erfolgreicher Kompilierung ist das Ergebnis des ODE BPEL Compilers der ausführbare Prozess. Bei nicht erfolgreicher Kompilierung wird eine Fehlerliste, die auf die fehlerhaften Artefakte hinweist, ausgegeben.

Die Struktur des ausführbaren Prozesses ähnelt der Struktur des BPEL-Prozess-Dokuments. Allerdings sind Namen und Typen aus der WSDL-Beschreibung aufgelöst und weitere Objekte, beispielsweise implizite C*ompensation Handler* angelegt. Das kompilierte Prozessmodell wird als .cbp-Datei gespeichert und stellt das wichtigste Artefakt der *BPEL Runtime.* Die *ODE BPEL Engine Runtime* kann diese kompilierten Prozesse ausführen.

3.3.2 ODE BPEL Engine Runtime

Innerhalb des BPEL-Runtime Moduls stellt die ODE BPEL Engine Runtime alles zur Ausführung von kompilierten BPEL-Prozessen zur Verfügung:

- Die Implementierung verschiedenster BPEL-Konstrukte,
- die Logik, wann eine neue Instanz kreiert werden muss,
- zu welcher Instanz eine eingehende Nachricht gehört und
- die Process Management API, die zur Interaktion des Benutzers mit der Maschine benötigt wird.

Um die verlässliche Ausführung von Prozessen in einer unzuverlässigen Umgebung zu gewährleisten, baut die Runtime auf Data Access Objects, die die Persistenz sicherstellen.

3.3.3 JACOB

Die Implementierung der BPEL-Konstrukte zur Laufzeit auf Instanz-Ebene ist mit Hilfe des *ODE Java Concurrent Objects* (Jacob) Framework umgesetzt. Das Framework stellt Funktionalitäten zum Umgang mit Nebenläufigkeit und der Persistenz des Ausführungsstatus zur Verfügung.

Dadurch, dass diese beiden Objekte im Framework implementiert sind, gestaltet sich die Implementierung der BPEL-Artefakte deutlich einfacher, da nur die BPEL-Logik und nicht die Infrastruktur erstellt werden muss. Hieraus resultiert eine strikte Trennung der Ebenen. Jacob stellt eine persistente virtuelle Maschine zur Ausführung von BPEL-Konstrukten dar.

3.3.3.1 Channels

Channels sind Interfaces, die zur Kommunikation zwischen Aktivitäten in der ODE benötigt werden. *TerminationChannel*, *ParentScopeChannel* und *CompensationChannel* sind einige der unterschiedlichen Channels. Einige grundlegende Channels werden jeder Aktivität bei der Erstellung zur Verfügung gestellt, um ihnen die Kommunikation mit der Umgebung zu ermöglichen.

Es existiert keine Implementierung der Channels, sie werden über einen dynamischen Proxy zur Verfügung gestellt. Dies ist eine der Ebenen zur Trennung von Ausführung und Aufruf in Jacob.

3.3.3.2 JacobObject / JacobRunnable

JacobObject stellt ein C*losure* da. C*losures* werden standardmäßig nicht von Java unterstützt. *Closures* reproduzieren einen Teil ihres Erstellungskontextes beim Aufruf, auch wenn dieser Kontext außerhalb der Funktion nicht mehr existiert. *Closures* sind also Programmfunktionen, die ihren eigenen Kontext erhalten. *JacobObject* stellen keine wirklichen *Closures* dar, da sie statisch programmiert sind. Sie erheben aber den Anspruch, die Lücke der fehlenden *Closures* in Java zu schließen. Aufgabe der *JacobObjects* ist es, Methoden zu implementieren, Methoden zur Manipulation von Channels zur Verfügung zu stellen und sich selbst zu vervielfältigen.

JacobRunnable sind *JacobObjects*, die nur eine Methode *run()* implementieren. Da alle Aktivitäten von *JacobRunnable* erben, müssen sie auch ihre Hauptfunktionalität in der *run()*-Methode implementieren. Die Initialisierung findet in den jeweiligen Konstruktoren statt.

3.3.3.3 Channel Listener

Channel Listener stellen das andere Ende eines *Channels* dar. Sie werden allerdings nicht direkt beim Aufruf des *Channels* aufgerufen. Normalerweise werden *Channel Listener* innerhalb der *run()*-Methode einer Aktivität definiert. Oft erben die Objekte von *JacobObject*, so dass die *Jacob*-Runtime später eine eingehende Nachricht zum dazugehörigen *Channel Listener* weiterreichen kann.

3.3.3.4 Virtual Processing Unit and ExecutionQueue

Innerhalb der Virtual Processing Unit (VPU) findet die komplette Jacob-Verarbeitung statt. Bei Aufruf eines *JacobObjects* innerhalb der VPU wird dieses als *Continuation* registriert. Eine *Continuation* verbindet das *JacobObject* mit der *run()*-Methode des *JacobObjects,* um es auszuführen.

Alle von der VPU verarbeiteten Teile werden in der *ExecutionQueue* abgelegt. Sie stellt einen Container dar, um alle Artefakte in Queues zu organisieren, von denen sie geholt und darauf gelegt werden können. Gleichzeitig werden einige Ausführungs-Statistiken von der *ExecutionQueue* aufgezeichnet.

Die VPU ist zudem verantwortlich für die Persistierung des eigenen internen Status. Wenn eine Ausführung gestoppt wird, wird der VPU Status serialisiert und für den späteren Gebrauch gespeichert. *Continuations* bleiben nicht dauerhaft in den VPU-Queues, sondern werden geholt, ausgeführt und verworfen.

3.3.4 ODE Integration Layer

Die *ODE BPEL Engine Runtime* kann nicht alleine existieren, da sie nicht in der Lage ist, mit der „restlichen Welt“ zu kommunizieren. Deswegen baut sie auf dem ODE Intergration Layer (IL) auf. Der Integration Layer bindet die Runtime in die Umgebung ein, beispielsweise existieren Integration Layer für AXIS2 und JBI. Die Hauptfunktion eines Integration Layers ist es, die Kommunikationskanäle für die Runtime zur Verfügung zu stellen. Beim *AXIS2 Intergation Layer* wird dies über die AXIS2-Libraries, die es der Runtime ermöglichen über Web Services zu kommunizieren, realisiert. Beim *JBI Integration Layer* wird die Runtime mit dem *JBI Message Bus* verbunden und kann darüber kommunizieren.

Zusätzlich zur Kommunikation sind die Aufgaben des Integration Layers der Runtime, eine Thread-Planung zur Verfügung zu stellen und den Lebenszyklus der Runtime zu leiten.

3.3.5 ODE Data Access Objects

Die *ODE Data Access Objects* (DAO) sind für die Interaktion zwischen der *ODE BPEL Engine Runtime* und der darunterliegenden Datenbank zuständig. Standardmäßig mitgeliefert wird die Unterstützung einer relationalen JDBC Datenbank. In diesem Fall sind die DAOs mit Hilfen von OpenJPA oder Hibernate implementiert. Es besteht die Möglichkeit, eigene DAOs zu implementieren, um die Unterstützung anderer Datenbanken zu erreichen.

DAOs werden von der *ODE BPEL Engine Runtime* benötigt, um die folgenden Persistenz-Probleme zu lösen:

- Aktive Instanzen – Welche Instanzen wurden gestartet oder laufen
- Nachrichtenverarbeitung – Welche Instanz wartet auf welche Nachricht
- Variablen – Der aktuelle Wert der BPEL Variablen für jede Instanz
- Partner Links – Der aktuelle Inhalt der BPEL Partner Links für jede Instanz
- Status der Prozessausführung – Der serialisierte Status von *Jacob "persistent virtual machine"*

3.4 Management API

Die Management API kann genutzt werden, um zu sehen, welche Prozesse deployed wurden, welche Instanzen gerade ausgeführt werden oder schon beendet sind und um Variablenwerte abzufragen. Hauptsächlich besteht die Management API aus den zwei Interfaces ProcessManagement und InstanceManagement, die als Web Service zur Verfügung gestellt werden.

3.4.1 ProcessManagement

ProcessManagement dient der allgemeinen Verwaltung aller Prozessmodelle. Folgende Methoden stehen zur Verfügung:

- *listAllProcesses():* Listet alle verfügbaren Prozesse mit Informationen, wie ID, Zustand, Version, Status und Endpunkten auf.
- *getProcessInfo():* Listet für einen einzelnen Prozess die Informationen auf.
- *activate():* Aktiviert einen Prozess.
- *setRetired():* Verändert den Prozessstatus auf RETIRED. Dadurch kann der Prozess nicht mehr gestartet werden.

3.4.2 InstanceManagement

InstanceManagement dient zur Verwaltung von Prozessinstanzen der Apache ODE.

- *listAllInstances():* Listet alle existierenden Instanzen auf.
- *resume():* Führt eine pausierte Instanz fort.
- *suspend()*: Pausiert eine Instanz.
- *terminate():* Beendet eine Instanz sofort, ohne *fault* oder *compensation handler.*
- *fault():* Wirft einen Fehler und verhindert die erfolgreiche Ausführung der Instanz.
- *delete():* Löscht eine Prozessinstanz.
- *getInstanceInfo():* Listet für eine Instanz die Informationen auf.
- *getVariableInfo():* Listet alle Informationen über eine Variable auf.

3.5 Deployment Interface

Das Deployment Interface der Apache ODE dient dem Deployment von Prozessen innerhalb der Apache ODE. Dazu stellt es fünf Operationen zur Verfügung:

- *deploy()*: Deployed einen Prozess auf der Apache ODE.
- *undeploy()*: Entfernt einen Prozess von der Apache ODE.
- *listDeployedPackages()*: Listet alle bereits auf der Apache ODE vorhandenen Prozesspakete auf.
- *listProcesses()*: Listet alle in einem Prozesspaket vorhandenen Prozesse auf.
- *getProcessPackage()*: Liefert den Namen des zum Prozess gehörigen Pakets, in dem der Prozess deployed wurde, zurück.

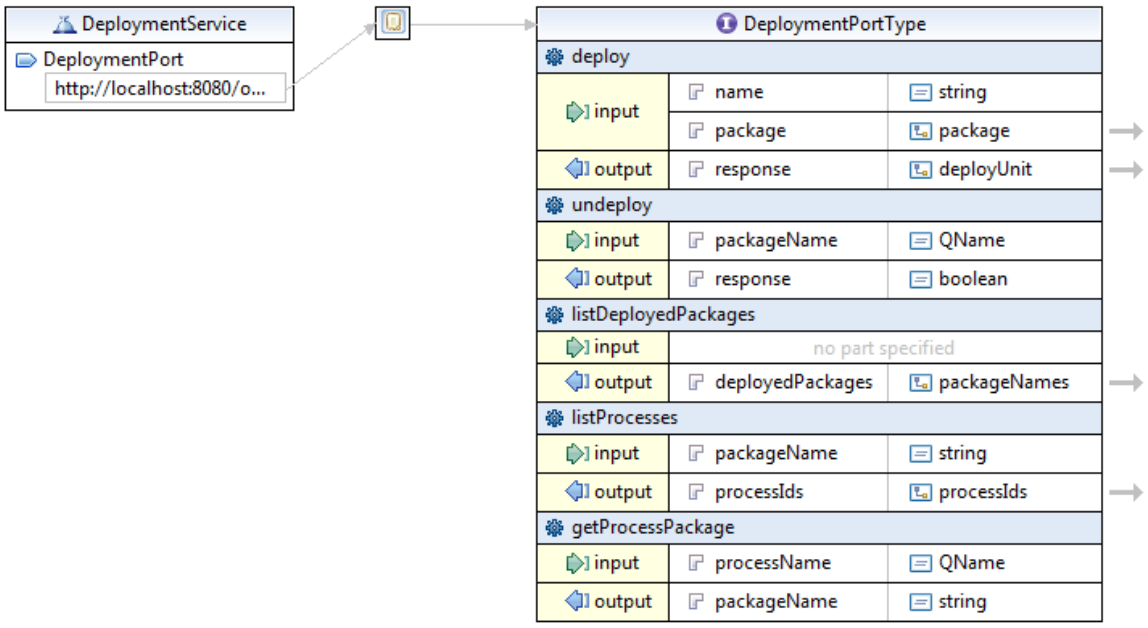


Abbildung 10: Deployment-API der Apache ODE

3.6 Oberfläche GUI

Die Oberfläche der Apache ODE besteht aus vier Reitern.

- Home: ist eine Übersichtsseite über die derzeitigen Prozesse und Instanzen
- Processes: zeigt alle ausführbaren Prozesse mit zusätzlichen Informationen an und stellt die Retitre- und Activate-Funktionalität zur Verfügung
- Instances: zeigt alle Instanzen und ihren Status an. Je nach Status der Instanz steht die Terminate-, Suspend- oder Resume-Funktionalität zur Verfügung
- Deployed: zeigt alle in der Apache ODE verfügbaren Prozesse an und stellt die Undeploy-Funktionalität zur Verfügung

Alle Funktionalitäten, die über Buttons aus der Apache ODE Oberfläche aufrufbar sind, stehen auch als Web Service zur Verfügung.

3.7 Deployment

Die Apache ODE unterstützt zwei unterschiedliche Wege, ein Prozessmodell zu deployen. Der erste Weg führt über einen Deployment-Web Service, der zweite über das direkte Kopieren des Deployment-Bundles in das WEB-INF/processes Verzeichnis der Apache ODE im Dateisystem. Für diese Arbeit ist nur das Deployen über den Web Service interessant.

Prozesse werden in der ODE in einem Deployment-Bundle deployed. Das Deployment-Bundle ist entweder ein Ordner oder eine zip-Datei und enthält den Deployment Descriptor, die BPEL-Datei und alle weiteren Artefakte, wie die WSDLs oder Schemas, die zur Ausführung benötigt werden. Die ODE identifiziert Prozesse ausschließlich am Namen des Deployment-Bundles. Im Deployment Descriptor wird festgelegt, welche Prozesse mit welchen Servicen kommunizieren. Jeder Partner Link, der über eine receive-Aktivität benutzt wird, muss einem provide-Element zugeordnet werden, jeder Partner Link mit einer invoke-Aktivität muss mit einem invoke-Element verbunden werden.

Beim Deployment werden die BPEL-Prozesse in eine für die ODE BPEL Engine Runtime lesbare Form umgewandelt. Dabei wird die Kompatibilität zum ODE Objekt Modell überprüft und in das ODE Objekt Modell überführt. Dieser BPEL-Prozess wird daraufhin als .cbp-Datei abgespeichert. Ab diesem Zeitpunkt wird nur noch auf die .cbp-Datei zugegriffen. Die eigentliche BPEL-Datei wird bei der Ausführung eines Prozesses nicht mehr benötigt. Mit Hilfe von DAOs werden die BPEL-Prozesse, die als ODE Objekt Modell vorliegen, ausgeführt und die Persistenz und Speicherung der Daten sichergestellt.

3.8 Versionierung

In der Apache ODE gibt es eine Versionierung, die im Gegensatz zu einer klassischen Versionierung über alle Deployment-Bundles angewandt wird. Es ist hierbei egal, ob das Deployment-Bundle bereits früher deployed wurde oder ob es sich um ein komplett neues Bundle handelt. Standardmäßig werden Prozessmodelle, wenn von ihnen eine neue Version deployed wird, retired. Prozessmodelle, die retired sind, können nicht mehr instanziiert werden. Die laufenden Instanzen werden ausgeführt bis sie beendet sind.

Beim Deployen eines Bundles werden in Bezug auf die Versionierung folgende Schritte ausgeführt[8]:

1. Eine neue Versionsnummer, die um eins höher ist als die Versionsnummer des vorherigen, wird an das Deployment-Bundle angefügt.
2. Es wird geprüft, ob dasselbe Deployment-Bundle schon einmal deployed wurde. Dies ermittelt die Apache ODE anhand des Namens des Deployment-Bundles. Wenn dies der Fall ist, werden alle alten Deployment-Bundles retired.
3. Die Prozesse werden in der ODE unter derselben Versionsnummer wie das Deployment-Bundle deployed.
4. Daraufhin können die neuen Prozesse gestartet werden.

[8] http://ode.apache.org/process-versioning.html

Die Versionsnummer ist eine einfache ansteigende Nummer. Alle Prozesse innerhalb eines Deployment-Bundles haben die Versionsnummer des Deployment-Bundles.

Beim Deployen eines Deployment-Bundles, das dieselben Prozesse enthält wie ein anderes Bundle, aber mit einem anderen Namen versehen ist, bemerkt die ODE nicht, dass es sich um dieselben Prozesse handelt. In diesem Fall wird nichts retired sondern es gibt zwei identische Prozesse mit unterschiedlichem Namen und unterschiedlicher Versionsnummer. Das Verhalten der ODE ist für diesen Fall nicht genau spezifiziert. Die Frage, welcher der beiden Prozesse die Nachricht bekommt, kann durch unterschiedliche Endpoints geklärt werden.

Prozesse können auch manuell über den entsprechenden Web Service oder die Oberfläche der Apache ODE retired oder wieder aktiviert werden.

4 Konzeption einer Deploy New Version-Strategie

Wissenschaftler gehen beim Erstellen neuer Berechnungen im Bereich von computergestützten Experimenten und Simulationen oft iterativ bzw. experimentell vor. Sie haben bei der Modellerstellung eine grobe Vorstellung von den benötigten Programmen, das genaue Zusammenspiel steht jedoch noch nicht fest. Es entwickelt sich oft erst im Laufe der Berechnung. Je nach deren Verlauf können sich auch die Anforderungen an die zu erstellende Berechnungssoftware zur Laufzeit ändern. Beispielsweise könnte eine andere Visualisierungsmethode nötig werden. Eine iterative Entwicklung von Workflows wird weder von geschäftlichen noch von wissenschaftlichen Workflow-Maschinen zufriedenstellend erfüllt. Insbesondere ist eine konzeptionelle Betrachtung der experimentellen Workflow-Entwicklung nötig. Um diesen Anforderungen gerecht zu werden, wird in dieser Arbeit die Deploy New Version-Funktionalität entwickelt. Ziel der Deploy New Version-Funktionalität ist es, eine Möglichkeit zu schaffen, ein Prozessmodell so verändern zu können, dass eine oder mehrere ausgewählte Prozessinstanzen ein anderes zukünftiges Verhalten verfolgen werden als durch das ursprüngliche Modell vorgegeben. Dazu soll es möglich sein, Instanzen zu pausieren, deren zukünftiges Verhalten zu verändern und sie daraufhin weiterlaufen zu lassen. Instanzen, die ihr Ausführungsende erreicht haben, sollen automatisch am Leben erhalten werden, so dass weitere Funktionalität angehängt werden kann. Mit diesen Funktionalitäten soll verhindert werden, dass Instanzen, an deren Anforderungen sich etwas ändert, neu gestartet werden müssen. Damit können Zeit- und Datenverluste und somit finanzielle Verluste verhindert werden.

Nachfolgend ein Anwendungsbeispiel. Ein Oberarzt soll eine Entscheidung seines Assistenzarztes genehmigen, ist aber der Ansicht, dass zuerst eine weitere Untersuchung zur Bestätigung der Diagnose notwendig ist. Wunsch des Oberarztes ist es, die neue Untersuchung, eine neue Aktivität, in die Instanz einzufügen. Dies wird durch die Deploy New Version-Funktionalität ermöglicht. Ein weiteres Beispiel ist das Ändern der Behandlungsmethoden aufgrund einer Fehldiagnose.

Ein Beispiel aus dem wissenschaftlichen Bereich ist die Strömungssimulation. Bei der Simulation von Meeresströmungen wird anhand der globalen Erwärmung und einigen anderen Faktoren überprüft, wie sich die Strömungen innerhalb der Meere verändern oder ob es zum Versiegen einzelner Ströme kommt. Anhand dieser Ergebnisse kann daraufhin simuliert werden, welche Auswirkungen die Veränderungen der Ströme auf das Klima haben. Die einzelnen Simulationen bauen dabei aufeinander auf. Allerdings hängt der nächste Schritt oft von den Ergebnissen der vorangegangenen Simulation ab. Diese Schritt für Schritt-Entwicklung des Prozessmodells wird durch die Deploy New Version-Funktionalität möglich.

Der Punkt, an dem die Ausführung einer Instanz aktuell ist, wird als Wavefront bezeichnet. Da die Vergangenheit von Instanzen nicht geändert werden kann, ist die Wavefront die Stelle, ab der die Instanz nach dem neuen Modell laufen soll.

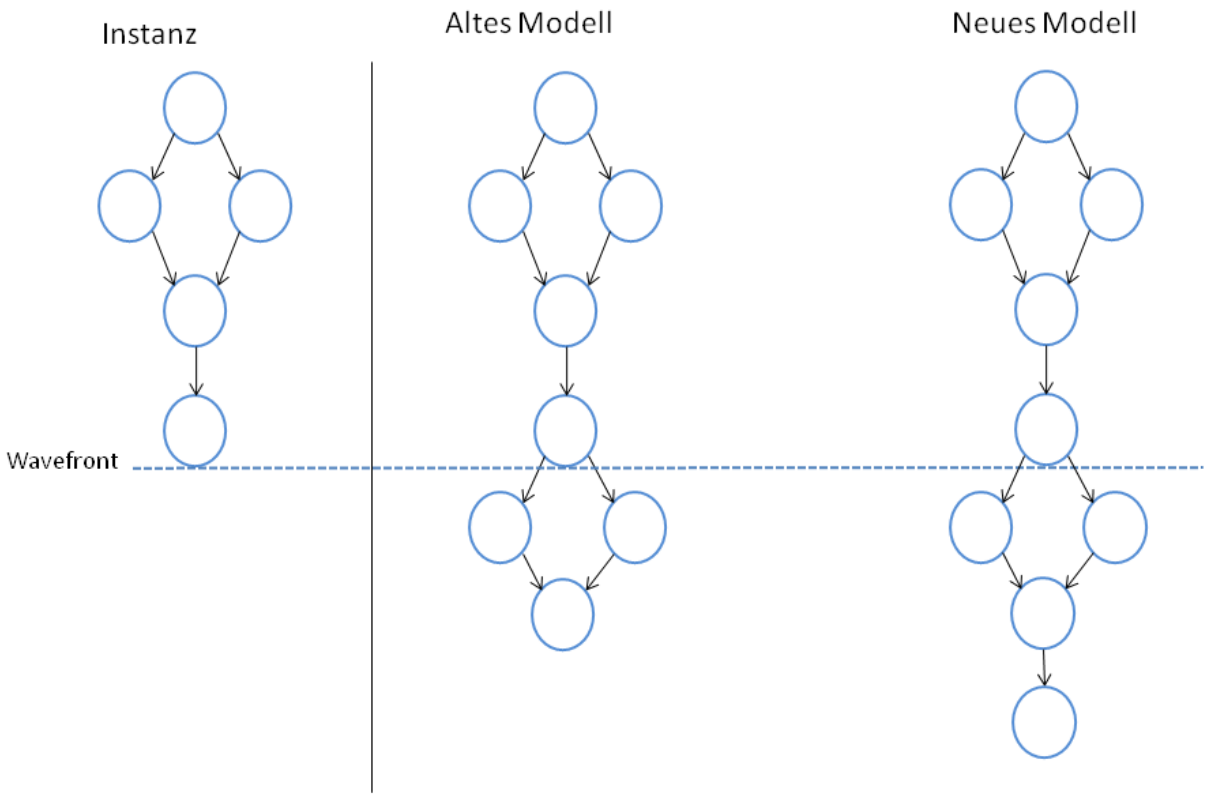


Abbildung 11: Wavefront einer Instanz

Zu Beginn dieser Arbeit gab es zwei Ansätze um die Deploy New Version-Funktionalität umzusetzen:

1. Die pausierte Instanz auf ein neues Prozessmodell umzuziehen.
2. Den Teil des alten Modells ab der Wavefront durch den Teil des neuen Modells ab der Wavefront zu ersetzen.

Die drei Hauptanforderungen an die Deploy New Version-Funktionalität sind:

1. Instanzen sollen nach ihrer erfolgreichen Ausführung auf eine neuere Prozessmodell-Version migrierbar und dann nach dem neuen Modell fortzuführen sein. Dadurch müssen Instanzen, auch wenn sie erfolgreich beendet wurden, am Leben erhalten werden.
2. Instanzen sollen während ihrer Ausführung in den Status SUSPENDED überführbar sein, um sie dann auf eine neue Prozessmodell-Version umzuziehen.
3. Es sollen beide Versionen des Prozessmodells weiterhin aktiv und instanziierbar sein und bei der Migration dürfen keine aktiven Instanzen verloren gehen.

Die dritte Anforderung ist mit dem zweiten Ansatz nicht kompatibel, da bei diesem Ansatz das alte Modell verloren ginge und nur noch das neue Modell existieren würde. Aus diesem Grund wird in dieser Arbeit der erste Ansatz behandelt. Zusätzlich zu den neuen Anforderungen an die Versionierung und das Deployment stehen die Migration von Instanzen und die Veränderung des Lebenszyklus von Instanzen im Vordergrund. In Abbildung 12 ist der konzeptionelle Ansatz der Deploy New Version-Funktionalität graphisch dargestellt. Nachdem eine neue Prozessversion deployed wurde und beide Prozessversionen aktiv sind, wird die Instanz auf die neue Prozessmodell-

Version migriert. Die Instanz kann dann reaktiviert werden und es wird die durch die Migration neu hinzugefügte Logik ausgeführt. Des Weiteren werden Instanzen nach ihrer Ausführung am Leben gehalten, um sie später migrieren zu können und weiter auszuführen.

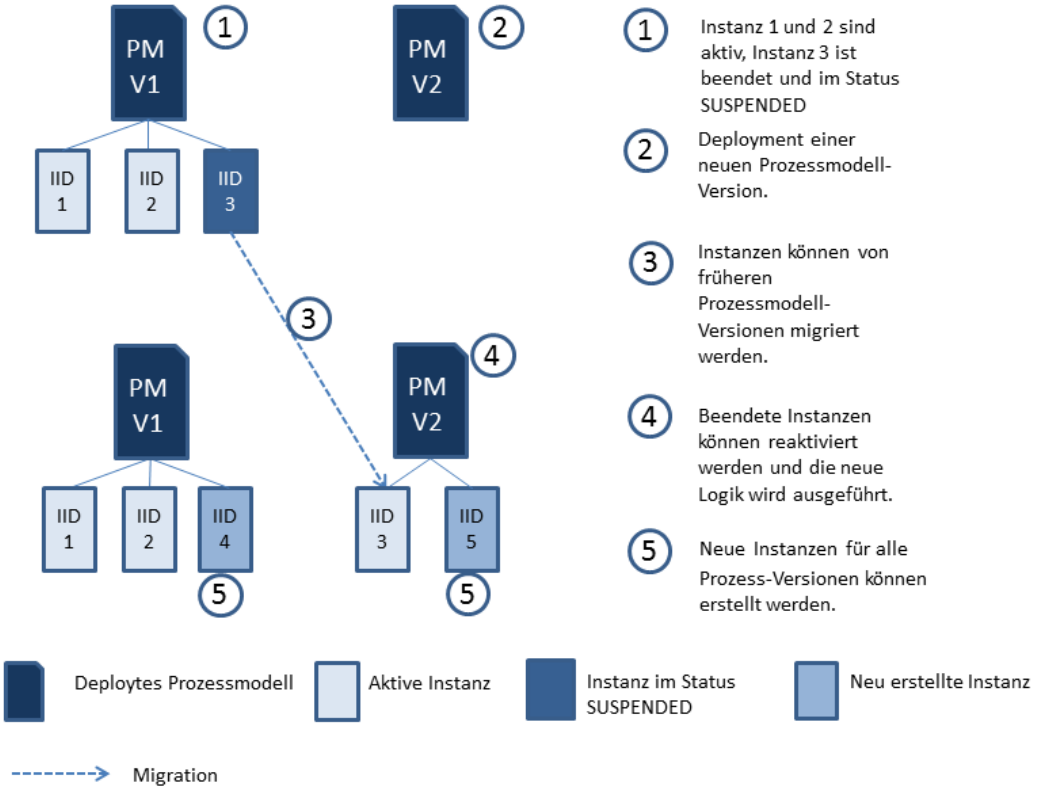


Abbildung 12: Die drei Hauptfunktionalitäten der Deploy New Version-Funktionalität

4.1 State of the art

In diesem Kapitel werden andere Workflow-Maschinen mit ihrem Funktionsumfang vorgestellt. Der Funktionsumfang bezieht sich hier hauptsächlich auf Funktionen, die mit der Deploy New Version-Funktionalität vergleichbar sind.

Alle vorgestellten Workflow-Maschinen erhalten erfolgreich ausgeführte Instanzen nicht am Leben, sondern beenden die Instanzen nach ihrer Ausführung endgültig.

4.1.1 Apache ODE 1.3.4

Seit der Apache ODE Version 1.3.4 existiert die Instance Replayer-Funktionalität[9]. Der Replayer erweitert die Management API der Apache ODE um zwei Operationen:

1. *replay()*
2. *getCommunications()*

replay migriert langlaufende Instanzen auf ein neueres Prozessmodell anhand ihrer Kommunikation. Dazu wird ein zweiter Scheduler, der Replay Scheduler, eingeführt. Auf diesem werden die in der Instanz schon abgearbeiteten Aktivitäten „erneut ausgeführt“. Die noch nicht bearbeiteten

[9] http://ode.apache.org/instance-replayer.html

Aktivitäten werden auf dem Apache ODE Scheduler registriert. „Erneut ausgeführt“ steht in diesem Fall für das Wiederholen der Aktivitäten. Dabei werden die Aktivitäten, die einen Nachrichtenaustausch erzeugen, nicht noch mal ausgeführt, sondern es werden die alten Nachrichten verwendet. Probleme entstehen, wenn die Instanz beispielsweise innerhalb einer *wait*-Aktivität migriert wird, da beim Nachspielen nochmals der komplette Zeitraum gewartet wird. *getCommunications* dient dem Nachbilden von Fehlerszenarien von einer ODE-Instanz zu einer anderen ODE-Instanz, beispielsweise von einem Produktiv- hin zu einem Entwicklungssystem. Dazu wird als erstes mit Hilfe von *getCommunications* der Nachrichtenaustausch der Instanz abgefragt und in ein „Instanz-Kommunikations“-Format gebracht. Damit können die bereits ausgetauschten Nachrichten auf die zweite ODE-Instanz migriert werden, so dass sie dort zur Verfügung stehen. Daraufhin wird *replay* auf der anderen ODE-Instanz ausgeführt, um die Instanz zu replizieren.

Erfolgreich ausgeführte Instanzen sind beendet und können nicht migriert werden. Beim Deployment einer neuen Prozessmodell-Version wird die alte Prozessmodell-Version inaktiv gesetzt. Die Versionierung der Apache Ode ist vom Prozessmodell unabhängig, wie in Kapitel 3.8 ausführlich beschrieben.

4.1.2 Oracle Application Server 10g

Der Oracle BPEL Process Manager[10] unterstützt die Instanzmigration und Versionierung von Prozessmodellen in der Version 10g. Instanzmigration ist nur möglich zwischen Prozessmodellen, die denselben Prozessnamen besitzen, genauer gesagt nur zwischen unterschiedlichen Prozessmodell-Versionen. Es können nur asynchrone Prozesse migriert werden, wobei jedoch einige Regeln zu beachten sind. Die zwei Prozessmodell-Versionen müssen kompatible Interfaces besitzen, die Variablentypen und -namen sowie Partner Link Definitionen müssen kompatibel sein.

4.1.3 Bonitasoft

In der Open Source Workflow-Maschine Bonitasoft[11][12] [16] ist es möglich, Prozess-Versionierung vorzunehmen und mehrere Versionen eines Prozessmodells parallel aktiv zu halten. Dies ist möglich, da die Instanzen manuell in der GUI gestartet werden und dabei eine Prozessmodell-Version ausgesucht wird. BPEL-Prozesse dagegen werden über eine Nachricht gestartet, wodurch instanziierbare Operationen von mehreren aktiven Versionen eines Prozessmodells in Konflikt stehen können. Ein Vorteil von Bonitasoft gegenüber anderen Workflow-Maschinen ist, dass sie eine einfach zu bedienende graphische Oberfläche anbietet, um Workflows in BPMN 2.0, einer graphischen Spezifikationssprache zur Modellierung von Geschäftsprozessen, zu zeichnen. Diese können per Knopfdruck zu einer lauffähigen AJAX-Webanwendung kompiliert werden.

Instanzmigration wird von Bonitasoft in der derzeitig aktuellen Version 5.3 nicht unterstützt.

4.1.4 ADEPTflex

ADEPTflex steht für Application Development Based on Pre-Modeled Templates. In [17] und [18] wird der Funktionsumfang von ADEPTflex beschrieben. Unter anderem werden Ad-hoc–Modifikationen und Schema-Evolution zur Migration von Instanzen unterstützt. Die Migration von Instanzen ist unter

[10] http://www.oracle.com/technetwork/middleware/ias/overview/index.html
[11] http://www.ancud.de/
[12] http://www.bonitasoft.com/

den unterschiedlichen Prozessmodell-Versionen möglich. Über die Funktionsweise der Versionierung von Prozessen in ADEPTflex werden keine genauen Aussagen getroffen.

4.1.5 E-BioFlow

I. Wassink, M. Ooms und P. van der Vet [19] haben einen Ad-Hoc-Editor für die E-BioFlow Workflow-Maschine entwickelt, der den Ansprüchen von explorativen Entwicklungsansätzen von Wissenschaftlern gerecht wird. Bei der E-BioFlow sind das Design und die Ausführung von Workflows im Gegensatz zu anderen Workflow-Maschinen nicht getrennt. Der Ad-Hoc-Editor ermöglicht ein Ad-Hoc-Design von Workflows. Einzelne Aktivitäten oder Gruppen von Aktivitäten können in dem Ad-Hoc-Editor ausgewählt und ausgeführt werden. Dazu erstellt der Editor einen partiellen Workflow aus den ausgewählten Aktivitäten und zusätzlich zwei Aktivitäten, die vom Benutzer bearbeitet werden müssen. Die erste dieser zwei Aktivitäten ist die sogenannte *inputTask,* die zum Starten des Workflow-Fragments benötigt wird. Diese *inputTask*-Aktivität zeigt die schon verfügbaren benötigten und die noch fehlenden Daten an. Der Benutzer trägt die fehlenden Daten ein und kann die bereits verfügbaren Daten ändern. Die zweite Aktivität wird an das Ende des Workflows angefügt und wird als *outputTask* bezeichnet. Diese Aktivität zeigt nach erfolgreicher Ausführung des Workflow-Fragments die Ergebnisdaten an.

Dieser partielle Workflow wird auf der Workflow-Maschine ausgeführt und die Ergebnisse und Zwischenergebnisse im Ad-Hoc-Editor angezeigt. Anhand der einzelnen Ergebnisse der partiellen Workflows kann entschieden werden, wie weiter vorgegangen werden soll. Ebenso ist es möglich, ein Fragment zu korrigieren und es nochmals auszuführen. Nicht möglich ist es jedoch, ein Fragment während der Ausführung anzuhalten und daraufhin zu verändern. Ein Fragment wird immer komplett ausgeführt. Das Debuggen von Aktivitäten oder Fragmenten wird ebenfalls möglich, da man sie einzeln und isoliert ausführen kann. Vor allem das späte Binding von Fragmenten an den Service, das bei der E-BioFlow erst bei der Ausführung zum Tragen kommt, ermöglicht dieses Vorgehen.

E-BioFlow unterstützt die Versionierung von Prozessen. Über die genaue Funktionsweise wird keine Aussage getroffen.

4.1.6 IBM WebSphere Process Server

Der IBM WebSphere Process Server unterstützt in der derzeit aktuellen Version 7 Instanzmigration und Prozess-Versionierung [20]. Prozess-Versionierung bedeutet in diesem Fall, tatsächlich eine neue Version eines Prozessmodells zu deployen. Um die Instanzmigration erfolgreich abzuschließen und das neue Modell als Version des alten Modells zu erkennen, existieren folgende Einschränkungen [21]:

- Keine Namespace-Änderungen oder charakteristischen Änderungen an den implementierten Interfaces von langlaufenden Prozessen.
- Keine Namespace-Änderungen an den Geschäfts-Objekten, die von den langlaufenden Prozessen implementiert werden.
- Keine Änderungen an den Correlation Sets oder den Correlation-Eigenschaften, die benötigt werden.
- Alle Änderungen sollten abwärts kompatibel sein. Nur optionale Attribute sollten zu den Geschäfts-Objekten, die von den Prozessen benötigt werden, hinzugefügt werden

Als neue Version wird nur eine tatsächlich neue Version eines Prozessmodell erkannt, das über den Menüpunkt „New Process Version“ angelegt wird. Es kann immer nur eine Version eines Prozessmodells aktiv sein. Entweder ist die neue Version sofort aktiv und dadurch alle älteren nicht mehr oder die neue Version wird erst in der Zukunft aktiv und bis dahin ist die zuletzt gültige Version aktiv.

4.2 Instanz-Lebenszyklus

Eine Instanz kann sechs Zustände haben. Das Zusammenspiel dieser Zustände ist in Abbildung 13 dargestellt:

- ACTIVE: Die Instanz wird gerade ausgeführt.
- SUSPENDED: Die Instanz wurde „pausiert“.
- COMPLETED: Die Instanz wurde erfolgreich beendet.
- TERMINATED: Die Instanz wurde über die *exit*-Aktivität beendet.
- FAILED: Ein Fehler im globalen Scope ist aufgetreten.
- ERROR: Ein Fehler, der nicht die Ausführung verhindert, aber Beachtung erfordert, ist aufgetreten.

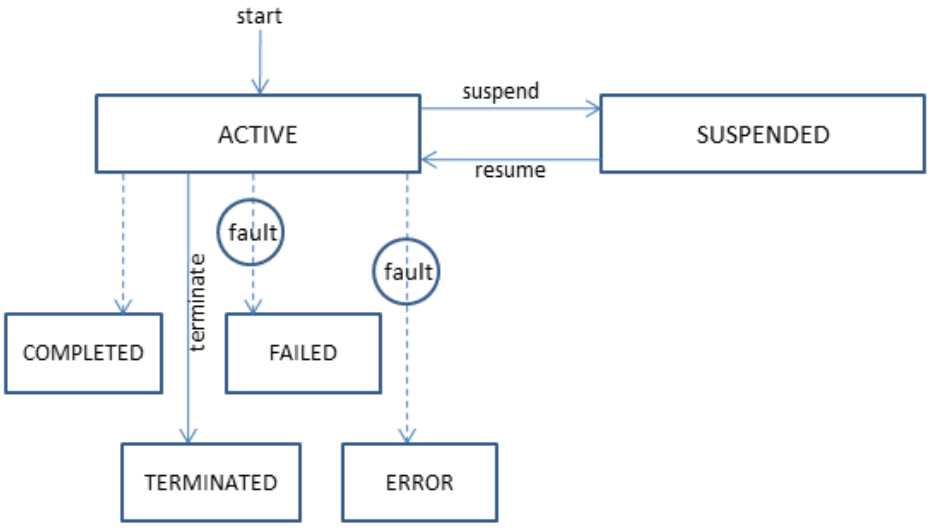


Abbildung 13: Instanz-Lebenszyklus

Eine gestartete Instanz ist im Zustand ACTIVE und geht nach erfolgreicher Ausführung in den Zustand COMPLETED über. Eine aktive Instanz kann über die *suspend*-Funktionalität pausiert werden und befindet sich dann im Status SUSPENDED. Aus dem Status SUSPENDED kann die Instanz über die *resume*-Funktionalität weiter ausgeführt werden. Dadurch wird die Instanz wieder aktiv. Tritt ein Fehler im globalen Scope auf und die Instanz kann nicht beendet werden, geht sie in den Status FAILED über. Wenn ein Fehler auftritt, der die Ausführung der Instanz nicht verhindert, geht die Instanz in den Status ERROR über.

Die drei Funktionalitäten *suspend*, *resume* und *terminate* werden normalerweise als Web Services zur Verfügung gestellt.

Die sechs Zustände bleiben auch bei dem neuen Instanz-Lebenszyklus erhalten, allerdings soll eine Instanz nach erfolgreicher Ausführung nicht mehr automatisch in den Zustand COMPLETED übergehen sondern in den Status SUSPENDED. Dadurch wird die Instanz am Leben erhalten und die Möglichkeit geschaffen, über die *resume*-Funktionalität die Instanz später weiter auszuführen. Um weiterhin die Möglichkeit zu haben, eine Instanz erfolgreich und endgültig zu beenden, soll eine neue Funktion *finish* implementiert werden. Diese *finish*-Funktion soll wie die anderen Funktionen als Web Service zur Verfügung stehen. Der gewünschte Instanz Lebenszyklus ist in Abbildung 14 dargestellt.

Ein Sonderfall, der nicht Gegenstand dieser Arbeit ist, entsteht durch die fehlerhafte Beendigung einer Prozessinstanz. Tritt ein Fehler auf, der die erfolgreiche Ausführung verhindert, so wird die Instanz je nach Fehler in den Status FAILED oder ERROR überführt. Es ist dadurch nicht möglich, weitere Logik hinzuzufügen und die Instanz auf die neue Prozessversion zu migrieren, um sie weiter auszuführen. Zur Behandlung dieses Sonderfalls müsste eine Möglichkeit geschaffen werden, an eine Stelle in der Vergangenheit der Instanz zu springen, quasi ein Verschieben der Wavefront durch das Kompensieren aller Aktivitäten einschließlich der fehlerhaften Aktivität. Daraufhin muss die Instanz in den Zustand SUSPENDED überführt werden, wonach die Instanz migriert und von der Wavefront aus weiter ausgeführt werden könnte.

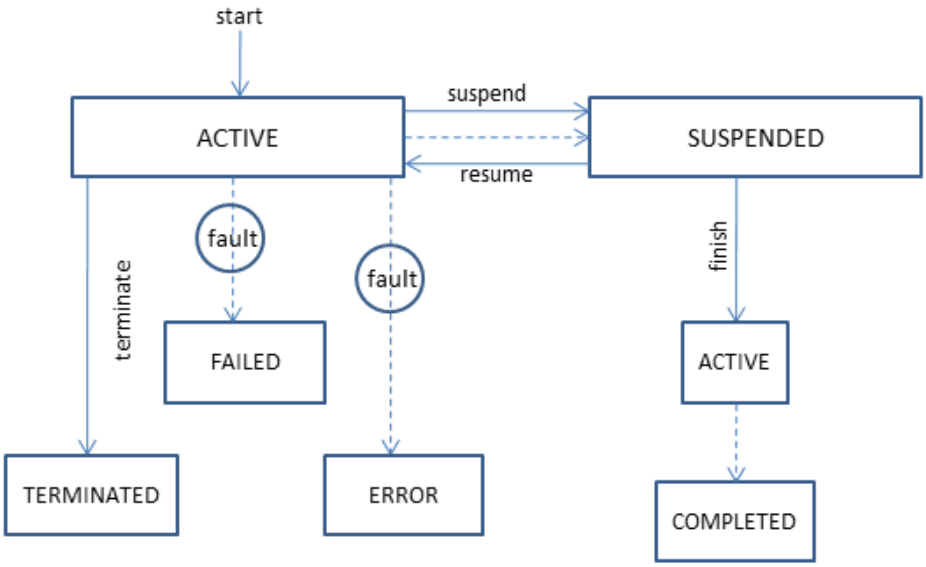


Abbildung 14: gewünschter Instanz Lebenszyklus

4.3 Versionierung und Deployment

Um die Deploy New Version-Funktionalität zu ermöglichen, werden neue Anforderungen an das Deployment und die Versionierung gestellt. Wenn sich die Anforderungen an einen Prozess ändern, wird ein neues Prozessmodell erstellt und deployed. Dieser neue Prozess ist eine neue Version des alten Prozesses. Es sollen weiterhin beide Prozessversionen aktiv bleiben. Aktiv bedeutet, dass beide Prozessversionen weiterhin instanziierbar sein müssen. Da die BPEL-Prozesse als Web Service angeboten werden, kann es hierbei zu Konflikten zwischen den einzelnen Versionen kommen. Wenn nur eine aktive Prozessversion existiert, wird diese über eine von der Workflow-Maschine vergebene

Versionsnummer eindeutig gekennzeichnet. Diese Versionsnummer wird allerdings nicht für den Nachrichtenaustausch verwendet sondern nur innerhalb der Workflow-Maschine. Um einen Prozess zu instanziieren, wird die Versionsnummer nicht benötigt, da es nur einen aktiven Prozess geben kann. Durch die Deploy New Version-Funktionalität ist es jetzt erforderlich, mehrere aktive Prozessversionen parallel aktiv zu halten. Dadurch ist es nicht mehr möglich, eine Instanz einer bestimmten Prozessversion zu erzeugen, da der Client nicht weiß, an welche Prozessversion er die Nachricht zum Instanziieren senden soll. In [22] werden einige Lösungsansätze vorgestellt:

- Um eine Prozessversion zu instanziieren, wird die Prozessversionsnummer mitgesendet. Dadurch wird Kopplung zwischen Client und Workflow-Maschine erhöht, da verschiedene Workflow-Maschinen verschiedene Arten der Versionierung unterstützen.
- Eine zweite Möglichkeit ist der Gebrauch von Metadaten. Zur Designtime des Prozessmodells werden Metadaten spezifiziert und sind Teil des Deployment Bundles. Die Metadaten können beispielsweise durch Schlüsselpaare realisiert werden. Die Workflow-Maschine muss die Eindeutigkeit der Metadaten garantieren. Metadaten adressieren eine Prozessversion, um eine bestimmte Prozessversion zu instanziieren.

In beiden Ansätzen wird ein zusätzlicher Parameter benötigt, damit der Client eine eindeutige Adressierung vornehmen kann. Dieser Parameter kann optional sein, wenn nur eine aktive Version eines Prozesses existiert oder ein Standard-Prozess spezifiziert wurde, der instanziiert wird, wenn kein Parameter angegeben wird. Der spätere Nachrichtenaustausch zwischen Client und Instanz kann weiterhin ohne Parameter stattfinden, da die bekannten Correlation-Mechanismen dafür genutzt werden.

In dieser Arbeit wird ein Konzept vorgestellt, das sich am ersten Lösungsansatz orientiert. Um unterschiedliche Versionen desselben Prozessmodells eindeutig adressieren zu können, wird die Adressierung in Abhängigkeit von der Versionsnummer durchgeführt.

```
<wsdl:service name="HelloService.v2">
        <wsdl:port binding="tns:HelloSoapBinding" name="HelloPort">
        <soap:address location="http://localhost:8080/ode/processes/helloWorld.v2"/>
        </wsdl:port>
</wsdl:service>
```

Listing 9: WSDL Adressierung

In der WSDL-Datei werden der Service-Name und die SOAP address location durch das Anhängen der eindeutigen Versionsnummer identifizierbar. Die SOAP-Nachricht wird, um den BPEL-Prozess zu instanziieren, an die SOAP address location (http://localhost:8080/ode/processes/helloWorld.v2) gesendet.

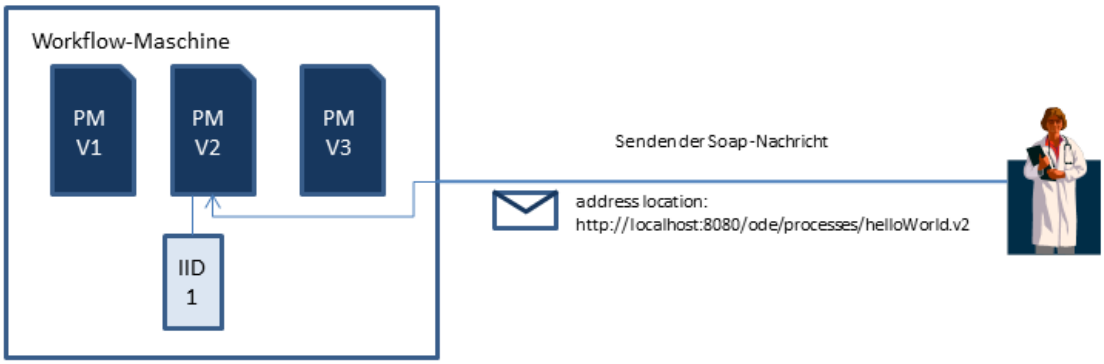


Abbildung 15: Konzept der Adressierung

Wie in Abbildung 15 graphisch dargestellt ist es durch das Senden der Nachricht an die eindeutige SOAP address location möglich, jede Prozessversion zu instanziieren. Dadurch wird gewährleistet, dass mehrere Versionen eines Prozessmodells parallel aktiv und instanziierbar sein können.

4.4 Instanzmigration

Ziel ist es, bei der Migration von Prozessinstanzen, möglichst viele Änderungen an den einzelnen BPEL-Aktivitäten zuzulassen. In diesem Abschnitt werden alle theoretisch möglichen Änderungen an den einzelnen BPEL-Aktivitäten erläutert. Die Aktivitäten werden in zwei Kategorien unterschieden. *Structured Activitites* sind strukturierende Aktivitäten, die wiederum andere Aktivitäten beinhalten. *Basic Activities* sind atomar, sie beinhalten keine weiteren Aktivitäten.

Generell gilt, dass in der Zukunft, also nach der Wavefront liegende Aktivitäten ohne Einschränkungen verändert oder gelöscht werden können. Außerdem können neue Aktivitäten hinzugefügt werden. Da das Konzept so viele Änderungen wie möglich zulassen soll, können auch die Aktivitäten in der Wavefront geändert werden. Unter Umständen können sich diese Änderungen auch auf das zukünftige Verhalten der Instanz auswirken. Änderungen an Aktivitäten, die in der Zukunft liegen sind uneingeschränkt möglich. Im Folgenden werden zu jeder Aktivität die Einschränkungen aufgezeigt, die für Aktivitäten in der Wavefront gelten. Prinzipiell ist es bei Aktivitäten in der Wavefront möglich, ausgehende Links zu verändern, da diese noch nicht evaluiert wurden.

4.4.1 Standard-Elemente und Standard-Attribute

Jede BPEL-Aktivität kann optionale Standard-Elemente und Standard-Attribute enthalten. Es existieren zwei Standard-Attribute:

- *name*=“NCName“?
- *suppressJoinFailure*=“yes|no“?

Das *name*-Attribut dient dazu, maschinenlesbare Aktivitätsnamen zu vergeben. Das *suppressJoinFailure*-Attribut gibt an, ob Join-Fehler unterdrückt werden sollen. Ein Join-Fehler tritt auf, wenn die Evaluierung eines Links fehlschlägt. Änderungen an diesen Attributen stellen im Großen und Ganzen bei laufenden Aktivitäten kein Problem dar. Beim Ändern des *name*-Attributs muss beachtet werden, dass die Aktivität dadurch unter Umständen nicht mehr gefunden werden

kann. Beispielsweise werden bei der Kompensierung von Scopes die Namen der Scopes für die Referenzierung genutzt.

Die Standard Elemente sind zwei Container. Zum einen der <*sources*>- und zum anderen der <*targets*>-Container. In diesen Containern sind Links enthalten. Beim Ändern der Container muss beachtet werden, dass die Änderungen im kompletten Prozessmodell konsistent umgesetzt werden. Beispielsweise müssen die Links angepasst werden, wenn sich der <*sources*>- oder <*target*>-Container ändert. Wenn sich der Link ändert, müssen <*sources*>-und <*targets*>-Container angepasst werden. Unter dieser Bedingung sind Änderungen an den Containern unproblematisch.

4.4.2 Process

Das process-Element stellt das äußerste Element eines BPEL-Prozesses dar. Es wird nicht als BPEL-Aktivität angesehen. Innerhalb des process-Elements werden folgende Elemente definiert:

- *queryLanguage*: Definiert die Sprache, um innerhalb von Zuweisungen Knoten auszuwählen. Standardmäßig ist die Sprache XPath.
- *expressionLanguage*: Definiert die Sprache innerhalb des process-Elements. Der Standard-Wert ist XPath.
- *suppressJoinFailure*: Definiert, ob Join-Fehler unterdrückt werden oder von einem *fault-handler* bearbeitet werden sollen. Ist das Attribut in einer untergeordneten Aktivität nicht definiert, so wird der Wert aus dem process-Element vererbt. Standardmäßig ist der Wert des Attributs *no*.
- *exitOnStandardFault*: Der Standard-Wert ist *no*. Wenn das Attribut den Wert *yes* hat, muss der Process beim Auftreten eines Fehlers sofort beendet werden. Wenn der Wert *no* ist, werden auftretende Fehler durch einen *fault-handler* behandelt. Ebenso wie das *suppressJoinFailure*-Attribut wird es vererbt.
- *import*: Beschreibt eine Abhängigkeit zu externen XML-Schemas oder WSDL-Definitionen.
- *partnerLinks*: Definiert die benötigten Partner Links.
- *messageExchanges*: Wird benötigt, um die Zuordnung zwischen eingehenden Nachrichten und den reply-Aktivitäten eindeutig zu machen.
- *variables*: Definiert alle innerhalb des Prozesses benötigten Variablen.
- *correlationSets*: Ermöglicht dem Prozess während der Ausführung, Nachrichten zu empfangen und der korrekten Prozessinstanz zuzuordnen. Dazu werden bestimmte Daten aus der empfangenen Nachricht (sogenannte Properties) genutzt.
- *faultHandler:* Definiert die Fehlerbehandlung.
- *eventHandler*: Definiert einen *event-handler*, der aufgerufen wird, wenn das definierte Event auftritt. Das Event kann entweder ein wirkliches Event sein und wird dann durch das *onEvent*-Attribut angegeben. Es kann aber auch ein Timer-Event sein, das durch das *onAlarm*-Attribut beschrieben wird.

Jeder Prozess beinhaltet mindestens eine Aktivität. Diese Aktivität kann entweder eine *Structured* oder *Basic Activity* sein.

Aktivitäten, die sich gerade in der Ausführung befinden, können mit Einschränkungen verändert oder gelöscht werden. Diese Einschränkungen sind von der jeweiligen Aktivität abhängig. Die einzelnen Aktivitäten, ihre Einschränkungen und Probleme werden im weiteren Verlauf dieses Kapitels

beschrieben. Aktivitäten, die nach der Wavefront, also in der Zukunft liegen, können problemlos geändert werden.

Das Ändern der Attribute ist nur teilweise möglich. Das *queryLanguage*- und das *expressionLanguage*-Attribut können nur für noch nicht ausgewertete Ausdrücke geändert werden. Das Correlation Set darf geändert werden so lange alle dazugehörigen Correlations noch nicht initialisiert sind. Der *fault-handler* und der *event-handler* können verändert, gelöscht und hinzugefügt werden, wenn sie sich gerade nicht in der Ausführung befinden. Beim Hinzufügen eines *fault*- oder *event-handlers* ist zu beachten, dass der Process Scope dann neu initialisiert werden muss. Das *messageExchange*-Element darf nur dann verändert werden, wenn die Änderungen keinen Nachrichtenaustausch betreffen, der schon begonnen hat. Die *suppressJoinFailure*-, *exitOnStandardFault*-Attribute und das *partnerLinks*-Element können problemlos geändert werden. Die innerhalb des *variables*-Attributs definierten Variablen können gelöscht und verändert werden, wenn sie noch nicht initialisiert wurden, d.h. noch nicht benutzt wurden. Dabei muss sichergestellt werden, dass alle innerhalb des Prozesses verwendeten Variablen definiert sind. Es können auch neue Variablen hinzugefügt werden. Für Partner Links gelten dieselben Regeln wie für Variablen. Beim Verändern des *import*-Attributs muss darauf geachtet werden, dass die zu importierenden Schemas und WSDLs an den jeweiligen Stellen vorhanden sind.

Um diese Änderungen zu ermöglichen, müssen alle Attribute nachgeladen werden bevor die migrierte Instanz weiter ausgeführt wird. Das Nachladen der Änderungen wird über sogenannte Change Operations realisiert [22].

4.4.3 Basic Activities

Basic Activities beschreiben einen einzelnen Ausführungsschritt des Prozessmodells. Sie können, wenn sie sich gerade in der Ausführung befinden, nur bedingt verändert werden. Dies gilt nicht für Standard Elemente und Attribute. Genauer gesagt können nur die ausgehenden Links angepasst werden, da diese noch nicht evaluiert wurden. Warum *Basic Activities* nicht verändert werden können, werden anhand der invoke- und der receive-Aktivität beispielhaft erläutert.

invoke-Aktivität

Die invoke-Aktivität ruft von Service Providern angebotene Web Services auf. Es können *inputVariable* und *outputVariable* angegeben werden Die invoke-Aktivität kann weitere Aktivitäten innerhalb eines *compensation*- oder *fault-handlers* beinhalten. Es existieren zwei verschiede invoke Arten: *request-response* und *one-way*. Bei einem synchronem invoke, also *request-response*, werden *inputVariable* und *outputVariable* benötigt, da auf die Antwort gewartet wird und die invoke-Aktivität gleichzeitig den Rücksprung-Punkt darstellt.
Bei einem *one-way*-invoke wird nur die inputVariable benötigt, da es sich um einen asynchronen Aufruf handelt.

In beiden Fällen sind Änderungen an der aktiven invoke-Aktivität nicht möglich. Bei einem *one-way*-invoke wird sofort beim Betreten der Aktivität der Web Service aufgerufen, die invoke-Aktivität ist daraufhin beendet. Bei einem *request-response*-invoke wird ebenfalls wie beim *one-way*-invoke der Web Service aufgerufen. Die invoke-Aktivität bleibt so lange aktiv, bis die Antwort-Nachricht eintrifft.

Änderungen sind nicht erlaubt, da durch Änderungen an der output-Variablen die Antwort-Nachricht unter Umständen nicht mehr zugeordnet werden kann.

receive-Aktivität

Die receive-Aktivität empfängt eingehende Nachrichten. Die Start-Aktivität eines Prozessmodells ist entweder eine receive-Aktivität oder eine pick-Aktivität mit dem Attribut *createInstance= yes.* Innerhalb der receive-Aktivität werden Partner Links, optional Port Types und Operationen definiert. Wenn das receive die Startaktivität ist, wird nach dem Eintreffen der Nachricht die receive-Aktivität aktiv und instanziiert sofort das Prozessmodell. Wenn die receive-Aktivität keine Start-Aktivität ist, wird der Inhalt der Nachricht sofort in die davor vorgesehene Variable kopiert. In beiden Fällen kann nichts an der aktiven Aktivität geändert werden, da diese sofort ausgeführt sind.

Alle *Basic Activities* führen sobald sie aktiv sind sofort ihre Logik aus. Danach sind sie entweder sofort beendet oder warten auf eine Antwort. In beiden Fällen kann nichts an ihnen geändert werden, da sie entweder schon zur Vergangenheit gehören oder aber die Antwort-Nachricht nicht mehr zugeordnet werden kann.

Es existieren Ausnahmen, wenn sich eine *Basic Activity* innerhalb einer *Structured Activity* befindet. Diese werden bei den einzelnen *Structured Activities* behandelt. Eine weitere Ausnahme ist die wait-Aktivität. Diese *Basic Activity* darf wie folgt verändert werden, wenn sie sich gerade in der Ausführung befindet:

wait-Aktivität

Die wait-Aktivität (Listing 10) pausiert die Ausführung für eine bestimmte Zeitspanne oder bis zu einem bestimmten Zeitpunkt.

```
<wait standard-attributes>
        standard-elements
        (
        <for expressionLanguage="anyURI"?>duration-expr</for>
        |
        <until expressionLanguage="anyURI"?>deadline-expr</until>
        )
</wait>
```

Listing 10: wait-Aktivität [14]

Die wait-Aktivität ist die einzige *Basic Activity*, bei der es möglich ist, den Inhalt der Aktivität zu verändern. Wenn beispielsweise eine wait-Aktivität existiert, die drei Jahre warten soll und nach einem Jahr dann festgestellt wird, dass genug Zeit vergangen ist und der Prozess weiterlaufen soll, muss die wait-Aktivität verändert werden können. Es wird die Dauer der wait-Aktivität in einer neuen Prozessversion auf ein Jahr geändert und die Instanz migriert. Daraufhin wird die Instanz weiter ausgeführt. Die wait-Aktivität wird beendet, da ein Jahr bereits vergangen ist und die Instanz weiter ausgeführt. Dies funktioniert problemlos, da der Zeitpunkt an dem die wait-Aktivität gestartet wurde, bekannt ist. Ist die zu wartende Zeitspanne noch nicht verstrichen, bleibt die wait-Aktivität solange aktiv, bis die Zeitspanne verstrichen ist. Ist innerhalb der wait-Aktivität eine until-Bedingung angegeben, funktioniert es nach demselben Prinzip. Liegt die until-Bedingung in der Vergangenheit,

wenn die Instanz weiter ausgeführt wird, wird die Aktivität beendet. Ansonsten wird bis zum vorgegebenen Zeitpunkt gewartet.

4.4.4 Structured Activities

Structured Activities stellen den Kontrollfluss eines Prozessmodells dar. Sie können rekursiv weitere *Basic* und *Structured Activities* beinhalten. Im weiteren Verlauf sind die einzelnen Aktivitäten mit ihren möglichen Änderungen und Einschränkungen aufgeführt.

scope-Aktivität

Eine scope-Aktivität (Listing 11) bietet die Möglichkeit das Ausführungs-Verhalten der beinhalteten Aktivitäten zu bestimmen. Dazu werden Variablen, Partner Links, der Nachrichtenaustausch (*message exchange*), Correlation Sets, *event-*, *fault-*, *compensation-* und *termination-handler* innerhalb des Scopes definiert. Der Kontext der scope-Aktivitäten kann hierarchisch verschachtelt sein, der "Wurzel"-Kontext wird innerhalb des process-Elementes definiert.

Obwohl das process-Element und die scope-Aktivität in ihrem Aufbau sehr ähnlich sind, gibt es doch Unterschiede:

- Das process-Element stellt keine Aktivität im eigentlichen Sinne dar, deshalb können die Standard-Attribute und Elemente nicht für das process-Element verwendet werden.
- Ein c*ompensation-* und t*ermination-handler* können nicht an das process-Element angehängt werden.
- Das *isolated*-Attribut, das die Kontrolle von Datenzugriffen bei paralleler Ausführung zweier Scopes steuert, existiert nicht für das process-Element.

Jede scope-Aktivität benötigt eine so genannte „*primary activity*“, die das Standard-Verhalten des Scopes definiert. Sie kann eine *Structured* oder *Basic Activity* darstellen. Alle anderen Konstrukte einer scope-Aktivität sind optional. Das innerhalb der scope-Aktivität definierte Verhalten gilt für alle innerhalb des Scopes vernetzten Aktivitäten.

```
<scope isolated="yes|no"? exitOnStandardFault="yes|no"?
        standard-attributes>
        standard-elements
        <variables>?
                ...
        </variables>
        <partnerLinks>
                ...
        </partnerLinks>
        <messageExchanges>
                ...
        </messageExchanges>
        <correlationSets>
                ...
        </correlationSets>
        <eventHandlers>
                ...
```

```
        </eventHandlers>
        <faultHandlers>
                ...
        </faultHandlers>
        <compensationHandler>
                ...
        </compensationHandler>
        <terminationHandler>
                ...
        </terminationHandler>
        activity
</scope>
```

Listing 11: Scope-Aktivität [14]

Beim Ändern des *variables*-, *partnerLinks*-, *correlationSets*- *messageExchanges*-Elements oder des *exitOnStandardFault*-Attributs gelten die in Kapitel 4.4.2 beschriebenen Einschränkungen. Es können keine zwei ineinander verschachtelten Scopes, die beide das Attribut *isolated* mit dem Wert *yes* besitzen, parallel ausgeführt werden. Deshalb darf das *isolated*-Attribut nur dann von *yes* auf *no* geändert werden, wenn weder der Eltern-Scope noch ein Kind-Scope *isolated* sind. Das Ändern des *isolated*-Attributs von *yes* auf *no* ist immer möglich. Sämtliche Handler können verändert und gelöscht werden, wenn sie sich gerade nicht in der Ausführung befinden. Das Hinzufügen der unterschiedlichen Handler ist problemlos möglich. Wie auch beim process-Element muss nach Änderungen an einem Scope eine Re-Initialisierung erfolgen, bevor die migrierte Instanz weiter ausgeführt wird.

sequence-Aktivität

Die sequence-Aktivität (Listing 12) beinhaltet eine oder mehrere Aktivitäten, die sequentiell abgearbeitet werden. Sie ist beendet, wenn die letzte Aktivität in der Sequenz abgearbeitet wurde.

```
<sequence standard-attributes>
        standard-elements
        activity+
</sequence>
```

Listing 12: sequence-Aktivität [14]

Änderungen innerhalb der sequence-Aktivität können das Hinzufügen, Ändern und Löschen von noch nicht ausgeführten Aktivitäten sein. Des Weiteren können Links verändert, hinzugefügt oder gelöscht werden. Bezieht sich die Änderung auf die gerade in der Wavefront liegende Aktivität, gelten bei *Basic Activitities* folgende Einschränkungen:

- Der Inhalt der Aktivität darf nicht verändert werden.
- Die eingehenden Links können nicht verändert werden

Bei *Structured Activities* gelten die Einschränkungen der jeweiligen Aktivität. Ausgehende Links können bei *Basic* und *Structured Activities* angepasst werden. Nach dem Ändern der Sequence-Aktivität muss die Liste von auszuführenden Aktivitäten in der Sequence-Aktivitätsinstanz aktualisiert werden, wenn die Ausführung der migrierten Prozessinstanz wieder aufgenommen wird.

flow-Aktivität

Die flow-Aktivität (Listing 13) ermöglicht die nebenläufige und synchronisierte Ausführung von Aktivitäten. Ein Flow ist beendet, wenn alle beinhalteten Aktivitäten beendet sind. Aktivitäten, die über Bedingungen gesteuert werden, werden auch als beendet angesehen, wenn ihre Bedingung false ist und sie dadurch nie ausgeführt werden. Die Aktivitäten innerhalb eines Flows können durch Links miteinander verbunden sein und beliebig tief verschachtelt werden. Im *links*-Element sind alle Synchronisations-Abhängigkeiten zwischen den Aktivitäten des Flows definiert.

```
<flow standard-attributes>
        standard-elements
        <links>?
                <link name="NCName">+
        </links>
        activity+
</flow>
```

Listing 13: flow-Aktivität [14]

Das Hinzufügen von Aktivitäten innerhalb eines Flows ist problemlos möglich. Beim Löschen von Aktivitäten innerhalb eines Flows muss darauf geachtet werden, dass die Verlinkungen zwischen den einzelnen Aktivitäten nicht zerstört werden. Beim Weiterlaufen der Prozessinstanz müssen eingefügte Aktivitäten, die keine eingehenden Links besitzen, direkt gestartet werden. Das Löschen und Ändern von Aktivitäten ist für bereits beendete Aktivitäten nicht möglich. Innerhalb des Flows können Aktuell laufende *Structured* oder *Basic Activities* unter Beachtung der jeweiligen Einschränkungen verändert werden. Das link-Attribut kann umbenannt werden. Allerdings muss beachtet werden, dass die Referenzen auf die umbenannten Links ebenfalls geändert werden.

Eine Ausnahme innerhalb eines Flows ist es, dass ausgehende Links an bereits beendeten Aktivitäten verändert werden können. Innerhalb eines aktiven Flows sind unter Umständen schon einige Aktivitäten beendet und andere noch aktiv. Da der Flow noch aktiv ist, ist es möglich, eine neue Aktivität in den Flow einzufügen und an eine bereits abgeschlossene Aktivität zu verlinken. Wenn der Link keine Transition Condition besitzt, ist dies problemlos möglich. Wenn eine Transition Condition verwendet wird, um die Aktivität zu starten, muss sichergestellt werden, dass die Daten, die zur Auswertung der Transition Condition benötigt werden, vorliegen. Da die vorhergegangene Aktivität bereits beendet ist, wäre eine Möglichkeit, um die Daten nach einer Migration weiterhin zur Verfügung zu haben, die Daten durch Snapshots vorzuhalten.

while-Aktivität

Die while-Aktivität (Listing 14) besteht aus einer booleschen Bedingung und einer Aktivität. Diese Aktivität kann eine strukturierende Aktivität sein, die wiederum weitere Aktivitäten beinhaltet. Die Aktivität wird so oft hintereinander ausgeführt, wie die Bedingung gültig ist.

```
<while standard-attributes>
        standard-elements
        <condition expressionLanguage="anyURI"?>bool-expr</condition>
        activity
</while>
```

Listing 14: while-Aktivität [14]

Befindet sich die while-Aktivität gerade in der Wavefront, können ausgehende Links sowie die boolesche Bedingung nach Belieben, also unabhängig von den jeweiligen Einschränkungen, verändert werden. Die beinhaltete Aktivität kann im aktiven Zustand mit den jeweiligen Einschränkungen oder vor und nach jeder Iteration beliebig geändert werden. Es gilt zu beachten, dass Änderungen an der laufenden while-Aktivität oder einer beinhalteten *Basic Activity* erst bei der darauffolgenden Iteration greifen. Aus diesem Grund kann auch eine beinhaltete *Basic Activity* nach Belieben verändert werden. Bei späterer Betrachtung des Prozessmodells können diese Änderungen nicht mehr im Detail nachvollzogen werden.

if-Aktivität

Die if-Aktivität (Listing 15) besteht aus einer Liste von einem oder mehreren elseif- oder else-Zweigen. Die einzelnen Zweige werden in der Reihenfolge, wie sie angeordnet sind, betrachtet. Ein Zweig kann genau eine Aktivität beinhalten, die wiederum weitere Aktivitäten beinhalten kann. Der erste Zweig, dessen Bedingung wahr ist, wird ausgeführt. Wenn keine Bedingung wahr ist, wird der else-Zweig ausgeführt. Die if-Aktivität ist beendet, wenn die Aktivität des auszuführenden Zweiges beendet ist oder wenn keine Bedingung wahr ist und kein else-Zweig existiert.

```
<if standard-attributes>
        standard-elements
        <condition expressionLanguage="anyURI"?>bool-expr</condition>
        activity
        <elseif>*
                <condition expressionLanguage="anyURI"?>bool-expr</condition>
                activity
        </elseif>
        <else>?
                activity
        </else>
</if>
```

Listing 15: if-Aktivität [14]

Bei einer laufenden if-Aktivität können die beinhalteten Aktivitäten geändert werden, wenn sie noch nicht ausgeführt werden. Das bedeutet, dass alle inaktiven Zweige geändert werden können. Sobald die beinhaltete Aktivität läuft, kann die beinhaltete Aktivität entsprechend ihrer Einschränkungen verändert werden. Die Bedingungen der if-Aktivität dürfen verändert werden. Änderungen an den

nicht gewählten Zweigen, an allen Bedingungen und den beinhalteten *Basic Activities*, auch des gewählten Zweiges, sind nur dann sinnvoll, wenn die if-Aktivität sich innerhalb einer Schleife befindet. Die Änderungen an der Bedingung werden ab der nächsten Auswertung der Bedingung, also der nächsten Iteration der Schleife, gültig.

pick-Aktivität

Die pick-Aktivität (Listing 16) wartet darauf, dass ein bestimmtes Event aus einer Liste von Events auftritt. Wenn dies geschieht, wird die mit diesem Event verknüpfte Aktivität ausgeführt. Nach dem Auftreten eines Events werden die anderen Events von dieser pick-Aktivität nicht mehr betrachtet. Pick besteht aus mehreren Zweigen und jeder Zweig enthält ein Event-Aktivitäts-Paar. Die pick-Aktivität ist beendet, wenn die Aktivität des gewählten Zweiges beendet ist. Es gibt zwei Möglichkeiten, ein Event zu definieren:

- <*onMessage*>: verhält sich ähnlich wie die receive-Aktivität. Es wartet auf eine eingehende Nachricht.
- <*onAlarm*>: ist zeitgesteuert. Wenn die definierte Dauer innerhalb des <for>-Elementes null oder negativ oder der im <until>-Element definierte Zeitpunkt erreicht ist, wird das Event ausgeführt.

Jede pick-Aktivität muss mindestens ein <*onMessage*>-Event enthalten.

```
<pick createInstance="yes|no"? standard-attributes>
        standard-elements
        <onMessage partnerLink="NCName"
                portType="QName"?
                operation="NCName"
                variable="BPELVariableName"
                messageExchange="NCName">+
                        <correlations>?
                                <correlation set="NCName" initiate="yes|join|no" />+
                        </correlations>
                        <fromParts>
                                <fromPart part="NCName" toVariable="BPELVariableName" />+
                        </fromParts>
                        activity
        </onMessage>
        <onAlarm>*
                (
                <for expressionLanguage="anyURI"?>duration-expr</for>
                |
                <until expressionLanguage="anyURI"?>deadline-expr</until>
                )
                activity
        </onAlarm>
</pick>
```

Listing 16: pick-Aktivität [14]

Die *onMessage*- und *onAlarm*-Attribute können geändert, gelöscht oder hinzugefügt werden. Beim Löschen oder Ändern gibt es die Einschränkung, dass ein bereits aktiver Zweig nicht mehr gelöscht/geändert werden kann. Außerdem gilt zu beachten, dass mindestens ein *onMessage*-Event vorhanden sein muss. Zusätzlich muss nach der Migration einer Instanz überprüft werden, ob eine der Nachrichten, auf die gewartet wird, unter Umständen schon eingetroffen ist. Ist dies der Fall, wird der dementsprechende pick-Zweig ausgeführt. Für den Fall, dass mehrere passende Nachrichten nach der Migration vorhanden sind, muss eine Bedingung definiert werden, anhand der entschieden wird, welcher pick-Zweig ausgeführt wird. Eine Möglichkeit wäre es, anhand des Eingangszeitpunkts der Nachrichten zu entscheiden. Der Zweig, dessen Nachricht als erstes eintraf, wird ausgeführt. Ein zweiter denkbarer Ansatz wäre es, die Zweige unterschiedlich zu gewichten und wenn mehrere Nachrichten bereits eingetroffen sind, den Zweig mit der höchsten Gewichtung auszuführen. Dasselbe gilt für das *onAlarm*-Event. Das Correlation Set darf geändert werden, so lange alle dazugehörigen Correlations noch nicht initialisiert sind. Das *createInstance*-Attribut darf aus folgenden Gründen nicht geändert werden. Die Instanz ist instanziiert sobald die pick-Aktivität aktiv wird, wenn *createInstance* den Wert *yes* hatte. Wenn *createInstance no* ist, wurde die Instanz bereits über eine andere pick-Aktivität oder eine receive-Aktivität instanziiert. Alle anderen Attribute innerhalb der pick-Aktivität können problemlos geändert werden.

forEach-Aktivität

Die forEach-Aktivität (Listing 17) führt den beinhalteten Scope N+1 mal aus. N entspricht dem Wert des *finalCounterValue*-Elements. Beim Starten der forEach-Aktivität werden die Ausdrücke im *startCounterValue*- und *finalCounterValue*-Element evaluiert. Die Ausdrücke bleiben während der Ausführung der forEach-Aktivität konstant und gültig. Falls der Startwert größer als der Endwert ist, wird die forEach-Aktivität nicht ausgeführt und ist beendet. Die Kind-Aktivität einer forEach-Aktivität muss eine scope-Aktivität sein.

Eine forEach-Aktivität kann entweder parallel oder seriell sein. Dies wird über das *parallel*-Attribut festgelegt. Bei der parallelen forEach-Aktivität werden beim Start der Aktivität N+1-Instanzen des beinhalteten Scope erstellt und parallel ausgeführt. Bei der seriellen forEach-Aktivität wird eine Instanz des beinhalteten Scopes nach der anderen ausgeführt. Die Anzahl der auszuführenden Scope-Aktivitäten kann mit der *completionCondition* auf den in der *completionCondition* spezifizierten Wert begrenzt werden. Durch das *successfulBranchesOnly*-Attribut werden entweder nur erfolgreich ausgeführte Scopes gezählt (*yes*) oder alle beendeten Scopes (*no*). Wenn die *completionCondition* erfüllt ist, werden im seriellen Fall keine weiteren Iterationen mehr durchgeführt. Im parallelen Fall werden noch nicht beendete Scope-Aktivitäten terminiert, wenn die *completionCondition* erfüllt ist. Die *completionCondition* ist optional.

```
<forEach counterName="BPELVariableName" parallel="yes|no"
        standard-attributes>
        standard-elements
        <startCounterValue expressionLanguage="anyURI"?>
                unsigned-integer-expression
        </startCounterValue>
        <finalCounterValue expressionLanguage="anyURI"?>
                unsigned-integer-expression
        </finalCounterValue>
```

```
<completionCondition>?
    <branches expressionLanguage="anyURI"?
        successfulBranchesOnly="yes|no">?
        unsigned-integer-expression
    </branches>
</completionCondition>
<scope ...>...</scope>
</forEach>
```

Listing 17: forEach-Aktivität [14]

Bei der forEach-Aktivität müssen zwei Fälle, die serielle und die parallele forEach-Aktivität, betrachtet werden. In beiden Fällen kann der *startCounterValue* nicht geändert werden, da er schon evaluiert wurde. Bei der parallelen Ausführung müssen alle Änderungen am Scope oder seinen Kindelementen an alle N+1 Scope-Instanzen propagiert werden. Die erlaubten Änderungen hängen von den jeweiligen Aktivitäten ab. Die *completionCondition* kann geändert werden. Es muss aber nach der Änderung der *completionCondition* überprüft werden, ob diese direkt beendet werden muss, da unter Umständen die Bedingung schon in der Vergangenheit erfüllt war oder gerade erfüllt ist. Das *finalCounterValue*-Attribut kann vergrößert, jedoch nicht verkleinert werden. Wenn es vergrößert wird, müssen bei der parallelen Ausführung nachträglich Scope-Instanzen gestartet werden.

Bei der seriellen forEach-Aktivität können beinhaltete Aktivitäten beliebig verändert, hinzugefügt und gelöscht werden. Wenn Aktivitäten in die Zukunft hinzugefügt oder aus der Zukunft des Scopes gelöscht werden, werden die Änderungen sofort gültig. Aktivitäten, die in die Vergangenheit des Scopes eingefügt oder aus der Vergangenheit gelöscht werden, werden ab der nächsten Iteration beachtet. Der *finalCounterValue* kann auch problemlos geändert werden, so lange der neue Wert größer ist als die derzeitige Iteration. Veränderungen an der completionCondition, dem *finalCounterValue* und den beinhalteten *Basic Activities* sind erst bei der darauffolgenden Iteration gültig. Veränderungen an den *Structured Activities* hängen von den Einschränkungen der jeweiligen Aktivität ab und sind zum Teil sofort gültig.

repeatUntil-Aktivität

Die repeatUntil-Aktivität (Listing 18) besteht wie die while-Aktivität aus einer booleschen Bedingung und einer Aktivität. Die Aktivität wird so lange ausgeführt bis die boolesche Bedingung wahr ist. Im Gegensatz zur while-Aktivität wird die repeatUntil-Aktivität mindestens einmal ausgeführt.

```
<repeatUntil standard-attributes>
    standard-elements
    activity
    <condition expressionLanguage="anyURI"?>bool-expr</condition>
</repeatUntil>
```

Listing 18: repeatUntil-Aktivität [14]

Befindet sich die RepeatUntil-Schleife gerade in der Ausführung, kann die Bedingung der RepeatUntil-Schleife nach Belieben verändert werden, da die veränderte Bedingung erst nach der laufenden Iteration neu ausgewertet wird. Beinhaltete Aktivitäten dürfen beliebig verändert werden, auch wenn sie gerade ausgeführt werden, da die Änderungen erst für die nächste Iteration der repeatUntil-Aktivität gültig sind. Beinhaltete strukturierende Aktivitäten, die gerade ausgeführt

werden, können zusätzlich anhand ihrer Einschränkungen verändert werden. Diese Änderungen sind sofort gültig. Bei späterer Betrachtung des Prozessmodells können all diese Änderungen nicht mehr im Detail nachvollzogen werden.

5 Prototypische Umsetzung der Strategie

In diesem Kapitel wird die prototypische Umsetzung der Deploy New Version-Funktionalität erläutert. Der Prototyp besitzt die Einschränkung, dass der Prozess von einer Sequence-Aktivität als umschließendes Element umgeben sein muss. Beginnt der Prozess mit einem anderen Element, beispielsweise einer Flow-Aktivität, wird die Deploy New Version-Funktionalität derzeit nicht unterstützt.

Innerhalb des Deployment Web Service wurde eine neue Funktion implementiert, die den Deploy New Version-Mechanismus, wie in Kapitel 4.3 beschrieben, realisiert. Zuerst wird die neue Version des Prozessmodells deployed, wobei die alte Version weiterhin aktiv bleibt. Daraufhin soll die Migration der ausgewählten Instanz erfolgen. Hierzu sind folgende Schritte notwendig, die in den nächsten Kapiteln aufgeführt und ausführlich erklärt werden:

1. Der XML-Parser wird aufgerufen, um die neue Version des Prozessmodells so zu verändern, dass beide Versionen parallel aktiv bleiben können.
2. Das eigentliche Deployment wird wie in Kapitel 3.7 beschrieben ausgeführt.
3. Die Datenbank-Änderungen, die zum Migrieren der Instanz auf die neue Prozessmodell-Version nötig sind, werden durchgeführt.

Danach ist die Migration der Instanz auf das neue Prozessmodell abgeschlossen und die Instanz kann über die *resume*-Funktion weiter ausgeführt werden.

In Abbildung 16 ist der DeploymentService mit der neu integrierten DeployNewVersion-Operation und den bereits früher existierenden Operationen dargestellt.

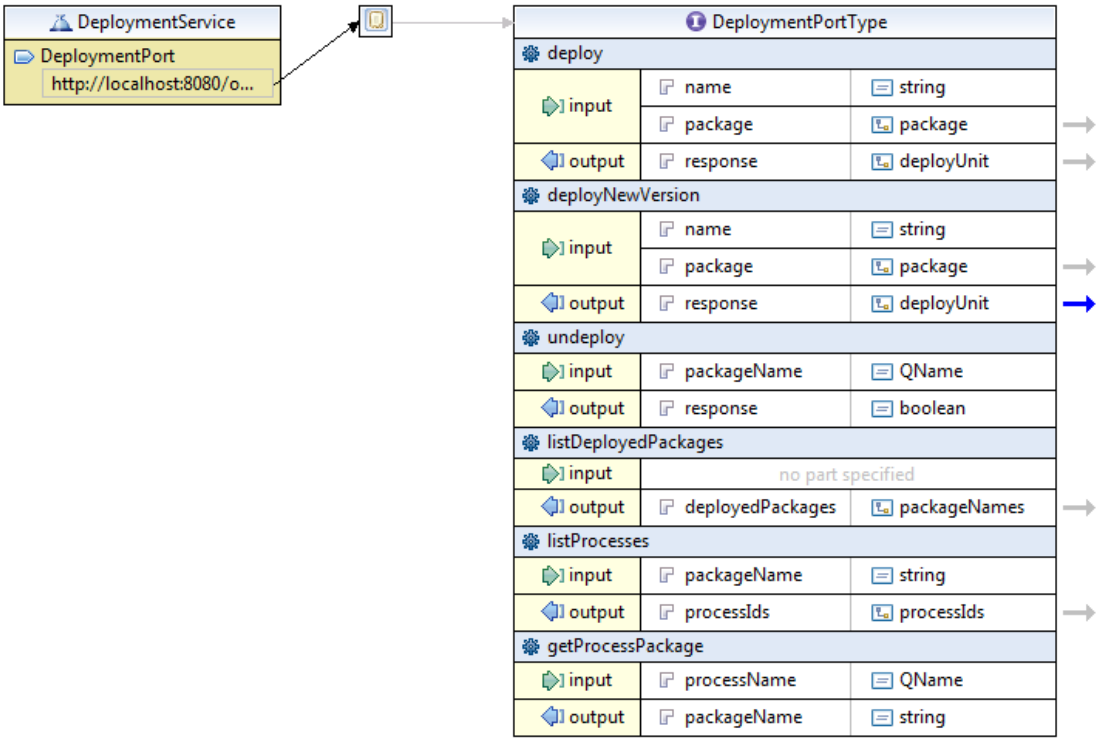


Abbildung 16: graphische Darstellung des Deploy-Web Services mit deployNewVersion-Operation

Abbildung 17 zeigt einen Ausschnitt der Architektur der Apache ODE mit allen Komponenten, die bei der prototypischen Implementierung geändert wurden. Zusätzlich wird in diesem Bild der Client dargestellt, der zwar keine Komponente der Apache ODE ist, aber für die Deploy New Version-Funktionalität benötigt wird. Der Client wurde hauptsächlich zu Testzwecken der Deploy New Version-Funktionalität entwickelt. In das Web-Interface der Apache ODE und die Process and Instance Management-API wurde die *finish()*-Operation integriert, um am Leben gehaltene Prozessinstanzen beenden zu können. Die *finish()*-Operation machte eine Änderung am MySQL-Schema der Apache Ode erforderlich. Der Deploy-Web Service wurde um die *DeployNewVersion()*-Operation erweitert. Um die Funktionalität der *DeployNewVersion()*-Operation zu realisieren, wurde innerhalb des ODE BPEL Compilers ein Parser implementiert. Aufgabe des Parsers ist es, eine eindeutige Adressierung der Prozessversionen zu ermöglichen. Innerhalb der ODE BPEL Runtime wurde die Deploy New Version-Funktionalität prototypisch für die sequence-Aktivität implementiert. Die veränderte Implementierung der sequence-Aktivität ermöglicht zusätzlich, dass eine Prozessinstanz am Leben erhalten werden kann. Alle Änderungen werden im Detail in diesem Kapitel beschrieben.

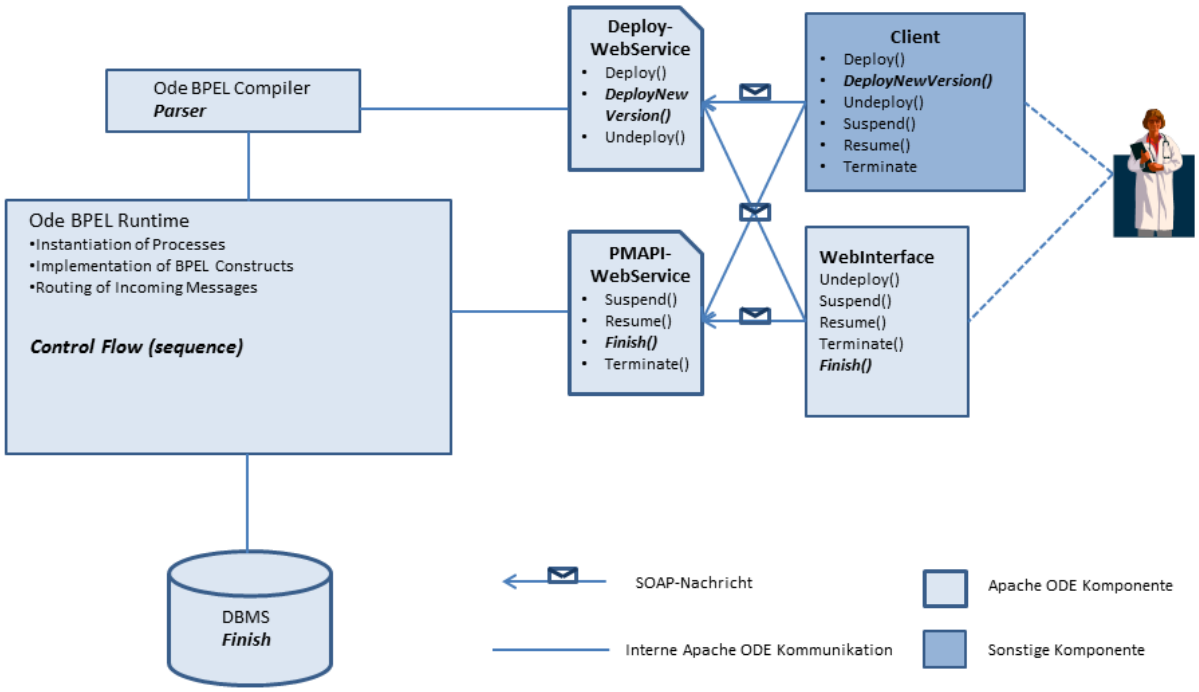


Abbildung 17: Architekturbild der Apache ODE mit Änderungen

5.1 Deploy New Version-Client

Der Deploy New Version-Client ist ein einfacher Java-Command-Line-Client, der mit dem Web Service-Interface der Apache ODE kommuniziert. Der Client dient dem Aufruf von Operationen des Deploy- und InstanceManagement -Web Services der Apache ODE. Unterstützt werden die *deploy()-, undeploy()-, suspend()-, resume()- und terminate()*-Funktionen. Der bereits zu Testzwecken bestehende Client wurde im Zuge dieser Arbeit erweitert, um die neue *deployNewVersion()*-Funktion der ODE aufrufen zu können. Zusätzlich wurde eine GUI erstellt (siehe Abbildung 18), die es Benutzern auf einfache und graphische Weise erlaubt, die Deploy New Version-Funktionalität der

ODE zu nutzen. Die GUI wurde in erster Linie zu Testzwecken erstellt, da in Zukunft die Deploy New Version-Funktion direkt von einem erweiterten Modellierungswerkzeug aus aufgerufen werden wird.

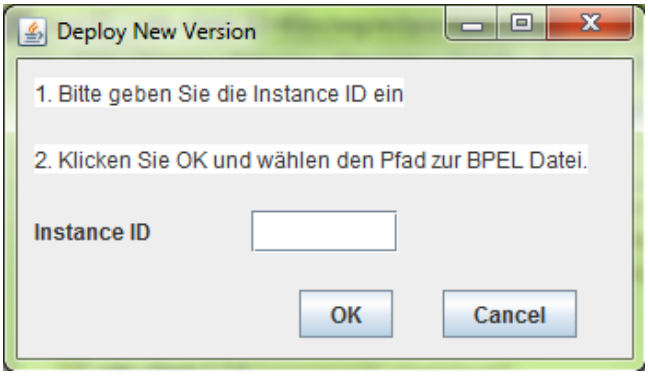


Abbildung 18: graphische Oberfläche des Deploy New Version-Client

Um die Deploy New Version-Funktionalität aufzurufen, muss zuerst die Instanz-ID der zu migrierenden Instanz eingetragen werden. Diese kann der Benutzer beispielsweise in der Web-Oberfläche der Apache ODE unter dem Reiter “Instances“ finden. Durch Drücken des OK-Buttons wird ein Dateiauswahl-Dialog geöffnet, in dem die BPEL-Datei der neuen Prozessmodell-Version ausgewählt wird.

Die Klasse ODEClientAdapter des Clients wurde um die *deployNewVersionProcess()*-Methode erweitert (Listing 19). Innerhalb der Methode werden alle Dateien des Verzeichnisses, in dem die BPEL-Datei liegt, das Deployment Bundle, zu einer zip-Datei gepackt. Daraufhin werden das zip-Paket und die Instanz-ID über *deployNewVersionService.deploy()* an die DeployNewVersion-Operation des ODE Deploy-Web Service gesendet.

```
public static String deployNewVersionProcess(IPath path, String InstanceID) {
        String processFileName = "test";
        processFileName = path.removeFileExtension().lastSegment();

        try {

                //send Instance ID to the server
                DeployUnit ID = new DeployUnit();
                ID.setInstanceID(InstanceID);
                File processFolder = path.toFile().getParentFile();
                System.out.println(processFolder);
                ByteArrayOutputStream dataOut = ManagementAPIHandle..zipFolder(processFolder);
                DeployNewVersionServicePortType deployNewVersionService = new
                DeployNewVersionService().getDeployNewVersionServiceSOAP11PortHttp();

                Package zipPackage = new Package();
                Base64Binary zip = new Base64Binary();
                zip.setValue(dataOut.toByteArray());
                dataOut.close();
                zipPackage.setZip(zip);
```

```
                DeployUnit response = deployNewVersionService.deploy(processFileName, zipPackage,
                InstanceID);

                return response.getName();

        } catch (RemoteException e) {
                JOptionPane.showMessageDialog(null, "Error Message: " + e.getMessage(), "Exception
                during Deploy New Version", JOptionPane.ERROR_MESSAGE);
        } catch (IOException e) {
                JOptionPane.showMessageDialog(null, "Error Message: " + e.getMessage(), "Exception
                during Deploy New Version", JOptionPane.ERROR_MESSAGE);
        } catch (Exception e) {
                JOptionPane.showMessageDialog(null, "Error Message: " + e.getMessage(), "Exception
                during Deploy New Version", JOptionPane.ERROR_MESSAGE);
        }
        return null;
}
```

Listing 19: deployNewVersionProcess()-Methode innerhalb des ODE-Clients

5.2 Erweiterung des ODE Deployment-Mechanismus

Zu Beginn dieser Arbeit gab es einige Überlegungen, wie man die eindeutige Adressierung von Prozessversionen innerhalb der Apache ODE ermöglichen kann, etwa über das Ändern von Namespaces oder Ports. Der Ablauf der Erweiterung des ODE Deployment Mechanismus ist in Abbildung 19 durch ein UML-Sequenzdiagramm graphisch dargestellt. Der letzte Schritt dieses Mechanismus, das Umhängen der Instanz in der ODE Datenbank, wird in Kapitel 5.4 detailliert beschrieben. Die vorherigen Schritte sind Teil des erweiterten ODE Deployment Mechanismus und werden nachfolgend beschrieben.

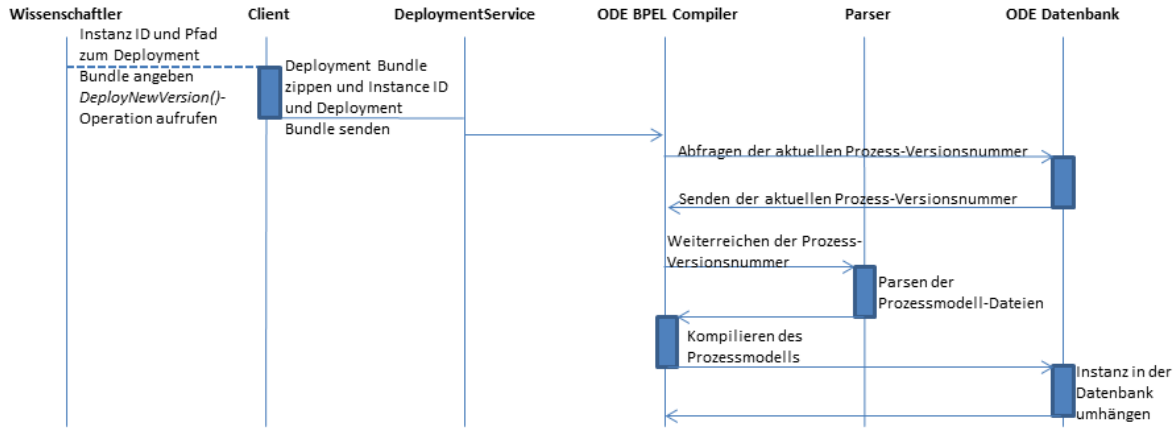


Abbildung 19: Sequenzdiagramm DeployNewVersion()-Operation

Zuerst sendet der Client das Deployment-Bundle an die Apache ODE. Daraufhin wird das Deployment-Bundle von der Apache ODE im Dateiverzeichnis im Ordner WEB-INF/processes gespeichert. Bevor es in der Apache ODE deployed wird, werden die BPEL-, die WSDL- und die deploy.xml-Datei von einem XML-Parser eingelesen und verändert. Diese Veränderungen sind notwendig, um eindeutige Prozessversionen zu erhalten und dadurch Konflikte zwischen den

einzelnen Prozessversionen zu verhindern. Zusätzlich wird dadurch eine eindeutige Adressierung der einzelnen Prozessversionen möglich. Dazu wird die aktuelle und eindeutige Versionsnummer der Apache ODE für deployte Prozesse aus der Datenbank abgefragt und zur Identifizierung des Prozesses an Elemente innerhalb der Dateien gehängt. Genauer gesagt wird der Text **.v** und die Versionsnummer **NR** hinter folgenden Elementen angefügt.

- BPEL-Datei
  - <process name="HelloWorld2**.vNr**">
    - Der Prozessname wird geändert, um eine eindeutige Identifizierung innerhalb der ODE zu ermöglichen.
- WSDL-Datei
  - <wsdl:service name="HelloService**.vNr**">
    - Das Attribut *service name* wird zur eindeutigen Identifizierung des Services angepasst.
  - <soap:address location="http://localhost:8080/ode/processes/helloWorld**.vNr**"/>
    - Das Attribut *address location* wird zur eindeutigen Adressierung des Prozessmodells für eingehende Nachrichten geändert.
- Deployment Descriptor
  - <process name="pns:HelloWorld2**.vNr**">
    - Muss aufgrund der Anpassungen in der BPEL-Datei angepasst werden.
  - <service name="wns:HelloService**.vNr**" port="HelloPort"/>
    - Muss aufgrund der Änderungen in der WSDL-Datei angepasst werden.

Nachdem die Änderungen des Parsers in das Datei-System geschrieben sind, wird der Deploy-Mechanismus aufgerufen.

Das Ändern der SOAP *address location* ist die am besten geeignete Lösung für die Apache ODE. Bei allen anderen Lösungsversuchen wurde die neue Version des Prozessmodells als neue Version erkannt und die vorherige Version automatisch retired. Dies ist nur zu verhindern, wenn der Erkennungsmechanismus, der eine neue Version identifiziert, in seiner Implementierung geändert wird.

5.3 Abgelaufene Instanzen am Leben erhalten

Um der Anforderung gerecht zu werden, dass abgelaufene Prozessinstanzen in den Zustand SUSPENDED wechseln und somit am Leben bleiben, wurden verschiedene Ansätze untersucht. Die erste Idee war eine Dummy-Aktivität, an der erkannt wird, wann das Prozessmodell zu Ende ist, einzuführen. Diese Aktivität wird als letzte Aktivität in die äußerste Sequence eingefügt. Dazu wird ein zweites Mal der Parser aufgerufen, der in die äußerste Sequence als letzte Aktivität eine *wait*-Aktivität mit einer Dauer von einer Sekunde in die BPEL-Datei schreibt. Oberhalb der Dummy-Aktivität können innerhalb der Sequence beliebig viele verschachtelte Aktivitäten stehen. Der Navigationsmechanismus wurde so verändert, dass die Prozessinstanz suspended wird, wenn nur noch eine Aktivität auszuführen bleibt und diese eine Instanz von OWait mit dem Namen „dummy“ ist. Dadurch wird die Ausführung der letzten Aktivität (d.h. der Dummy-Aktivität) nie begonnen und die Instanz kann später über die *resume*-Funktion weiter ausgeführt werden.

Da dieser Ansatz einen Eingriff in das Prozessmodell erforderlich macht, wurde ein zweiter Ansatz entwickelt. Dieser Ansatz wurde prototypisch implementiert, da er im Gegensatz zum ersten Ansatz nur eine Veränderung der Engine beinhaltet und nicht eine Veränderung am Prozessmodell. Auch dieser Ansatz wurde im Rahmen der sequence-Aktivität untersucht. Die Implementierung der sequence-Aktivität wurde so verändert, dass die Prozessinstanz pausiert wird anstatt diese zu beenden. Insbesondere werden nur Instanzen am Leben erhalten, die normal (d.h. ohne Fehler) beendet werden. Das heißt, nach der Ausführung eines *fault-handlers* geht eine Instanz nicht automatisch in den Zustand SUSPENDED über, sondern je nach Fehler in den Zustand ERROR oder FAILED. Wenn Instanzen auch am Leben erhalten werden sollen, nachdem ein Fehler aufgetreten ist, ist eine Erweiterung der Implementierung nötig.

Innerhalb der Klasse SEQUENCE wurde zuerst die *completed()*-Methode der Sequence verändert. Anstatt die Instanz zu beenden, wird diese zurück in die Execution Queue geschrieben und in den Zustand SUSPENDED überführt. In der Execution Queue befindet sich dann die Sequence, die in ihrer *_remaining*-Liste die als letztes ausgeführte Aktivität beinhaltet. Die *_remaining*-Liste gehört zur Prozessinstanz und beinhaltet normalerweise alle noch auszuführenden Aktivitäten der Sequence-Instanz. In diesem Sonderfall beinhaltet sie eine bereits ausgeführte Aktivität, da es sich um die letzte Aktivität gehandelt hat und diese benötigt wird, um die Instanz am Leben zu erhalten.
Wenn die Instanz nach der Migration fortgeführt wird, muss darauf geachtet werden, dass diese Aktivität nicht nochmals ausgeführt wird.
In Listing 20 ist der innerhalb der completed()-Methode eingefügte Code dargestellt. Wenn die Sequence die äußerste Aktivität ist, die Instanz nicht beendet werden soll und in der _remaining-Liste genau eine Aktivität vorhanden ist, wird wie beschrieben die Sequence in die Execution Queue zurückgeschrieben und daraufhin suspended.

```
if (_self.o.getParent().getParent() == null &&!letItFinish && _remaining.size() == 1)
{
        // SEQUENCE in Execution Queue schreiben
        ArrayList<OActivity> remaining = new ArrayList<OActivity>(_remaining);
        instance(new SEQUENCE(_self, _scopeFrame, _linkFrame,
        remaining, comps, false));

        // suspend Instance
        DebuggerSupport debugSupport =
        getBpelRuntimeContext().getBpelProcess().getDebuggerSupport();
        debugSupport.suspend(process_ID);

        // SUSPEND-Flag setzen
        try {
                st = conn.createStatement();
                int result = st.executeUpdate("INSERT INTO ode_instance_migration (InstanceID,
                        SUSPENDED) VALUES (" + process_ID + ", 1) ON DUPLICATE KEY
                        UPDATE SUSPENDED = 1");
        } catch (SQLException e) {
                e.printStackTrace();
        }
}
```

Listing 20: Am Leben erhalten der Prozessinstanz

Um zu gewährleisten, dass innerhalb der Apache ODE überall bekannt ist, dass die Instanz am Leben erhalten wurde, wurde in der Datenbank eine neue Tabelle erstellt. In dieser Tabelle wird, nachdem die Instanz suspended wurde, die Spalte SUSPENDED für diese Instanz auf 1 gesetzt. Wenn die Instanz-ID noch nicht in der Tabelle existiert, so wird eine neue Zeile erstellt. Dies ist erforderlich, um sicherzustellen, dass beim Resumen der Instanz bekannt ist, ob sie in ihrer *_remaining*-Liste eine schon abgearbeitete Aktivität beinhaltet oder nicht.

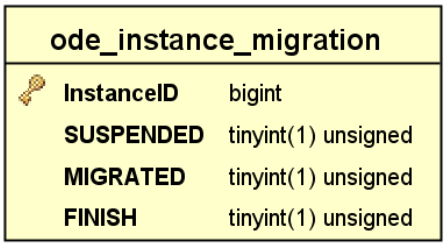


Abbildung 20: ode_instance_migration-Tabelle

Wenn eine am Leben gehaltene Instanz resumed wird, ohne davor migriert worden zu sein, muss innerhalb der *run()*-Methode zu Beginn der Klasse SEQUENCE, der ersten Methode, die nach dem resume ausgeführt wird, dieser Fall behandelt werden. Zuerst werden die Variablen *runOutofWork*, *letItFinish* und *wasMigrated* aus der Tabelle *ode_instance_migration* ausgelesen. Diese Variablen sind true, wenn der Wert in der jeweiligen Spalte für die Instanz *1* ist. Wenn genau eine Aktivität in *_remaining* ist, *runOutofWork true* ist und *letItFinish false* ist, darf die eine Aktivität nicht ausgeführt werden, da es sich um die Aktivität handelt, die bereits ausgeführt wurde bevor die Instanz am Leben erhalte wurde. In diesem Fall wird die Sequence einfach zurück in die Execution Queue geschrieben und wieder in den Zustand SUSPENDED überführt (Listing 21).

```
else if (_remaining.size() == 1 && runOutofWork && !letItFinish )
{
        // SEQUENCE in Execution Queue schreiben
        TreeSet<CompensationHandler> comps = new TreeSet<CompensationHandler>(
        _compensations);
        ArrayList<OActivity> remaining = new ArrayList<OActivity>(
        _remaining);

        instance(new SEQUENCE(_self, _scopeFrame, _linkFrame, remaining, comps, false));

        //SEQUENCE in Zustand SUSPENDED überführen
        DebuggerSupport debugSupport =
        getBpelRuntimeContext().getBpelProcess().getDebuggerSupport();
        debugSupport.suspend(process_ID);
}
```

Listing 21: Am Leben erhalten einer Instanz die resumed aber nicht migriert wurde

5.4 Migration der Prozessinstanz

Für die Apache ODE gibt es Datenbank-Schemas für Derby und MySQL. In dieser Arbeit wird nur das MySQL-Schema betrachtet. In Abbildung 21 sind die wichtigsten Tabellen für die Deploy New Version-Funktionalität abgebildet. In jeder dieser drei Tabellen besteht eine Beziehung zwischen Prozess-ID und Instanz-ID. In der Tabelle *ode_process_instance* stellt die Spalte *ID*, die Instanz-ID dar.

Hier werden eine Instanz, ihre Status-Informationen, ihr Ausführungsstatus sowie das dazugehörige Prozessmodell, das durch die ID identifiziert ist, hinterlegt. Ebenfalls wird in dieser Tabelle der dazugehörige *Correlator* über seine ID einer Instanz zugewiesen. In *ode_event* werden alle Events, die zu einer Instanz gehören, gespeichert. Zusätzlich zu der Instanz-ID ist auch die dazugehörige Prozess-ID hinterlegt.

ode_message_exchange ist die Tabelle, in der alle Nachrichten mit allen dazugehörigen Informationen abgelegt werden, unter anderem auch die Instanz- und Prozess-ID, zu denen eine Nachricht gehört. Im Anhang MySQL-Schema sind die Tabellen des Schemas und ihre Primärschlüssel graphisch dargestellt.

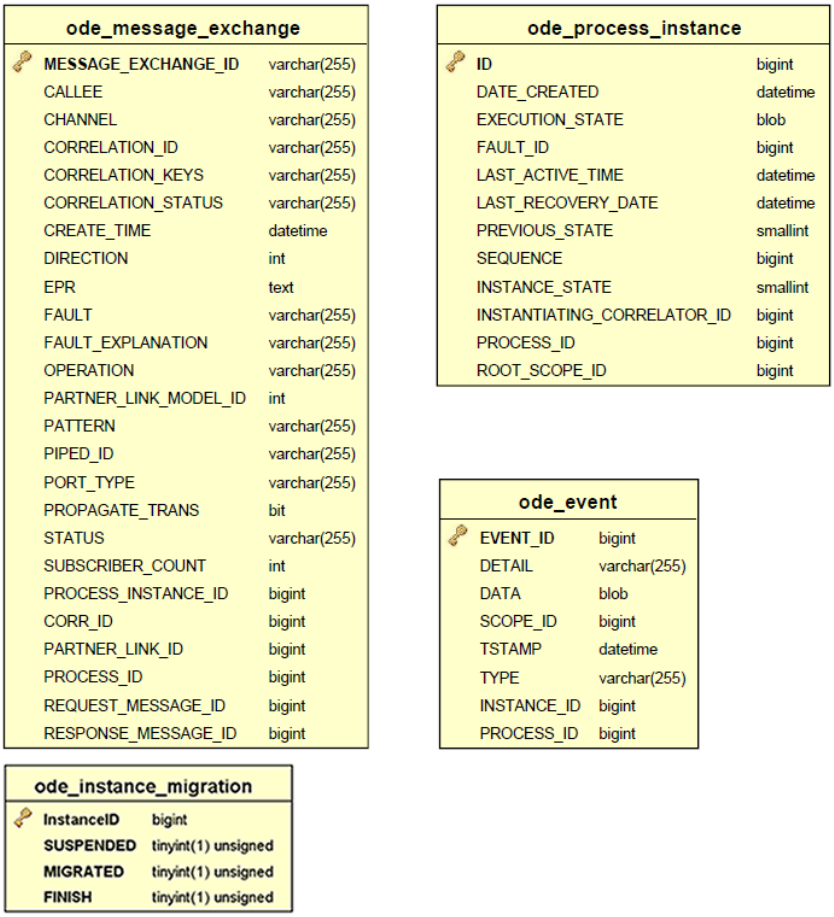


Abbildung 21: Schema der für diese Arbeit wichtigsten Tabellen

Nach Ausführung des in Kapitel 5.2 beschriebenen Deployment-Mechanismus, ist die neue Prozessmodell-Version deployed und in der Datenbank angelegt. Um eine Instanz von einer alten Prozessmodell-Version auf eine neue Version zu migrieren, müssen in der Datenbank an manchen Stellen die Prozess-ID der alten Prozessversion auf die der neuen Prozessversion umgeschrieben werden. Die ID der Instanz, die migriert werden soll, ist bekannt, da sie vom Deploy New Version-Client mitgeschickt wird. Alle weiteren benötigten IDs werden zuerst aus der Datenbank ausgelesen, um dann folgende Änderungen an den drei Tabellen aus Abbildung 21 durchzuführen:

ode_process_instance:

- Ersetzen der Prozess-ID durch die ID der neuen Prozessversion für die gegebene Instanz-ID.
- Ersetzen der Correlator-ID durch die Correlator-ID der neuen Prozessversion für die gegebene Instanz-ID.

ode_event:

- Ersetzen der Prozess-ID durch die ID der neuen Prozessversion für die gegebene Instanz-ID.

ode_message_exchange:

- Ersetzen der Prozess-ID durch die ID der neuen Prozessversion für die gegebene Instanz-ID.

Zusätzlich zu den oben beschriebenen Datenbank Änderungen wird in der Tabelle *ode_instance_migration* die Spalte MIGRATED für die jeweilige Instanz-ID auf 1 gesetzt. Dadurch ist überall bekannt, ob die Instanz migriert worden ist oder nicht und es kann sichergestellt werden, dass die Logik zur Migration nur wenn nötig ausgeführt wird.

In der SEQUENCE-Klasse, die die sequence-Aktivität repräsentiert, gibt es eine Liste *_remaining*. In dieser Liste stehen alle Aktivitäten, die noch abgearbeitet werden müssen. Standardmäßig wird diese Liste beim Beginn der Sequence erstellt und daraufhin abgearbeitet. Zusätzlich gibt es eine Liste *sequence*. Diese Liste wird in der Klasse *OSequence* implementiert, die eine Aktivität des Modells repräsentiert, aber innerhalb der SEQUENCE-Klasse bekannt ist Sie beinhaltet alle Aktivitäten, die in der Sequence vorhanden sind. Diese Liste wird nach der Migration auf eine neue Prozessversion automatisch aktualisiert. Die *_remaining*-Liste dagegen beinhaltet nach Migration der Instanz immer noch die Aktivitäten der alten Prozessversion, da die Liste von der Instanz abhängig ist.

Die erste Methode innerhalb der SEQUENCE-Klasse, die nach resumen der Instanz ausgeführt wird, ist *run()*. Innerhalb dieser Methode ist die Logik zum Migrieren einer Instanz auf eine neue Prozessversion implementiert. Zuerst werden die Werte aus der *ode_instance_migration*-Tabelle ausgelesen und in Variablen gespeichert. Für die jeweilige Instanz ist *wasMigrated true*, wenn die Spalte MIGRATED den Wert *1* hat. In Listing 22 ist der Code, der ausgeführt wird, wenn die Instanz migriert wurde, dargestellt. Wenn *wasMigrated* true ist, wird die *_remaining*-Liste aktualisiert. Die Variable *runOutofWork* ist *true*, wenn die SUSPENDED Spalte den Wert 1 beinhaltet. Dies ist der Fall, wenn die Instanz davor automatisch am Leben erhalten wurde, wie in Kapitel Abgelaufene Instanzen am Leben erhalten5.3 beschrieben. Für diese Konstellation wird zuerst die erste Aktivität aus der *_remaining*-Liste entfernt. Diese erste Aktivität ist die vor dem suspenden der Instanz ausgeführte Aktivität. Danach ist die *_remaining*-Liste korrekt und die Ausführung der Aktivitäten kann beginnen. Um zu gewährleisten, dass eine Instanz mehrmals migriert werden kann, werden vor der Ausführung noch die Werte für die Instanz in den Spalten SUSPENDED und MIGRATED auf *0* gesetzt.

```
if (wasMigrated)
{
        List<OActivity> aSequence = ((OSequence) _self.o).sequence;
        Integer size = aSequence.size();
        Object oAcivity = _remaining.get(0);
        Integer sIndex = aSequence.indexOf(oAcivity);
```

```
                List<OActivity> remaining = aSequence.subList(sIndex,
                size);
                _remaining = remaining;

//Fall 3:       Instanz war automatisch im Zustand SUSPENDED wurde migriert und resumed
                if (runOutofWork)
                {
                        _remaining.remove(0);
                        runOutofWork = false;
                }
                try {
                        st = conn.createStatement();
                        int result = st.executeUpdate("UPDATE ode_instance_migration SET SUSPENDED
                        = 0, MIGRATED = 0 WHERE InstanceID = " + process_ID);
                } catch (SQLException e) {
                        e.printStackTrace();
                }
}
```

Listing 22: Implementierung der Aktualisierung von _remaining

Abbildung 22 veranschaulicht die grundlegende Aktualisierung der Aktivitäten-Liste nach erfolgreicher Migration der Prozessinstanz auf eine neue Prozessversion.

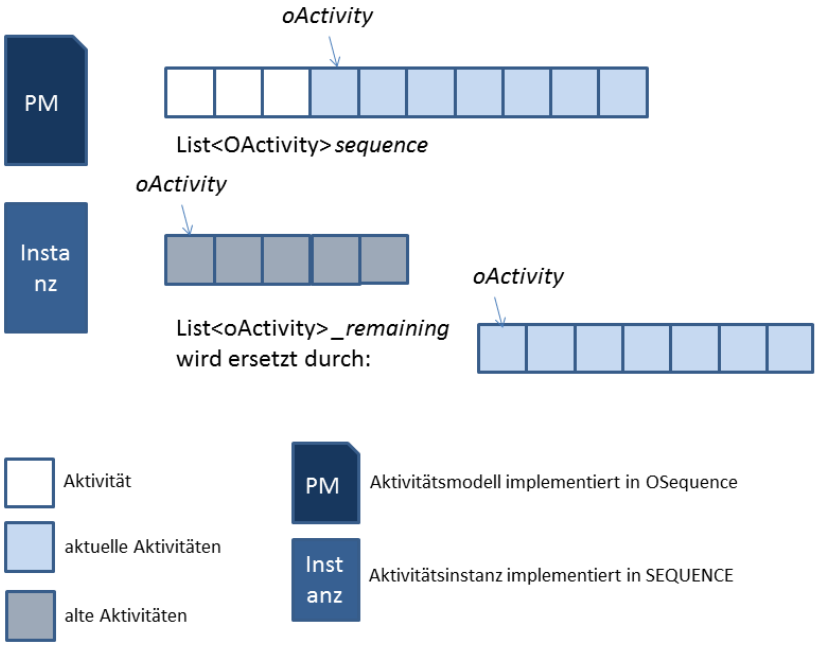


Abbildung 22: Aktualisieren der _remaining-Liste

Die derzeitige aktuelle (d.h. laufende) Aktivität ist *oActivity*. *_remaining* wird durch eine Teilliste der möglicherweise veränderten Aktivitätsliste der Sequence (*sequence*-Liste) ersetzt. Die erste Aktivität der Teilliste ist die laufende Aktivität (d.h. *oActivity*), die letzte Aktivität ist die letzte Aktivität der *sequence*-Liste. Das heißt, die Teilliste ist die *sequence*-Liste ohne die bereits ausgeführten

Aktivitäten. Dadurch wird sichergestellt, dass in *_remaining* auch nach Migration der Instanz immer die aktuellen Aktivitäten stehen.

5.5 Beenden von am Leben gehaltenen Prozessinstanzen

Um eine am Leben gehaltene Prozessinstanz zu beenden, wird die Spalte FINISH in der Tabelle *ode_instance_migration* benötigt. Der Wert dieser Spalte ist standardmäßig 0 und wird auf 1 gesetzt, wenn für eine Instanz die *finish()*-Funktion aufgerufen wird. Dadurch wird gewährleistet, dass die Instanz, auch wenn sie zwischenzeitlich nochmals suspended wurde oder ein Fehler in der Apache ODE auftritt, definitiv beendet wird sobald die Instanz wieder ausgeführt wird.

Die Management API stellt wie in Kapitel 3.4 beschrieben alle Funktionen zur Verfügung, um Prozesse und Instanzen zu verwalten. Der InstanceManagement-Teil der Management API wurde um die *finish()*-Funktionalität erweitert. Die *finish()*-Funktionalität lässt eine Instanz zu Ende laufen. Bei erfolgreicher Ausführung befindet sich die Instanz anschließend im Status COMPLETED. Dazu wurde in der Klasse *ProcessAndInstanceManagementImp*l folgende Methode implementiert:

```
public InstanceInfoDocument finish(Long iid) throws ManagementException
{
  DebuggerSupport debugSupport = getDebugger(iid);
  assert debugSupport != null : "getDebugger(Long) returned NULL!";
  debugSupport.finish(iid);
  return getInstanceInfo(iid);
}
```

Listing 23: Implementierung der *finish*-Funktionalität in ProcessAndInstanceMangementmpl

Außerdem wurde die Methoden-Signatur in das Instance-Management-Interface eingefügt:

```
/**
 * Finishes the (previously suspended) instance. This operation only affects
 * process instances that are in the suspended state.
  * @param iid
 *          instance id
 * @return post-change instance information
 */
InstanceInfoDocument finish(Long iid);
```

Listing 24: *finish*-Funktion in der Mangement API

Die Funktionalität steht als Web Service und als Button in der Web-Oberfläche der Apache ODE zur Verfügung. Die graphische Darstellung der Management-API ist im Anhang Process and Instance Management API zu finden.

Die finish()-Funktion, um abgearbeitete Prozess-Instanzen endgültig als beendet zu markieren, wird in der Klasse BpelProcess realisiert (Listing 25). Bei Aufruf der *finish()*-Funktion wird die Spalte FINISH der Tabelle *ode_instance_migration* für die zugehörige Instanz auf 1 gesetzt. Dadurch wird es möglich, aus anderen Klassen heraus abzufragen, ob die Instanz beendet oder standardmäßig wieder suspended werden soll. Die Prozessinstanz wird wie beim Aufruf der *resume()*-Funktion fortgesetzt.

```
case FINISH:
        if (__log.isDebugEnabled()) {
```

```
                __log.debug("handleWorkEvent: ResumeWork (and let it finish) event for iid "
                                + we.getInstanceId());
        }


        // set finish flag in database
         Connection conn = DatabaseConnection.getInstance().getConnection();
        Statement st = null;
        try {
                st = conn.createStatement();
                int result = st.executeUpdate("INSERT INTO ode_instance_migration (InstanceID,
                FINISH) VALUES (" + we.getInstanceId() + ", 1) ON DUPLICATE KEY UPDATE
                FINISH = 1");
        } catch (SQLException e) {
                e.printStackTrace();
        }

        BpelRuntimeContextImpl processInstance5 = createRuntimeContext(procInstance, null,
        null);

         processInstance5.execute();
break;
```

Listing 25: case FINISH in BpelProcess

In Listing 26 ist der Code dargestellt, der innerhalb der *completed()*- und *run()*-Operation der SEQUENCE-Klasse prüft, ob für die Instanz die *finish*-Funktion aufgerufen wurde und dadurch in der Tabelle *ode_instance_migration* die Spalte FINISH für die Instanz den Wert *1* hat.

```
Connection conn = DatabaseConnection.getInstance().getConnection();
Statement st = null;
ResultSet rs = null;

// get data from ode_instance_migration table
boolean letItFinish = false;
try {
        st = conn.createStatement();
        rs = st.executeQuery("SELECT FINISH FROM ode_instance_migration WHERE InstanceID
        = " + process_ID);

        if (rs.next())
        {
                int finish = rs.getInt("FINISH");

                letItFinish = (finish == 1);
        }
} catch (SQLException e) {
        e.printStackTrace();
}
```

Listing 26: *run()*- und *completed()*-Methode check letitFinish

Es sind drei verschiedene Szenarien möglich:

- Die *finish()*-Operation wurde aufgerufen. Davor wurde die Instanz am Leben erhalten und nicht migriert.
  Ist dies der Fall, so sind die Variablen *letItFinish* und *runOutofWork true*. Die *_remaining*-Liste enthält genau eine Aktivität. Diese Aktivität wurde bevor die Instanz in den Status SUSPENDED überführt wurde, bereits ausgeführt. Aus diesem Grund wird die Aktivität aus der *_remaining*-Liste entfernt und die Methode *completed* aufgerufen. Innerhalb der *completed*-Methode wird die Instanz dann nicht weiter am Leben erhalten, sondern da *letItFinish true* ist beendet.

```
if (_remaining.size() == 1 && runOutofWork && letItFinish)
{
        _remaining.remove(0);
        TreeSet<CompensationHandler> comps = new TreeSet<CompensationHandler>(
        _compensations);
        Activity_Complete(false, comps);
}
```

Listing 27: *run()*-Methode Instanz beenden

- Die *finish()*-Operation wurde aufgerufen. Davor wurde die Instanz am Leben erhalten und migriert.
  In diesem Fall sind beim Betreten der *run()*-Methode *runOutofWork, wasMigrated und letItFinish* true. Es wird wie in Kapitel 5.4 beschrieben die *_remaining*-Liste aktualisiert und die erste Aktivität aus der *_remaining*-Liste entfernt. *runOutofWork* und *wasMigrated* werden auf *false* gesetzt. Daraufhin wird die Instanz normal ausgeführt. Wenn die letzte Aktivität ausgeführt worden ist, wird die c*ompleted()*-Methode aufgerufen. Innerhalb der Methode wird der else-Zweig ausgeführt da *letItFInish* den Wert true hat. Es wird die Zeile der Instanz in der *ode_instance_migration*-Tabelle gelöscht und die *ActivityComplete()*-Methode aufgerufen, die die Instanz endgültig beendet.

```
// finish Instance
else
        {
                try {
                st = conn.createStatement();
                rs = st.executeQuery("DELETE ode_instance_migration WHERE
                InstanceID = " + process_ID);
        } catch (SQLException e) {
                e.printStackTrace();
        }
        Activity_Complete(_terminateRequested, comps);
        }
```

Listing 28: *completed()*-Methode Instanz beenden

- Die *finish()*-Operation wurde aufgerufen. Davor wurde die Instanz migriert und nicht am Leben erhalten.

In diesem Fall ist der Ablauf derselbe, wie wenn die Instanz zusätzlich auch noch am Leben gehalten wurde. Der einzige Unterschied ist, dass die erste Aktivität aus der *_remaining*-Liste nicht gelöscht wird. Dies ist nicht erforderlich, da nur Aktivitäten in der *_remaining*-Liste stehen, die noch nie ausgeführt wurden.

In Abbildung 23 wird das Zusammenspiel der einzelnen Komponenten bei der *finish()*-Operation mit Hilfe eines Sequenzdiagramms veranschaulicht.

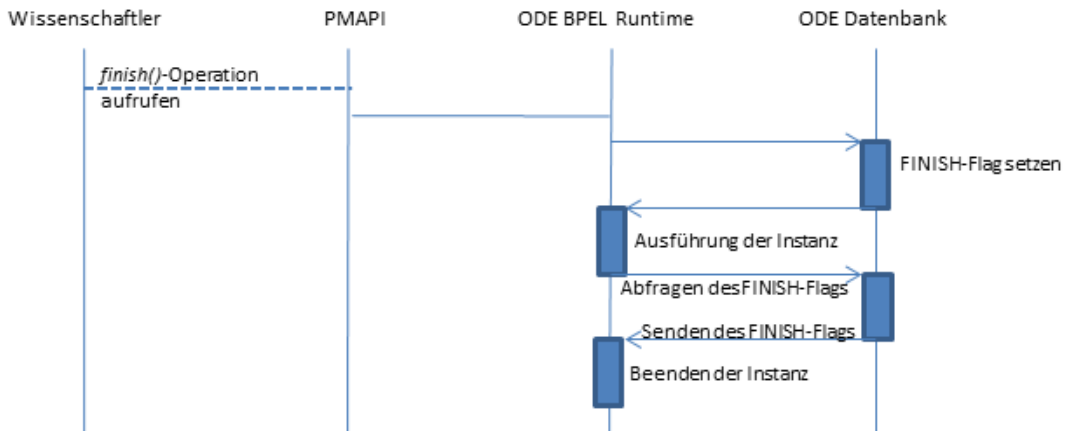


Abbildung 23: Sequenzdiagramm *finish()*-Operation

5.6 Web-GUI

Um die *finish()*-Funktion auch aus der Web-Oberfläche der Apache ODE heraus aufzurufen, wurde dort ein weiterer Button eingefügt. Dieser Button befindet sich auf dem Reiter Instances bei jeder einzelnen Instanz neben den Buttons „Terminate“, „Suspend“ und „Resume“.

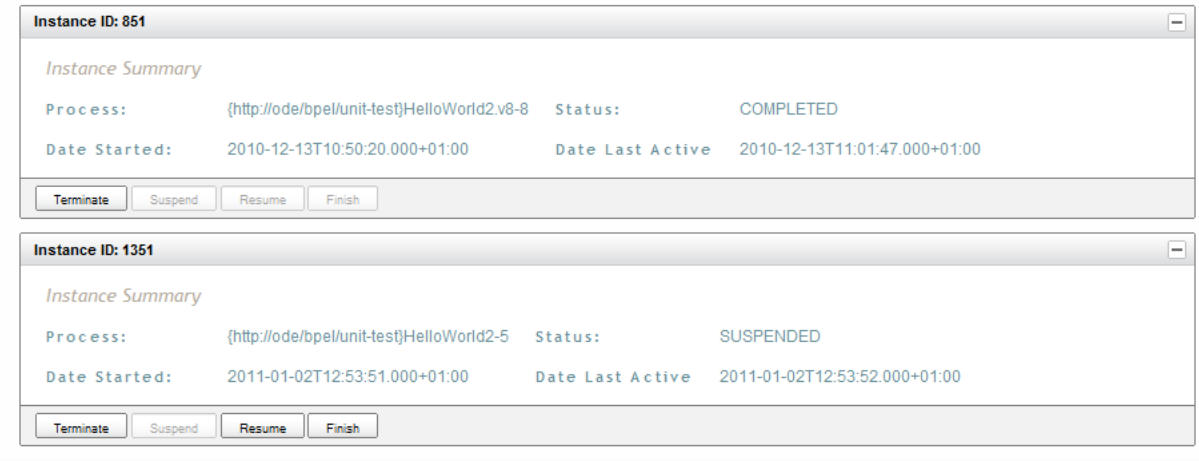


Abbildung 24: Instances-Reiter aus der Apache ODE Oberfläche

Der Finish-Button ist nur aktiv, wenn die Instanz im Status SUSPENDED ist. Beim Klicken des Finish-Buttons wird die Instanz zu Ende ausgeführt. Bei erfolgreicher Ausführung ist die Instanz dann im Status COMPLETED.

5.7 Erweiterung auf die Flow-Aktivität

Im Gegensatz zur sequence-Aktivität werden bei einer flow-Aktivität alle beinhalteten Aktivitäten parallel ausgeführt. Wird der Flow betreten wird von der Apache ODE eine Liste erstellt, die die Aktivitäts-Informationen von allen Aktivitäten innerhalb des Flows, beinhaltet. Aus der jeweiligen *ActivityInfo* werden Activity Guards erstellt und in die Execution Queue geschrieben. Die Activity Guards evaluieren die *JoinConditions* der einzelnen Aktivitäten und führen die Aktivität aus, wenn die *JoinCondition* „*true*“ ist. Wenn die *JoinCondition* für eine Aktivität „*false*“ ist, wird die DeadPathElimination angestoßen. In Listing 29 ist der oben beschriebene Code der Apache ODE aus der FLOW-Klasse dargestellt.

```
for (Iterator<OActivity> i = _oflow.parallelActivities.iterator(); i
                    .hasNext();) {
            OActivity ochild = i.next();
            ChildInfo childInfo = new ChildInfo(new ActivityInfo(
                    genMonotonic(), ochild,
                    newChannel(TerminationChannel.class),
                    newChannel(ParentScopeChannel.class)));
            _children.add(childInfo);

            instance(createChild(childInfo.activity, _scopeFrame, myLinkFrame));
        }
        instance(new ACTIVE());
    }
```

Listing 29: Instanziierung der Aktivitäten innerhalb des Flows

Nachdem alle Aktivitäten innerhalb des Flows beendet sind, wird der Flow beendet. Das am Leben erhalten der Prozess-Instanz, wie in Kapitel 5.3 beschrieben, ist für die Flow-Aktivität so nicht möglich. Beim ersten Lösungsansatz würde die Dummy-Aktivität sofort mit ausgeführt werden und die Apache ODE würde bemerken, dass alle Aktivitäten der Instanz ausgeführt sind und die Instanz in den Status COMPLETED überführen. Der zweite Ansatz kann in einem gewissen Rahmen auf die flow-Aktivität übertragen werden. Eine flow-Aktivität, die sich gerade in der Ausführung also innerhalb der Wavefront befindet und auf eine neue Prozess-Version migriert werden soll, kann nicht so viele Änderungen unterstützen wie beispielsweise die sequence-Aktivität. Wurden innerhalb der flow-Aktivität Aktivitäten hinzugefügt, müssen diese Änderungen direkt in die Execution Queue übertragen werden, da es nicht wie bei der Sequence die Möglichkeit gibt, die Liste der Kindelemente nachzuladen. Das Löschen von Aktivitäten innerhalb eines Flows ist eingeschränkt möglich, da von allen beinhalteten Aktivitäten nur die Activity Guards in die ExecutionQueue geschrieben werden. So lange die eigentliche Ausführung der Aktivität nicht gestartet wurde, können die Activity Guards aus der Execution Queue gelöscht werden. Dabei muss darauf geachtet werden, dass die Verlinkungen zwischen den einzelnen Aktivitäten nicht zerstört werden. Das Ändern von Aktivitäten ist, so lange sie noch nicht gestartet wurden, möglich. Dazu müssen die Activity Guards aktualisiert werden, wenn die flow-Aktivität weiter ausgeführt wird. *Basic Activities*, mit Ausnahme der wait-Aktivität, können nicht geändert werden, wenn sie gerade ausgeführt werden. Für das Ändern von strukturierenden Aktivitäten innerhalb des Flows gelten die in Kapitel 4.4 vorgestellten Einschränkungen. Problemlos ist es möglich, das am Leben erhalten der Prozessinstanz, sowie das Beenden von am Leben erhaltenen Prozessinstanzen auf die flow-Aktivität zu übertragen.

Das Verändern der flow-Aktivität innerhalb einer sequence-Aktivität ist, so lange der Flow sich noch nicht in der Ausführung befindet, mit der prototypischen Implementierung möglich.

6 Anwendungsbeispiel

In diesem Kapitel soll anhand eines einfachen Beispiels der Prototyp und seine Funktionsweise erklärt werden. In Abbildung 25 ist ein Prozessmodell zu sehen, das im BPEL-Designer von einem Benutzer entwickelt wurde. Die Einschränkung des Prototyps, dass alle Logik von einer Sequence-Aktivität umschlossen sein muss, ist erfüllt. Dieses Modell wird auf der Apache ODE deployed.

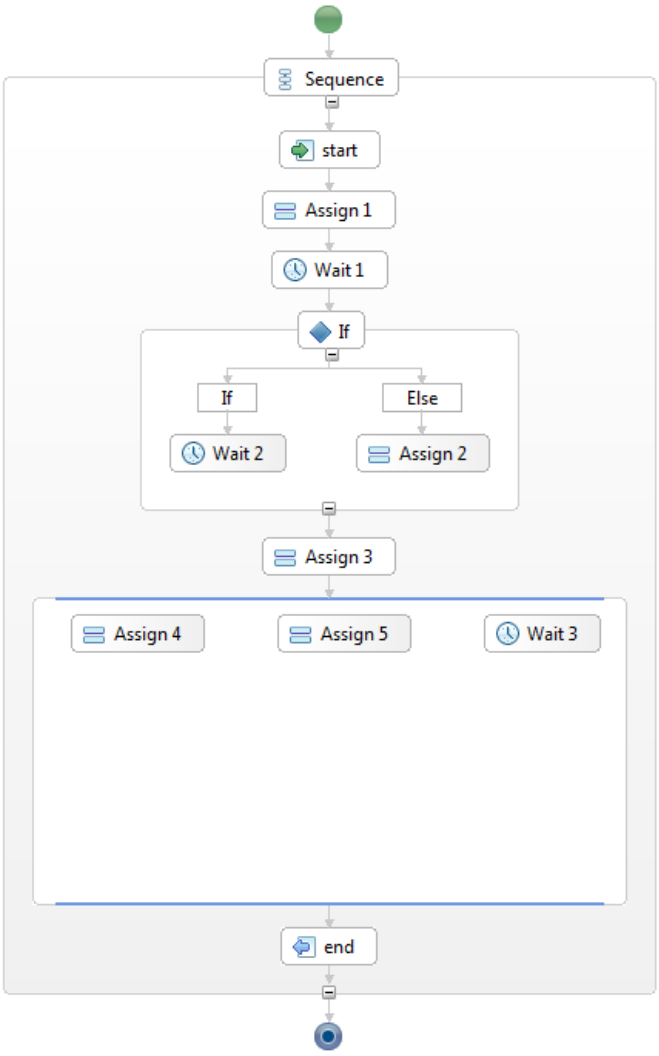


Abbildung 25: Ursprüngliches Prozessmodell

Daraufhin startet der Benutzer eine Instanz des Prozesses und pausiert sie kurz danach. In diesem Beispiel wird die Instanz während der Aktivität *Wait1* suspended. In Abbildung 26 ist zu sehen, welche Aktivitäten bis zu diesem Zeitpunkt abgearbeitet wurden. Abbildung 27 zeigt den Status der Instanz im Reiter Instances in der Apache ODE Web-GUI. Dort ist sichtbar, auf welche Prozessversion die Instanz verlinkt ist, welchen Status sie hat, wann sie gestartet wurde und wann sie zuletzt aktiv war. In diesem Fall ist das zugehörige Prozessmodell HelloWorld-1 und die Instanz befindet sich im Status SUSPENDED. Des Weiteren sind alle Operationen, die im derzeitigen Status der Instanz zur Verfügung stehen durch aktive Buttons erkennbar. Funktionen, die aufgrund des Status nicht verfügbar sind, werden durch ausgegraute Buttons dargestellt.

instance251

Event	Source	Timestamp	Persisted	State	Persisted
Process_Instantiated		Wed Feb 02 17:19:41 CET 2...	false		
Instance_Running		Wed Feb 02 17:19:41 CET 2...	false		
Activity_Ready	/process	Wed Feb 02 17:19:41 CET 2...	false	Ready	true
Activity_Executing	/process	Wed Feb 02 17:19:41 CET 2...	false	Executing	true
Activity_Ready	/process/sequence[1]	Wed Feb 02 17:19:41 CET 2...	false	Ready	true
Activity_Executing	/process/sequence[1]	Wed Feb 02 17:19:41 CET 2...	false	Executing	true
Activity_Ready	/process/sequence[1]/receive[1]	Wed Feb 02 17:19:41 CET 2...	false	Ready	true
Activity_Executing	/process/sequence[1]/receive[1]	Wed Feb 02 17:19:41 CET 2...	false	Executing	true
Variable_Modification	/process/variables[1]/variable[1]	Wed Feb 02 17:19:41 CET 2...	false		
Activity_Executed	/process/sequence[1]/receive[1]	Wed Feb 02 17:19:41 CET 2...	false	Waiting	true
Activity_Complete	/process/sequence[1]/receive[1]	Wed Feb 02 17:19:41 CET 2...	false	Completed	true
Activity_Ready	/process/sequence[1]/assign[1]	Wed Feb 02 17:19:42 CET 2...	false	Ready	true
Activity_Executing	/process/sequence[1]/assign[1]	Wed Feb 02 17:19:42 CET 2...	false	Executing	true
Variable_Modification	/process/variables[1]/variable[2]	Wed Feb 02 17:19:42 CET 2...	false		
Variable_Modification	/process/variables[1]/variable[1]	Wed Feb 02 17:19:42 CET 2...	false		
Activity_Executed	/process/sequence[1]/assign[1]	Wed Feb 02 17:19:42 CET 2...	false	Waiting	true
Activity_Complete	/process/sequence[1]/assign[1]	Wed Feb 02 17:19:42 CET 2...	false	Completed	true
Activity_Ready	/process/sequence[1]/wait[1]	Wed Feb 02 17:19:42 CET 2...	false	Ready	true
Activity_Executing	/process/sequence[1]/wait[1]	Wed Feb 02 17:19:42 CET 2...	false	Executing	true
Instance_Suspended		Wed Feb 02 17:19:52 CET 2...	false		

Abbildung 26: Auditing der Instanz (1)

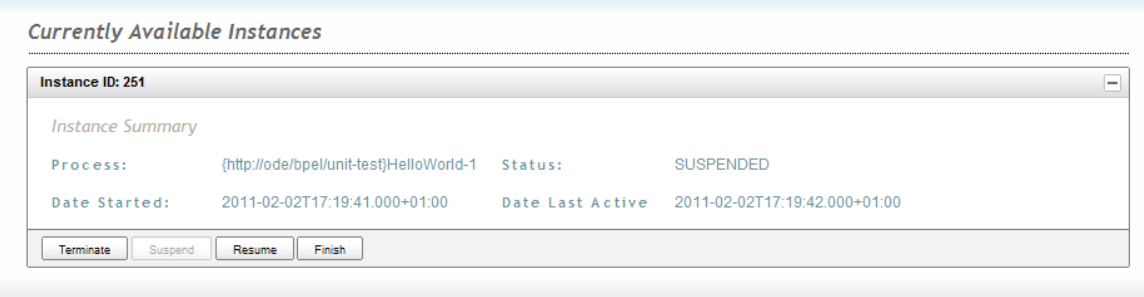


Abbildung 27: Status der Instanz in der Apache ODE GUI (1)

Im nächsten Schritt möchte der Benutzer das Prozessmodell verändern. Die Aktivität *Wait2* aus dem if-Zweig ersetzt er durch eine empty-Aktivität und verschiebt die Wait2-Aktivität in die flow-Aktivität. Die Aktivität *Assign4* wird aus der flow-Aktivität gelöscht und die Aktivität *Empty2* hinzugefügt. Das geänderte Prozessmodell ist in Abbildung 28 zu sehen. Die so genannte Wavefront der Instanz ist auf Höhe der Aktivität *Wait1*, da während dieser Aktivität die Instanz in den Zustand SUSPENDED überführt wurde.

Nachdem der Apache ODE Client gestartet wurde und die Instanz-ID *251* der zu migrierenden Instanz in die Oberfläche eingegeben, sowie der Pfad zu BPEL-Datei des neuen Prozessmodells angegeben wurde, wird das neue Prozessmodell deployed. Dabei werden sämtliche in Kapitel 5 beschriebenen Änderungen an der Datenbank und den Prozessbeschreibungs-Dokumenten durchgeführt.

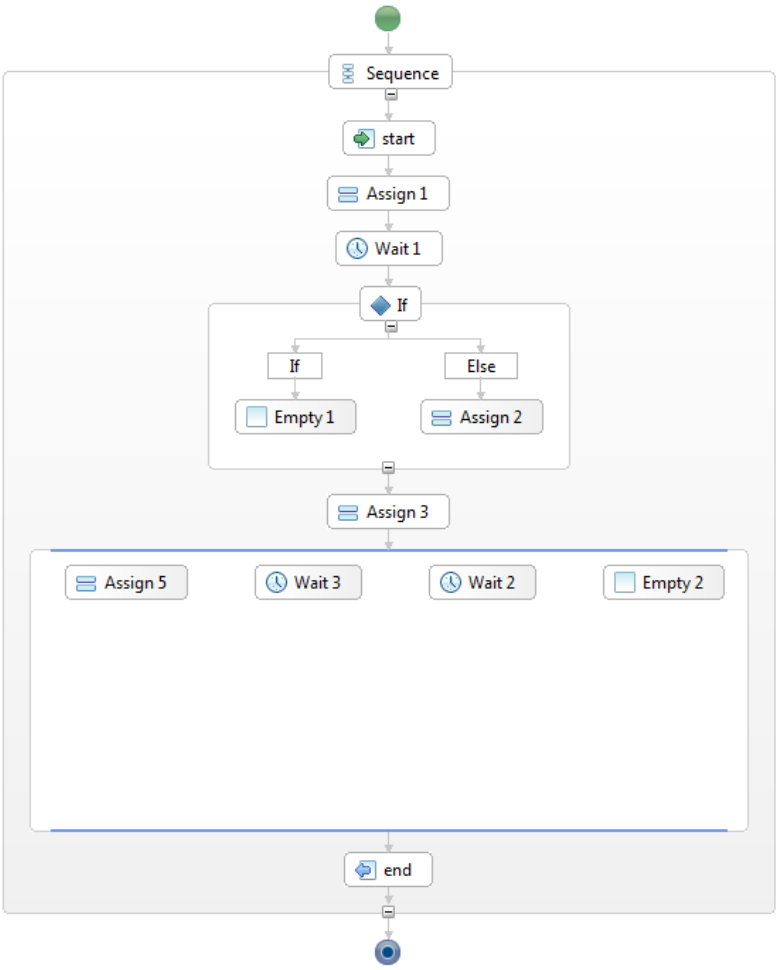


Abbildung 28: Neue Prozessmodell-Version

Nachdem das geänderte Prozessmodell erfolgreich deployed und die Instanz migriert wurde, aktualisiert sich der Reiter Instances der Apache ODE Web-GUI (Abbildung 29). Der Status der Instanz bleibt weiterhin unverändert SUSPENDED ebenso wie das Startdatum der Instanz und der Zeitstempel, der die letzte Aktivität der Instanz bestimmt. Der Name des zu der Instanz gehörenden Prozessmodells hat sich allerdings von *HelloWorld-1* zu *HelloWorld.v2-2* geändert. Die Benennung des Prozessmodells hängt wie in Kapitel 5.2 beschrieben von der aktuellen Versionsnummer innerhalb der Apache ODE ab.

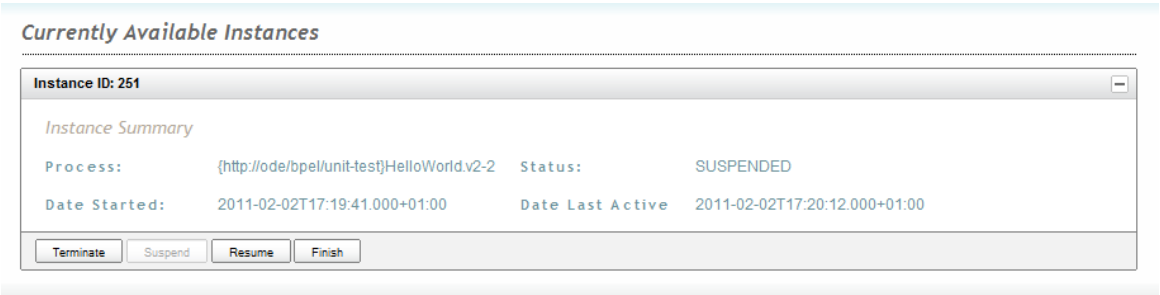


Abbildung 29: Status der Instanz in der Apache ODE GUI (2)

Als nächstes lässt der Benutzer die Instanz weiter laufen. Der über eine Auditing-Komponente mitgeschriebene Instanz-Verlauf ist in Abbildung 30 zu sehen. Die Instanz wird wieder aktiv und führt das geänderte Prozessmodell aus. Nachdem die letzte Aktivität aus Abbildung 28, die reply-Aktivität, ausgeführt wurde, wird die Instanz wieder in den Status SUSPENDED überführt, dieses Mal jedoch automatisch, da für die Sequence-Klasse das am Leben erhalten einer Prozessinstanz implementiert worden ist.

instance251

Event	Source	Timestamp	Persis...	State	Persisted
Activity_Executing	/process/sequence[1]/assign[1]	Wed Feb 02 17:19:42 CET 2...	false	Executing	true
Variable_Modification	/process/variables[1]/variable[2]	Wed Feb 02 17:19:42 CET 2...	false		
Variable_Modification	/process/variables[1]/variable[1]	Wed Feb 02 17:19:42 CET 2...	false		
Activity_Executed	/process/sequence[1]/assign[1]	Wed Feb 02 17:19:42 CET 2...	false	Waiting	true
Activity_Complete	/process/sequence[1]/assign[1]	Wed Feb 02 17:19:42 CET 2...	false	Completed	true
Activity_Ready	/process/sequence[1]/wait[1]	Wed Feb 02 17:19:42 CET 2...	false	Ready	true
Activity_Executing	/process/sequence[1]/wait[1]	Wed Feb 02 17:19:42 CET 2...	false	Executing	true
Instance_Suspended		Wed Feb 02 17:19:52 CET 2...	false		
Instance_Running		Wed Feb 02 17:24:47 CET 2...	false		
Activity_Executed	/process/sequence[1]/wait[1]	Wed Feb 02 17:24:47 CET 2...	false	Waiting	true
Activity_Complete	/process/sequence[1]/wait[1]	Wed Feb 02 17:24:47 CET 2...	false	Completed	true
Activity_Ready	/process/sequence[1]/if[1]	Wed Feb 02 17:24:47 CET 2...	false	Ready	true
Activity_Executing	/process/sequence[1]/if[1]	Wed Feb 02 17:24:47 CET 2...	false	Executing	true
Activity_Dead_Path	/process/sequence[1]/if[1]/empty[1]	Wed Feb 02 17:24:47 CET 2...	false	DeadPath	true
Activity_Ready	/process/sequence[1]/if[1]/else[1]/ass...	Wed Feb 02 17:24:47 CET 2...	false	Ready	true
Activity_Executing	/process/sequence[1]/if[1]/else[1]/ass...	Wed Feb 02 17:24:47 CET 2...	false	Executing	true
Variable_Modification	/process/variables[1]/variable[2]	Wed Feb 02 17:24:47 CET 2...	false		
Variable_Modification	/process/variables[1]/variable[1]	Wed Feb 02 17:24:47 CET 2...	false		
Activity_Executed	/process/sequence[1]/if[1]/else[1]/ass...	Wed Feb 02 17:24:47 CET 2...	false	Waiting	true
Activity_Complete	/process/sequence[1]/if[1]/else[1]/ass...	Wed Feb 02 17:24:47 CET 2...	false	Completed	true
Activity_Executed	/process/sequence[1]/if[1]	Wed Feb 02 17:24:47 CET 2...	false	Waiting	true
Activity_Complete	/process/sequence[1]/if[1]	Wed Feb 02 17:24:47 CET 2...	false	Completed	true
Activity_Ready	/process/sequence[1]/assign[2]	Wed Feb 02 17:24:47 CET 2...	false	Ready	true
Activity_Executing	/process/sequence[1]/assign[2]	Wed Feb 02 17:24:47 CET 2...	false	Executing	true
Variable_Modification	/process/variables[1]/variable[2]	Wed Feb 02 17:24:47 CET 2...	false		
Variable_Modification	/process/variables[1]/variable[1]	Wed Feb 02 17:24:47 CET 2...	false		
Activity_Executed	/process/sequence[1]/assign[2]	Wed Feb 02 17:24:47 CET 2...	false	Waiting	true
Activity_Complete	/process/sequence[1]/assign[2]	Wed Feb 02 17:24:47 CET 2...	false	Completed	true
Activity_Ready	/process/sequence[1]/flow[1]	Wed Feb 02 17:24:47 CET 2...	false	Ready	true
Activity_Executing	/process/sequence[1]/flow[1]	Wed Feb 02 17:24:47 CET 2...	false	Executing	true
Activity_Ready	/process/sequence[1]/flow[1]/wait[2]	Wed Feb 02 17:24:47 CET 2...	false	Ready	true
Activity_Executing	/process/sequence[1]/flow[1]/wait[2]	Wed Feb 02 17:24:47 CET 2...	false	Executing	true
Activity_Ready	/process/sequence[1]/flow[1]/assign[1]	Wed Feb 02 17:24:47 CET 2...	false	Ready	true
Activity_Executing	/process/sequence[1]/flow[1]/assign[1]	Wed Feb 02 17:24:47 CET 2...	false	Executing	true
Variable_Modification	/process/variables[1]/variable[2]	Wed Feb 02 17:24:47 CET 2...	false		
Variable_Modification	/process/variables[1]/variable[1]	Wed Feb 02 17:24:47 CET 2...	false		
Activity_Executed	/process/sequence[1]/flow[1]/assign[1]	Wed Feb 02 17:24:48 CET 2...	false	Waiting	true
Activity_Complete	/process/sequence[1]/flow[1]/assign[1]	Wed Feb 02 17:24:48 CET 2...	false	Completed	true
Activity_Ready	/process/sequence[1]/flow[1]/empty[1]	Wed Feb 02 17:24:48 CET 2...	false	Ready	true
Activity_Executing	/process/sequence[1]/flow[1]/empty[1]	Wed Feb 02 17:24:48 CET 2...	false	Executing	true
Activity_Executed	/process/sequence[1]/flow[1]/empty[1]	Wed Feb 02 17:24:48 CET 2...	false	Waiting	true
Activity_Complete	/process/sequence[1]/flow[1]/empty[1]	Wed Feb 02 17:24:48 CET 2...	false	Completed	true
Activity_Ready	/process/sequence[1]/flow[1]/wait[1]	Wed Feb 02 17:24:48 CET 2...	false	Ready	true
Activity_Executing	/process/sequence[1]/flow[1]/wait[1]	Wed Feb 02 17:24:48 CET 2...	false	Executing	true
Activity_Executed	/process/sequence[1]/flow[1]/wait[2]	Wed Feb 02 17:25:18 CET 2...	false	Waiting	true
Activity_Complete	/process/sequence[1]/flow[1]/wait[2]	Wed Feb 02 17:25:18 CET 2...	false	Completed	true
Activity_Executed	/process/sequence[1]/flow[1]/wait[1]	Wed Feb 02 17:25:18 CET 2...	false	Waiting	true
Activity_Complete	/process/sequence[1]/flow[1]/wait[1]	Wed Feb 02 17:25:18 CET 2...	false	Completed	true
Activity_Executed	/process/sequence[1]/flow[1]	Wed Feb 02 17:25:18 CET 2...	false	Waiting	true
Activity_Complete	/process/sequence[1]/flow[1]	Wed Feb 02 17:25:18 CET 2...	false	Completed	true
Activity_Ready	/process/sequence[1]/reply[1]	Wed Feb 02 17:25:18 CET 2...	false	Ready	true
Activity_Executing	/process/sequence[1]/reply[1]	Wed Feb 02 17:25:18 CET 2...	false	Executing	true
Activity_Executed	/process/sequence[1]/reply[1]	Wed Feb 02 17:25:18 CET 2...	false	Waiting	true
Activity_Complete	/process/sequence[1]/reply[1]	Wed Feb 02 17:25:18 CET 2...	false	Completed	true
Instance_Suspended		Wed Feb 02 17:25:18 CET 2...	false		

Abbildung 30: Auditing der Instanz (2)

Nach diesem Schritt sieht der Reiter Instances in der Apache ODE GUI exakt aus wie nach der Migration in Abbildung 29. Das Prozessmodell, auf das die Instanz verlinkt ist, bleibt *HelloWorld.v2-2* und die Instanz befindet sich wieder im Status SUSPENDED. Der einzige Unterschied ist der Aktivitätszeitstempel, der jetzt die Zeit angibt, zu der die Instanz zuletzt aktiv war.

Anschließend gibt es zwei denkbare Szenarien. Die Instanz könnte wieder auf ein neues Prozessmodell migriert werden oder die Instanz soll in den Status COMPLETED überführt werden. Wenn die Instanz abermals migriert werden soll, läuft die zweite Migration identisch zur ersten Migration ab. In diesem Beispiel ist der Benutzer mit dem experimentellen Erstellen des Prozessmodells fertig und möchte keine weiteren Aktivitäten einfügen. Er wählt die *finish()*-Operation, entweder über den Web Service oder die Apache ODE GUI, woraufhin die Prozessinstanz beendet wird.

In Abbildung 31 ist zu sehen, dass nach dem Instance_Suspended Event die Instanz wieder aktiv wird. Es werden die sequence- und process-Aktivität geschlossen und die Instanz in den Status COMPLETED überführt.

instance251

Event	Source	Timestamp	Persis...	State	Persisted
Activity_Executing	/process/sequence[1]/reply[1]	Wed Feb 02 17:25:18 CET 2...	false	Executing	true
Activity_Executed	/process/sequence[1]/reply[1]	Wed Feb 02 17:25:18 CET 2...	false	Waiting	true
Activity_Complete	/process/sequence[1]/reply[1]	Wed Feb 02 17:25:18 CET 2...	false	Completed	true
Instance_Suspended		Wed Feb 02 17:25:18 CET 2...	false		
Instance_Running		Wed Feb 02 17:27:08 CET 2...	false		
Activity_Executed	/process/sequence[1]	Wed Feb 02 17:27:08 CET 2...	false	Waiting	true
Activity_Complete	/process/sequence[1]	Wed Feb 02 17:27:08 CET 2...	false	Completed	true
Activity_Executed	/process	Wed Feb 02 17:27:08 CET 2...	false	Waiting	true
Activity_Complete	/process	Wed Feb 02 17:27:08 CET 2...	false	Completed	true
Instance_Completed		Wed Feb 02 17:27:08 CET 2...	false		

Abbildung 31: Auditing der Instanz (3)

Im Reiter Instance in der Apache ODE GUI verändert sich der Status der Instanz auf COMPLETED. Der Aktivitätszeitstempel zeigt den Zeitpunkt an, bei dem die Instanz zuletzt aktiv war. Die Buttons „Resume“ und „Finish“ wurden inaktiv gesetzt. Die Instanz ist jetzt beendet und kann nicht weiter ausgeführt oder migriert werden.

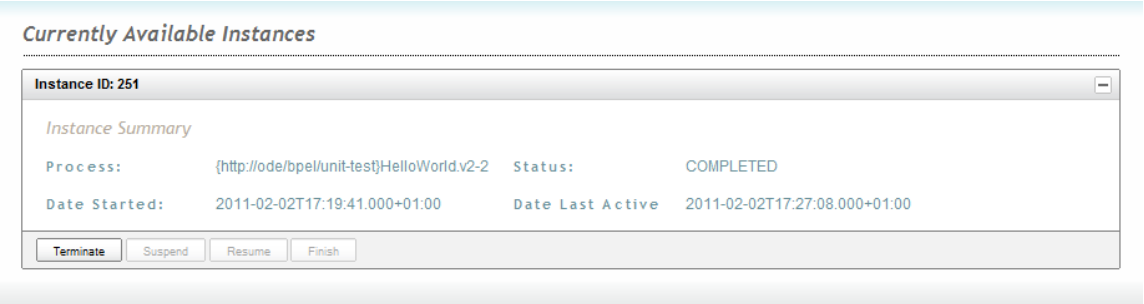


Abbildung 32: Status der Instanz in der Apache ODE GUI (3)

7 Zusammenfassung und Ausblick

Im Rahmen dieser Diplomarbeit wurde eine Möglichkeit untersucht, Workflow-Maschinen an die Ansprüche von Wissenschaftlern anzupassen und eine explorative Workflow-Entwicklung zu ermöglichen. Dazu wurden die Möglichkeiten der Modell-Versionierung und Instanzmigration auf eine neue Version eines Prozessmodells untersucht. Es werden die benötigten Änderungen am Lebenszyklus einer Instanz und am Deployment von Prozessmodellen aufgezeigt und teilweise mehrere Lösungsmöglichkeiten vorgeschlagen. Ein Konzept zur Instanzmigration wurde für die Sprache BPEL für die einzelnen Aktivitäten erarbeitet.

Auf Basis dieser Erkenntnisse wurde die Apache ODE prototypisch um die Deploy New Version-Funktionalität erweitert. Die Deploy New Version-Funktionalität ermöglicht es, mehrere Versionen eines Prozessmodells aktiv zu halten. Des Weiteren werden Prozessinstanzen nach dem Beenden ihrer letzten Aktivität am Leben erhalten und die Möglichkeit geschaffen, Instanzen auf neue Prozessmodell-Versionen zu migrieren. Anhand eines Beispiels wurde gezeigt, wie die Deploy New Version-Funktionalität angewendet werden kann.

Die Untersuchungen im Rahmen dieser Diplomarbeit haben gezeigt, dass BPEL-Prozessmodelle dazu geeignet sind, den wissenschaftlichen Ansprüchen einer explorativen Prozess-Entwicklung gerecht zu werden.

Die im Rahmen dieser Arbeit entwickelte Deploy New Version-Funktionalität kann als Grundlage zur Entwicklung einer Workflow-Maschine, die die Deploy New Version-Funktionalität anbietet oder zur Weiterentwicklung der Apache ODE verwendet werden. Die prototypische Implementierung der sequence-Aktivität kann als Vorlage dienen, alle weiteren BPEL-Aktivitäten zu unterstützen, um die Einschränkungen des gegenwärtigen Prototypens Schritt für Schritt aufzulösen.

Da der Wunsch nach Flexibilität in unserer Gesellschaft immer wichtiger wird, werden auf dem Gebiet der Workflow Technologie weiterhin Entwicklungen stattfinden, um die Flexibilität und Adaptivität von Workflows zu erhöhen. Außerdem wäre eine Entwicklung einer Monitoring-Anwendung, die die Instanzmigration sowie die Änderungen, die an dem Prozessmodell vorgenommen wurden, aufzeigt, denkbar. Diese Monitoring-Anwendung ist notwendig, da ansonsten nicht nachvollzogen werden kann, welche migrierte Instanz welche Aktivitäten ausgeführt hat. Um Ergebnisse zuverlässig auswerten zu können, ist es essentiell wichtig, zu wissen, welche Aktivitäten ausgeführt wurden.

Abbildungsverzeichnis

Verzeichnis der Listings

Quellenverzeichnis

1. Akram et al., Evaluation of BPEL to scientific workflows, Proc. of 6th IEEE International Symposium on Cluster Computing and the Grid, 2006.

2. Wassermann et al., Sedna: A BPEL-based environment for visual scientific workflow modeling, In: Taylor et al. (Eds.), Workflows for e-science – Scientific workflows for grids (Springer, 2007).

3. Sonntag et al., Towards simulation workflows with BPEL: Deriving missing features from GriCoL, In: Alhajj, R.S. (Hrsg); Leung, V.C.M. (Hrsg);

4. M. Reichert and S. Rinderle, On design principles for realizing adaptive service flows with BPEL. Proc. of EMISA 2006, GI Lecture Nots in Informatics, LNI P-95, 2006.

5. A. Fritzler: Migrating WS_BPEL Process Instances -Diplomarbeit Nr. 2966 Uni Stuttgart

6. M. Kern: Enforcement auf laufenden BPEL-Prozessen – Diplomarbeit Nr. 2898 Uni Stuttgart

7. G.Starke, S. Tilkov : SOA_Expertenwissen: Praxis, Methoden und Konzepte serviceorientierter Architektur –– dpunkt-Verlag, 2007

8. F. Leymann, S. Weeawarana, F. Curbera , D. F. Derguson : Web Service Platform Architecture, Prentice Hall, 2005

9. Frank Leymann, Dieter Roller: Production Workflow: Concepts and Techniques, Prentice Hall, 1999

10. M. Böhm, S. Jablonski, und W. Schulze : Workflow-Management - Entwicklung von Anwendungen und Systemen - Facetten einer neuen Technologie -, dpunkt-Verlag, 1997

11. D. Hollingsworth: The Workflow Reference Model. Technical report, Workflow Management Coalition, 1995.

12. T.Erl: Service-oriented architecture: concepts, technology, and design, Prentice Hall, 2005

13. M.E. Stevens : “Service-Oriented Architecture”, Java Web Services Architecture, Morgan Kaufmann, 2003

14. OASIS: “Web Services Business Process Execution Language (WS-BPEL) Version 2.0”, 2007

15. http://www.ode.apache.org/

16. BOS V5.3 - User & Reference Guide – 19 Oct 10

17. P.Dadam, M.Reichert, S. Rinderle-Ma: Prozessmanagementsysteme – Nur ein wenig Flexibilität wird nicht reichen – 2011 Informatik-Spektrum

18. P. Dadam, M.Reichert: ADEPTflex – Supporting Dynamic Changes of Workflows Without Losing Control – 1998 Kluwer Academic Publishers

19. I. Wassink, M. Ooms, P. van der Vet: Designing workflows on the fly using e-BioFlow

20. Lab exercise - WebSphere Process Server V7.0 – Process evolution and instance migration, IBM 2010

21. O. Cline, M. Surya: WebSphere Process Server Versioning: From Design to Production, IBM 2010

22. Sonntag, Mirko; Karastoyanova, Dimka: Concurrent Workflow Evolution. In: Proceedings of the Workshop on Flexible Workflows in Distributed Systems (WiVS), Conference on Communications in Distributed Systems (KiVS), GI-Edition Lecture Notes in Informatics (LNI), 2011. (to appear)

Anhang

I. BPEL

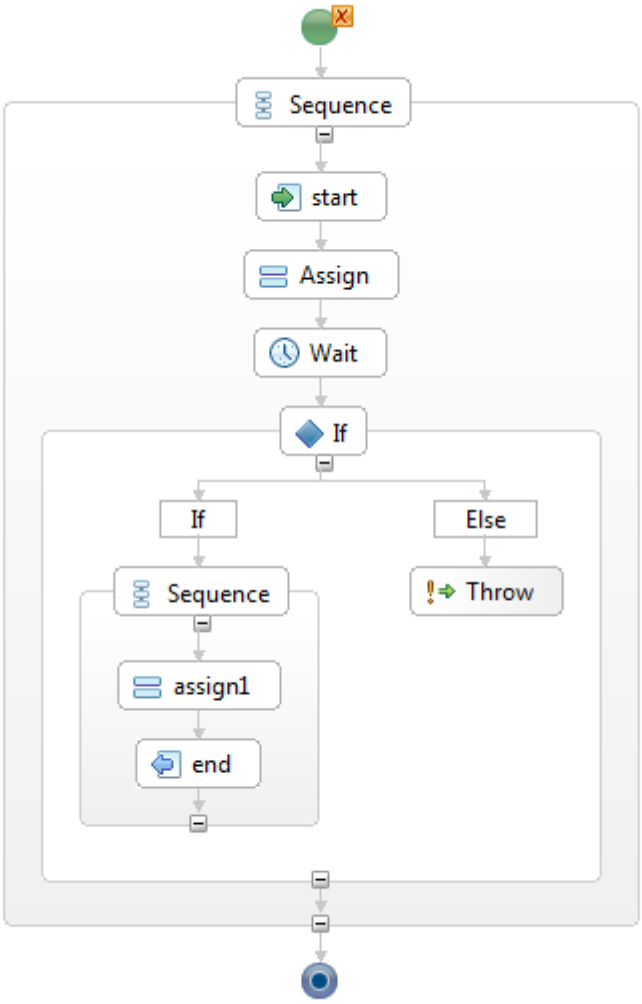


Abbildung 33: Grafisches Modell eines BPEL-Prozesses

```
<process name="HelloWorld2" targetNamespace="http://ode/bpel/unit-test"
    xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
    xmlns:tns="http://ode/bpel/unit-test" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:test="http://ode/bpel/unit-test.wsdl"
 queryLanguage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath2.0"
    expressionLanguage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath2.0"
    xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/executable">

    <import location="HelloWorld2.wsdl" namespace="http://ode/bpel/unit-test.wsdl"
        importType="http://schemas.xmlsoap.org/wsdl/" />

    <partnerLinks>
        <partnerLink name="helloPartnerLink"
 partnerLinkType="test:HelloPartnerLinkType"
            myRole="me" />
    </partnerLinks>

    <variables>
        <variable name="myVar" messageType="test:HelloMessage" />
        <variable name="tmpVar" type="xsd:string" />
        <bpel:variable name="number" type="xsd:integer"></bpel:variable>
    </variables>

    <bpel:faultHandlers>
        <bpel:catchAll>
            <bpel:sequence>
                <bpel:empty name="Empty"></bpel:empty>
            </bpel:sequence>
        </bpel:catchAll>
        <bpel:empty></bpel:empty>
    </bpel:faultHandlers>

    <sequence>

        <receive name="start" partnerLink="helloPartnerLink"
            createInstance="yes" operation="hello" portType="test:HelloPortType"
            variable="myVar">
        </receive>

        <bpel:assign validate="no" name="Assign">
            <bpel:copy>
                <bpel:from>
                    <bpel:literal xml:space="preserve">1</bpel:literal>
                </bpel:from>
                <bpel:to variable="number"></bpel:to>
            </bpel:copy>
        </bpel:assign>

        <bpel:wait name="Wait">
            <bpel:for><![CDATA['PT60S']]></bpel:for>
        </bpel:wait>

        <bpel:if name="If">
            <bpel:condition><![CDATA[($number > 2)]]></bpel:condition>
            <bpel:sequence>
                <assign name="assign1">
                    <copy>
                        <from variable="myVar" part="TestPart" />
                        <to variable="tmpVar" />
                    </copy>
                    <copy>
                        <from>concat($tmpVar,' World')</from>
```

```
                        <to variable="myVar" part="TestPart" />
                    </copy>
                </assign>
                <reply name="end" partnerLink="helloPartnerLink"
 portType="test:HelloPortType"
                    operation="hello" variable="myVar">
                    <bpel:documentation></bpel:documentation>
                </reply>
            </bpel:sequence>
            <bpel:else>
                <bpel:throw name="Throw"
 faultName="tns:ThrowDefaultFaultName"></bpel:throw>
            </bpel:else>
        </bpel:if>
    </sequence>
</process>
```

Abbildung 34: Code eines BPEL-Prozesses

II. MySQL-Schema

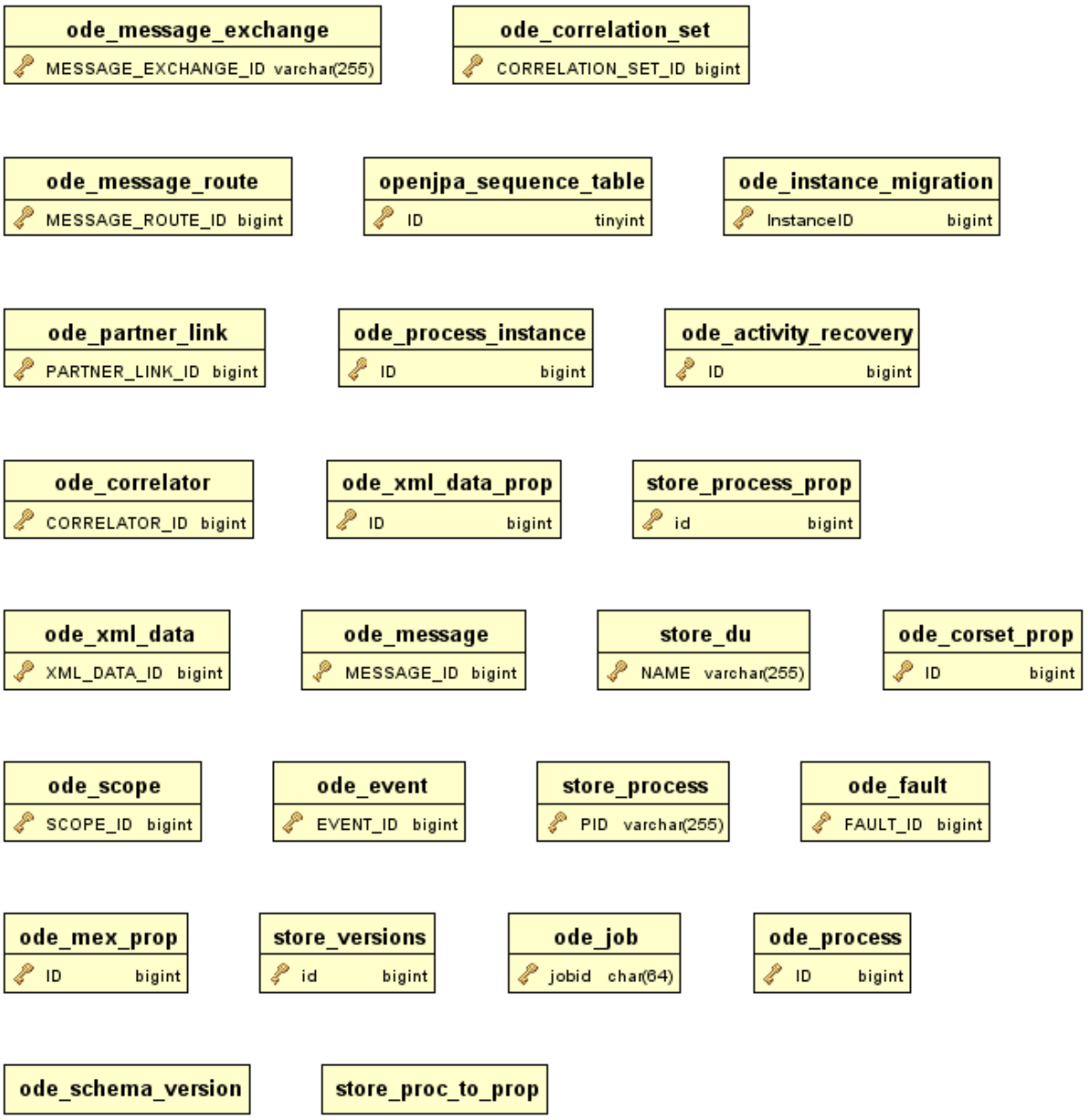
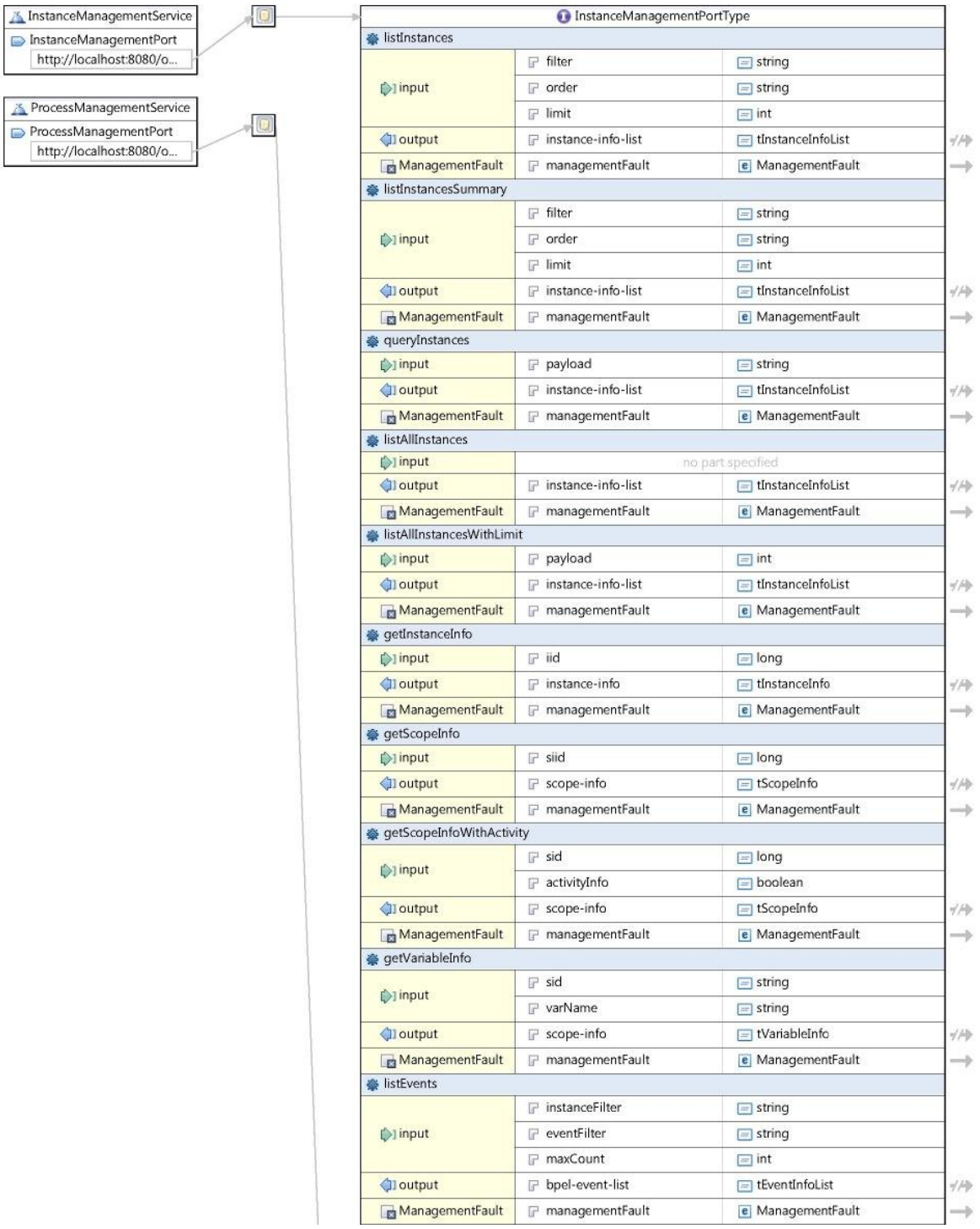


Abbildung 35: Alle Tabellen des Apache ODE MySQL-Schema

III. Process and Instance Management API



listEvents		
input	instanceFilter	string
	eventFilter	string
	maxCount	int
output	bpel-event-list	tEventInfoList
ManagementFault	managementFault	ManagementFault
getEventTimeline		
input	instanceFilter	string
	eventFilter	string
output	dates	listType
ManagementFault	managementFault	ManagementFault
suspend		
input	iid	long
output	instance-info	tInstanceInfo
ManagementFault	managementFault	ManagementFault
resume		
input	iid	long
output	instance-info	tInstanceInfo
ManagementFault	managementFault	ManagementFault
terminate		
input	iid	long
output	instance-info	tInstanceInfo
ManagementFault	managementFault	ManagementFault
finish		
input	iid	long
output	instance-info	tInstanceInfo
ManagementFault	managementFault	ManagementFault
fault		
input	iid	long
output	instance-info	tInstanceInfo
ManagementFault	managementFault	ManagementFault
delete		
input	filter	string
output	list	listType
ManagementFault	managementFault	ManagementFault
recoverActivity		
input	iid	long
	aid	long
	action	string
output	instance-info	tInstanceInfo
ManagementFault	managementFault	ManagementFault
replay		
input	replay	Replay
output	replayResponse	ReplayRespcnse
ManagementFault	managementFault	ManagementFault
getCommunication		
input	getCommunication	GetCommunication
output	getCommunicationResponse	GetCommunicationResponse
ManagementFault	managementFault	ManagementFault

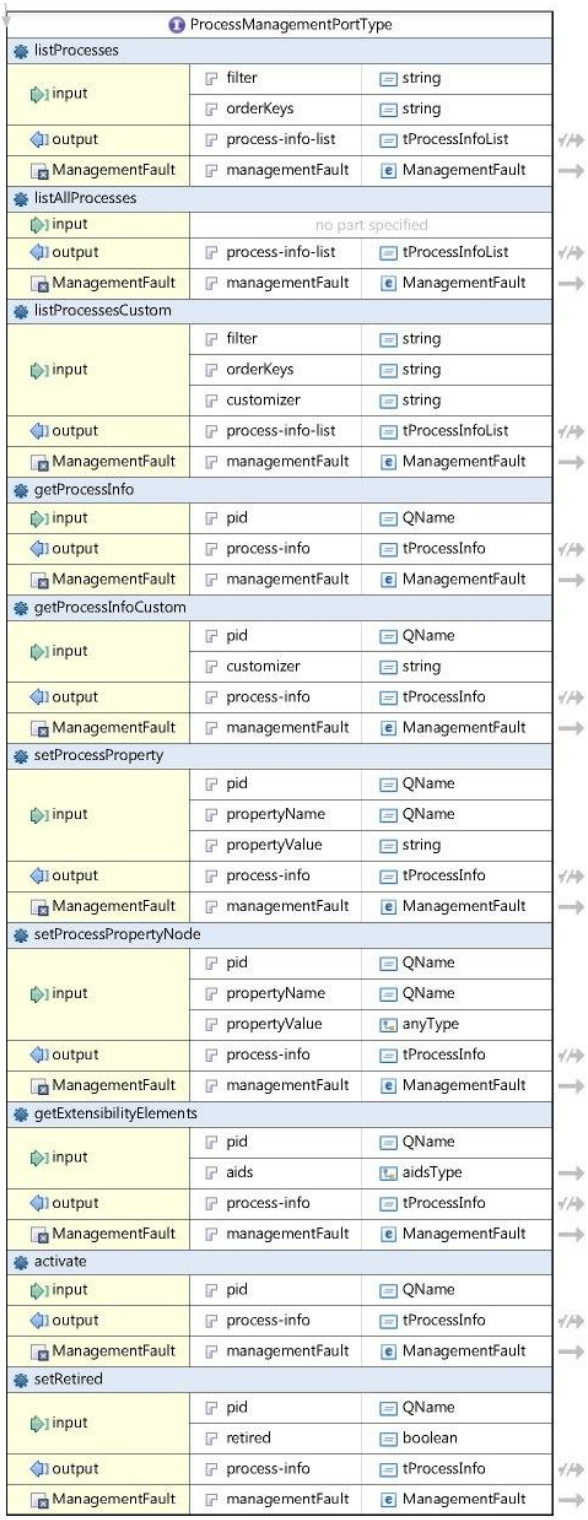


Abbildung 36: graphische Darstellung der Management API

Erklärung

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

Waiblingen, den 7. Februar 2011

_____

Tina Schliemann