Institut für Architektur von Anwendungssystemen
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3144

Ausführung von Workflow-Fragmenten in BPEL

Alex Hummel

Studiengang:	Softwaretechnik
Prüfer:	Jun.-Prof. Dr.-Ing. Dimka Karastoyanova
Betreuer:	Dipl.-Inf. Mirko Sonntag
begonnen am:	13. Januar 2011
beendet am:	15. Juli 2011
CR-Klassifikation:	H.4.1

Inhaltsverzeichnis

Abbildungsverzeichnis

Verzeichnis der Listings

1 Einleitung

Seit einiger Zeit besteht das Bestreben die Workflows modular aufzubauen und dadurch die Steigende Komplexität der Prozesse zu bewältigen, sowie die Wiederverwendung der Prozessfragmenten zu erreichen. Dabei orientiert man sich oft an dem Konzept der Softwarekomponenten, die eine bestimmte Funktionalität darstellen und bei bedarf wiederverwendet werden können.

Traditionell muss ein Prozess vollständig definiert werden, bevor es ausgeführt werden kann. Das verursacht die Starrheit der Prozesse. So kann nur langsam auf sich ändernden Umstände reagiert werden, da die Prozesse aktualisiert, deployt und bei bedarf die laufenden Prozessinstanzen auf die neuen Prozessmodelle migriert werden müssen.

Im Bereich von Business Workflows tritt zusätzlich das Problem auf, dass das Wissen über den Geschäftsprozess über mehrere Teilnehmer verteilt ist, was die Modellierung des Prozesses als Ganzes erschwert. Es können dabei oft nicht alle möglichen Situationen vorhergesehen und somit modelliert werden [ELU10]. Dieser Fakt erfordert von den Workflow Management Systemen die Möglichkeit der Zusammensetzung von Prozessen zur Laufzeit. Somit wäre es möglich, das lokale Wissen als Prozessfragmente zu modellieren und zur Laufzeit zusammenzusetzen, die unvorhergesehene Ereignisse ließen sich durch Hinzufügen neuer Prozessfragmente abdecken.

Im Bereich der Scientific Workflows verwenden die Wissenschaftler Workflows, um die Datenverarbeitung zu automatisieren. Dabei tritt das Problem auf, dass die Wissenschaftler nicht vorhersehen können, welche Schritte ab einem bestimmten Punkt im Prozess vorgenommen werden müssen. Sie brauchen eine Möglichkeit, die zur Laufzeit erzeugten Daten zu analysieren, und abhängig von den Daten die nächsten Schritte im Prozess zu modellieren, sowie die Prozessausführung auf dem vervollständigten Prozessmodell fortzusetzen [WOd09] [TDGS06]. Eine mögliche Lösung dieser Probleme ist Teile von Prozessen als separate Prozessfragmente zu modellieren und diese in einem Workflow Management System zur Laufzeit zu einem Prozess zusammensetzen lassen, was in dieser Diplomarbeit behandelt wird.

1.1 Verwandte Arbeiten

Heute existieren mehrere Ansätze, um den oben genannten Problemen entgegenzuwirken. Die *BPEL-SPE* [BPE05] Erweiterung von BPEL ermöglicht die Modularisierung und Wiederverwendung von Prozessfragmenten als Subprozesse. Diese können innerhalb von Prozessen als inline Subprozesse, oder als separate Subprozesse definiert werden, die aus

unterschiedlichen Prozessen aufgerufen werden können. Der Schwerpunkt dabei liegt auf der Kontrolle des Lebenszyklus des Subprozesses, es ist jedoch nicht vorgesehen die aufzurufende Subprozesse zur Laufzeit zu bestimmen. Die aufgerufene Subprozesse müssen einen bestimmten Porttyp implementieren, dieser wird zur Build-Time bestimmt.

In [ATEA06] wird ein Konzept von *Worklets* vorgestellt. *Worklets* stellen Subprozesse dar, und werden genutzt um die passende Logik für die im Prozess definierten abstrakten Aktivitäten zu implementieren. *Worklets* werden mit Informationen erweitert, die den Kontext beschreiben in dem diese ausgewählt werden sollen. Die Auswahl von den auszuführenden *Worklets* erfolgt zur Laufzeit abhängig von dem Kontext des Prozesses. Dieses Konzept erlaubt Evolution von Prozessen ohne die Veränderung des Prozessmodells. Die *Worklets* jedoch stellen vollständige Prozesse dar, was die im Konzept erreichte Flexibilität der Prozesse einschränkt. Eine andere Möglichkeit Flexibilität der Prozesse zu erreichen ist in [HBR08] beschrieben. Dabei werden im Prozess Punkte definiert an denen der Prozess verändert werden kann. Die möglichen Veränderungen des Prozesses werden als *Options* bezeichnet und beschreiben die prozessverändernde Operationen wie Einfügen neuer Aktivitäten, Löschen von Aktivitäten u.ä. Die *Options* haben den Zweck Varianten eines Prozesses durch die Abweichungen vom Standardprozess zu beschreiben. Obwohl dieser Ansatz die Veränderung des Prozesses zur Laufzeit erlaubt, bilden die *Options* keine eigenständigen Prozessfragmente. In [CF04] wird ein Konzept beschrieben, bei dem die Aktivitäten mit Aspekten versehen werden. Zur Laufzeit kann die Ausführung des Prozesses an diesen Punkten unterbrochen werden um den Prozess durch weitere Aktivitäten zu erweitern. Hier vorgestelltes Konzept dient nur der Erhöhung der Flexibilität der Prozesse, adressiert jedoch keine Lösungen bezüglich der Wiederverwendbarkeit von Prozessfragmenten.

In [EUL09] wird ein Konzept von Prozessfragmenten vorgestellt. Es wird davon ausgegangen, dass das Wissen über den Prozess über mehrere Personen verteilt ist und jeder Prozessfragment das lokale Wissen über den Prozess darstellt. Diese Fragmente werden zusammengesetzt um den vollständigen Prozess zu erhalten. Dieses Konzept wird in [ELU10] um backward und forward recovery Strategien erweitert. In [SAL+10] werden Prozessfragmente benutzt um die Compliance von den Prozessen nachzuweisen. Dafür wurde eine Erweiterung von BPEL vorgestellt, die die Compliance-Fragmenten beschreibt. In [Tel10] wurde ein Metamodell für die Prozessfragmente ausgearbeitet das auf Graphen basiert und Regeln für die Navigation von Prozessfragmenten definiert.

Als Grundlage dieser Diplomarbeit werden die Arbeiten [EUL09], [ELU10],[SAL+10] und [Tel10] verwendet.

Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Grundlagen beschreibt die Grundlagen, auf den diese Arbeit aufbaut.

Kapitel 3 – Konzept stellt das Lösungskonzept für die im Kapitel 1.2 gestellte Aufgabe vor.

Kapitel 4 – Übersicht über Apache ODE gibt einen Überblick über *Apache ODE*, das zu erweiternde WFMS.

Kapitel 5 – Umsetzung stellt die Umsetzung des Lösungskonzept vor.

Kapitel 6 – Anwendungsbeispiel beschreibt einen wissenschaftlichen Workflow, der verwendet wurde um das Lösungskonzept zu prüfen.

Kapitel 7 – Zusammenfassung und Ausblick fasst die Ergebnisse der Arbeit zusammen und stellt Anknüpfungspunkte vor.

1.2 Aufgabenstellung

Das Ziel dieser Diplomarbeit ist BPEL so zu erweitern, dass die Ausführung von unvollständigen Prozessen (Prozessfragmenten) ermöglicht wird. Der ausgeführte Prozess wird zur Laufzeit durch Ankleben weiterer Prozessfragmente vervollständigt. Dabei wird von einem Benutzer entschieden, welche Prozessfragmente wie an den ausgeführten Prozess geklebt werden. Konzeptionell baut die Arbeit auf [Tel10] auf. Eine Abbildung der graph-basierten Konzepte auf BPEL und eine daraus resultierende Erweiterung der Konzepte ist allerdings nötig. Die in [ELU10] beschriebenen Transaktionskonzepte werden in dieser Diplomarbeit nicht berücksichtigt. Außerdem soll eine bestehende BPEL-Engine [ODEa] erweitert werden um das Lösungskonzept an einem wissenschaftlichen Workflow zu prüfen.

2 Grundlagen

In diesem Kapitel werden Technologien Vorgestellt, die für das Verstehen dieser Diplomarbeit notwendig sind.

2.1 Workflows und Workflow Management Systeme

In unserer Welt gibt es eine große Anzahl von Vorgängen, die nach bestimmten Regeln ablaufen. In der Business-Welt werden viele Vorgänge immer wieder wiederholt. Diese Vorgänge sind mit der Abwicklung bestimmter Arbeitsschritten verbunden um ein definiertes Ziel zu erreichen. Die Workflow-Technoligie ist darauf ausgerichtet, diese Vorgänge zu automatisieren. Vorgang wird dabei als Prozess bezeichnet. Ein Prozessmodell beschreibt das Muster und mit ihm verbundene Regeln, denen ein Prozess folgt. Ein Prozessmodell kann instantiiert werden. Abhängig von den Daten dieser Prozessinstanz, wird durch das Prozess navigiert, und die Arbeitsschritte, die gemacht werden müssen, werden durchgeführt. Ein Teil der auszuführenden Arbeitsschritte kann dabei von den Menschen und ein anderes Teil von Computern ausgeführt werden.

2.1.1 Dimensionen eines Workflows

Ein Prozess besitzt drei Dimensionen, diese definieren *wer*, *was* und *womit* während einer bestimmten Arbeitseinheit (einer Aktivität) ausführt (Abbildung 2.1) [LRoo].

- Die *Prozesslogik*-Dimension beschreibt was und in welcher Reihenfolge ausgeführt werden soll.
- Die *Organisationsstrukturen*-Dimension beschreibt die an der Organisation beteiligten Abteilungen, Personen und Rollen. Die Organisationsstrukturen werden mit den einzelnen Aktivitäten mit Hilfe von Suchanfragen assoziiert. Diese Suchanfragen definieren, welche Personen oder Computer eine bestimmte Aktivität ausführen dürfen.
- Die *IT-Infrastruktur*-Dimension definiert womit eine Aktivität ausgeführt werden soll. Dazu gehören die Ressourcen und Programme mit deren Hilfe eine Aktivität ausgeführt werden soll.

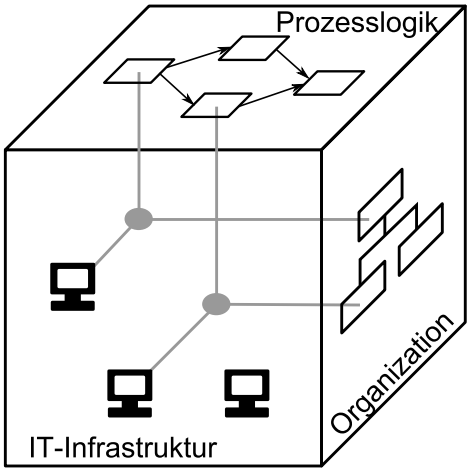

Abbildung 2.1: Dimensionen eines Workflows [LRoo].

2.1.2 Grundlagen von Worflow Management Systemen

Software, die den Ablauf von Prozessen steuert wird *Workflow Management System (WFMS)* genannt.

Zu den Hauptkomponenten eines Workflow Management Systems zählen das *Metamodell*, *Design-Komponente*, *Laufzeit-Komponente* und die *Datenbank* (Abbildung 2.2) [LRoo].

- **Metamodell** definiert die Konstrukte und die dazugehörigen Operationen, die vom WFMS unterstützt werden. Dazu gehören die allgemeine Struktur der unterstützten Prozessmodelle und die Operationen, die auf einer Prozessinstanz ausgeführt werden können.
- **Design-Komponente** ermöglicht dem Benutzer die im Metamodell erlaubten Konstrukte zu definieren. Dazu gehören das *Prozessmodell*, die *Organisationsstruktur*, und die *IT-Aspekte*.
- **Laufzeit-Komponente** dient der Ausführung der im Metamodell definierten Operationen wie z.B. das Erstellen einer Prozessinstanz und das Navigieren durch das Prozessmodell.
- **Datenbank** ist für die Speicherung der von der Design- und der Laufzeit-Komponente verwalteten Daten verantwortlich.

Damit ein Prozess von einem WFMS ausgeführt werden kann, wird eine maschinenlesbare Sprache benötigt, die den auszuführenden Prozess beschreiben würde. In dieser Diplomarbeit wurde *Apache ODE* verwendet, die die Ausführung von den in *BPEL* [BPEo7] beschriebenen Prozessen unterstützt. Aus diesem Grund wird im Kapitel 2.4 ein kurzer Überblick über *BPEL* geboten.

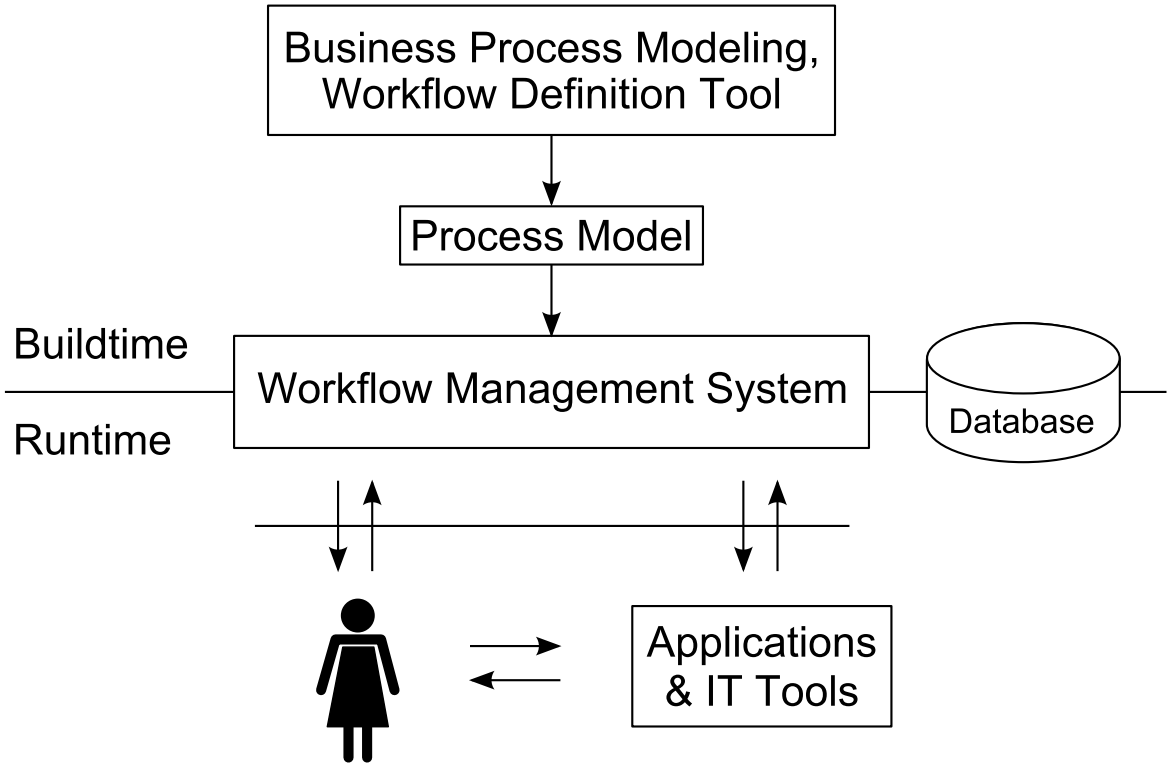

Abbildung 2.2: WFMS Architektur [LRoo].

2.2 Service Oriented Architecture (SOA)

Damit ein WFMS erfolgreich in der Praxis eingesetzt werden kann, muss es mit Programmen kommunizieren können unabhängig von der Plattform, auf den sie laufen, sowie unabhängig von den Programmiersprachen, in den diese implementiert sind. Eine Lösung dafür bietet *Service Oriented Architecture* [STF+10]. Eine wichtige Eigenschaft von *SOA* ist die *lose Kopplung*, d.h. die Dienste werden dynamisch zur Laufzeit gesucht und eingebunden, dabei ist das wissen über die aufzurufenden Services während Implementierung nicht notwendig.

In der *SOA* gibt es im wesentlichen drei Einheiten, die an der Kommunikation teilnehmen.

- **Dienstnutzer** hat das Ziel eine Komponente zu finden, die die von ihm benötigte Funktionalität anbietet, sowie die Funktionalität der gefundenen Komponente aufzurufen.
- **Dienstanbieter** bietet bestimmte Dienste in Form von Komponenten an.
- **Dienstverzeichnis** ermöglicht das Registrieren und Auffinden von Diensten/Komponenten.

Das Zusammenspiel von diesen drei Einheiten in der *SOA* ist auf der Abbildung 2.3 dargestellt. Der *Dienstanbieter* möchte, dass sein Dienst von möglichst vielen Nutzern verwendet

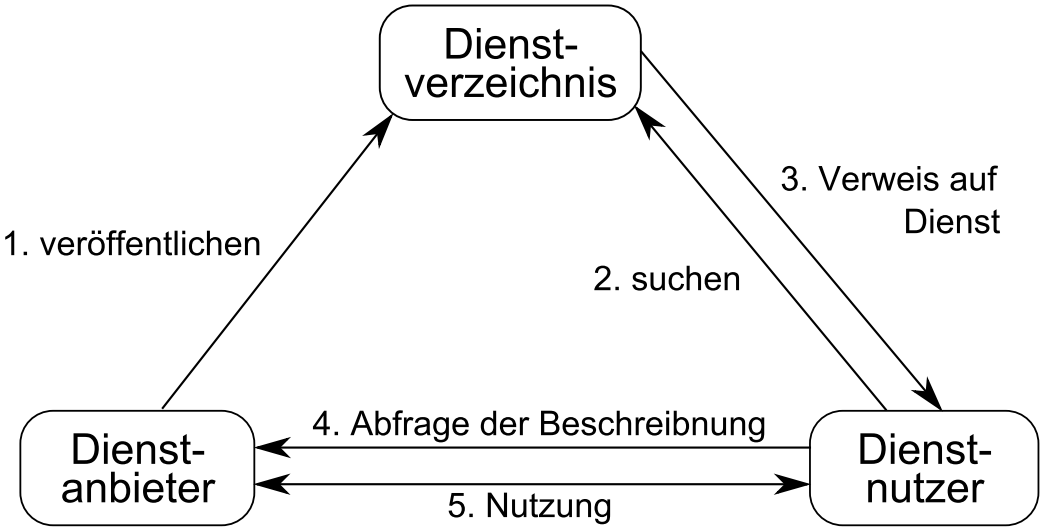

Abbildung 2.3: SOA Dreieck. An [STF+10] angelehnt.

wird. Zu diesem Zweck registriert er seinen Dienst bei einem *Dienstverzeichnis*. Ein *Dienstnutzer* sucht nach einem Dienst, der bestimmte Kriterien erfüllen soll. Dazu fragt er einen entsprechenden Dienst bei dem *Dienstverzeichnis* an. Das *Dienstverzeichnis* liefert einen Verweis auf den entsprechenden Dienst zurück. Der *Dienstnutzer* fragt bei dem *Dienstanbieter* die Beschreibung des Dienstes an. Die Beschreibung enthält dabei die angebotenen Operationen sowie die Art wie diese aufgerufen werden können. Mit diesem Wissen ist dem *Dienstnutzer* nun möglich den gefundenen Dienst aufzurufen.

Um die Plattformunabhängigkeit und Sprachunabhängigkeit zu ermöglichen ist *SOA* auf offene Standards angewiesen. Diese Standards sollen die Schnittstellenbeschreibungssprachen und die Kommunikation zwischen den oben genannten Einheiten definieren [STF+10].

2.3 Web Services

Web Services sind eine Umsetzung der *SOA*. *W3C*[1] Definiert Web Service als *"A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards."* [WSA04].

Ein *Web Service* entspricht dem Dienst in der *SOA*, der eine bestimmte Funktionalität kapselt, die Schnittstellenbeschreibung erfolgt oft in *WSDL* [WSD01], das eine XML-basierte Sprache

[1] http://www.w3.org/

darstellt. Damit ein *Web Service* von einem *Dienstnutzer* entdeckt werden kann, wird *Web Service* bei einem *UDDI*-Dienst registriert, der die Funktionalität eines *Dienstverzeichnisses* übernimmt. Die Kommunikation zwischen dem *Dienstnutzer*, der *UDDI* sowie dem *Web Service* erfolgt oft mit Hilfe von *SOAP* [SOA07]. Auf diese Standards wird in folgenden Abschnitten eingegangen.

2.3.1 SOAP

SOAP [SOA07] ist eine Spezifikation des Nachrichtenformats, das auf XML basiert und bei der Kommunikation mit Web Services benutzt wird. Die Nachrichten können dabei mit Hilfe eines fast beliebigen Protokolls übertragen werden.

Eine *SOAP*-Nachricht enthält folgende Elemente:

- **SOAP Envelope** ist das Wurzelelement des XML-Dokuments und dient als ein Briefumschlag für die Nachricht. Dieses enthält *SOAP Header*- und *SOAP Body*-Elemente. Listing 2.1 zeigt einen Beispiel einer SOAP-Nachricht.
- **SOAP Header** ist ein optionales Element und darf nur als erstes Element in *SOAP Envelope* vorkommen. In der SOAP-Spezifikation ist nicht definiert was in *SOAP Header* vorkommen kann, es ermöglicht lediglich die Anreicherung der SOAP-Nachricht mit weiteren Informationen. So können im *SOAP Header* sicherheitsrelevante Informationen enthalten sein.
- **SOAP Body** enthält die Nutzdaten und muss in jeder SOAP-Nachricht vorkommen. Die Nutzdaten müssen dabei im XML-Format vorliegen.

Während der Kommunikation können Fehler entstehen. Damit die an der Kommunikation beteiligten Partner entsprechend auf die auftretenden Fehler reagieren können, definiert *SOAP* ein entsprechendes Nachrichtenformat, das die auftretenden Fehler beschreibt.

Die Nachricht, die einen Fehler beschreibt, darf dabei nur einen *SOAP Fault* Block innerhalb von *SOAP Body* übertragen (Listing 2.2). *SOAP Fault* Block hat zwei verpflichtende Elemente, die den aufgetretenen Fehler beschreiben:

- **Code** Enthält die von SOAP spezifizierte Kodierung der Fehlerquelle.
- **Reason** Enthält die Textuelle Beschreibung des Fehlers.

2.3.2 Web Services Description Language (WSDL)

WSDL [WSD01] ist eine von *W3C* standardisierte XML-basierte Sprache, die benutzt wird um Web Services zu beschreiben. Zur Zeit existieren zwei Versionen von WSDL: WSDL 1.1 [WSD01] und WSDL 2.0 [WSD07]. In dieser Arbeit wird WSDL 1.1 beschrieben und verwendet.

Listing 2.1 Beispiel einer SOAP Nachricht [WCL+05]

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope"
                      xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03/addressing"
                      xmlns:wssec="http://schemas.xmlsoap.org/ws/2002/04/secext"
                      xmlns:wsrm="http://schemas.xmlsoap.org/ws/2003/03/rm">
       <env:Header>
              <wsa:ReplyTo>
                     <wsa:Address>http://business456.com/User12</wsa:Address>
              </wsa:ReplyTo>
              <wsa:To>http://fabrikam123.com/Traffic</wsa:To>
              <wsa:Action>http://fabrikam123.com/Traffic/Status</wsa:Action>
              <wssec:Security>
                     <wssec:BinarySecurityToken ValueType="wssec:X509v3"
                         EncodingType="wssec:Base64Binary">
                            dWJzY3JpYmVyLVBic.....eFw0wMTEwMTAwMD
                     </wssec:BinarySecurityToken>
              </wssec:Security>
              <wsrm:Sequence>
                     <wsu:Identifier>http:fabrikam123.com/seq1234</wsu:Identifier>
                     <wsrm:MessageNumber>10</wsrm:MessageNumber>
              </wsrm:Sequence>
       </env:Header>
       <env:Body>
              <app:TrafficStatus xmlns:env="http://highwaymon.org/payloads">
                     <road>520W</road>
                     <speed>3MPH</speed>
              </app:TrafficStatus>
       </env:Body>
 </env:Envelope>
```

WSDL-Beschreibung eines Web Services besteht aus einem abstrakten und einem konkreten Teil. Der abstrakte Teil beschreibt dabei die Schnittstellen des Web Services und seine Operationen. Der konkrete Teil beschreibt wie der Web Service aufgerufen werden kann. Die Beschreibung der Semantik des Web Services ist jedoch kein Bestandteil der *WSDL* [STF+10].

Die Struktur eines WSDL-Dokuments sieht wie folgt aus. Das XML-Wurzelelement ist *description*. Dieses Element enthält folgende Elemente:

- **documentation** enthält die textuelle Beschreibung des Web Services.
- **types** beschreibt die in den Nachrichten verwendeten Datentypen.
- **message** beschreibt die bei der Kommunikation verwendeten Nachrichten.
- **portType** definiert eine Menge von Operationen, die der Web Service anbietet, sowie referenziert die entsprechenden ein- und ausgehenden Nachrichten.
- **binding** beschreibt welches Protokoll und Datenformat für den Datenaustausch beim Aufruf einer im *portType* beschriebenen Operation verwendet wird.

Listing 2.2 Beispiel einer fehlerbeschreibenden SOAP Nachricht [WCL+05]

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope"
              xmlns:flt="http://example.org/faults">
       <env:Body>
              <env:Fault>
                     <env:Code>
                            <env:Value>env:Receiver</env:Value>
                            <env:Subcode>
                                   <env:Value>flt:BadValue</env:Value>
                            </env:Subcode>
                     </env:Code>
                     <env:Reason>
                            <env:Text>A Fault occurred</env:Text>
                     </env:Reason>
                     <env:Detail>
                            <flt:MyDetails>
                                   <flt:Message>Something went wrong at the
                                       Receiver</flt:Message>
                                   <flt:ErrorCode>1234</flt:ErrorCode>
                            </flt:MyDetails>
                     </env:Detail>
              </env:Fault>
       </env:Body>
</env:Envelope>
```

- **port** definiert einen Endpunkt für die Kommunikation mit dem Web Service und spezifiziert die Adresse des Services.
- **service** beschreibt die Menge der *port*s des Web Services.

2.3.3 Web Service Verzeichnisdienste

Falls eine *SOA* viele Dienste umfassen sollte, wird ein Verzeichnisdienst notwendig. Dieser erlaubt Dienste im Verzeichnis über standardisierte Schnittstellen zu suchen. In diesem Kapitel werden *Universal Description, Discovery and Integration (UDDI)* und *Web Services Inspection Language (WS-Inspection)* kurz vorgestellt. *UDDI* ist für wenige zentrale *Web Service Verzeichnisse* und viele *Web Service Anbieter* konzipiert, *WS-Inspection* ist für viele kleinere dezentrale *Verzeichnisse* und einen oder wenige *Web Service Anbieter* bestimmt (Abbildung 2.4) [STF+10].

UDDI

UDDI ermöglicht eine zentralisierte Verwaltung, Registrierung und Auffinden von Web Services im Web. Ein Web Service Anbieter kann seinen Web Service in einem *UDDI-Verzeichnis* registrieren lasen, dazu lädt der Web Service Anbieter die WSDL-Beschreibung seines Web Services auf den *UDDI-Verzeichnis* hoch. Ein Benutzer kann im *UDDI-Verzeichnis* nach den passenden Web Services suchen und deren WSDL-Beschreibungen abfragen. Die

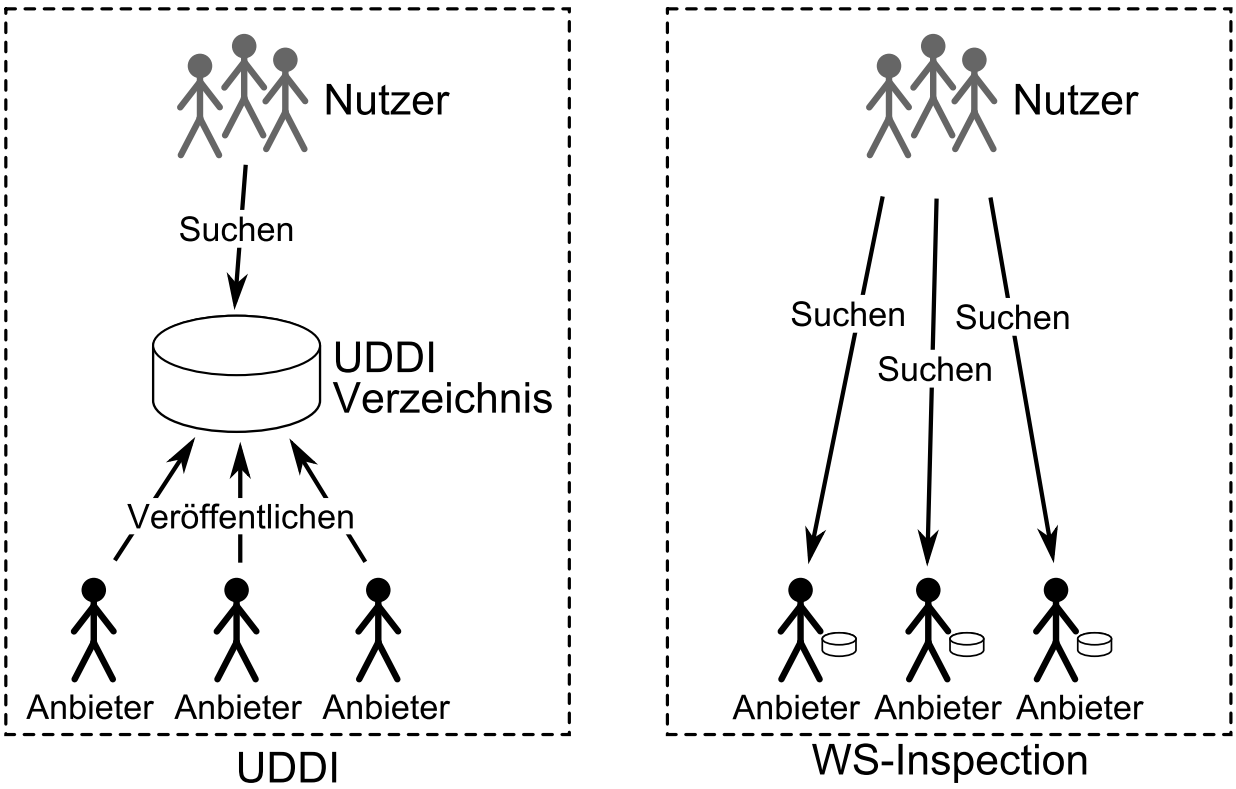

Abbildung 2.4: Web Service Verzeichnisdienste

Funktionalität eines *UDDI-Verzeichnisses* wird dabei als Web Service bereitgestellt und erlaubt das Auffinden von Web Services für Menschen und Anwendungen.

Intern enthält ein *UDDI-Verzeichnis* vier Haupttabellen in der UDDI-Datenbank.

- **White Pages** erlauben den Unternehmen, die Web Services anbieten, Informationen über sich selbst zu veröffentlichen. Anhand dieser Informationen kann der potentielle Web Service Benutzer eine Entscheidung treffen, ob er Web Services dieses Unternehmens nutzen möchte.
- **Yellow Pages** ermöglichen die Suche nach Web Services, falls der Name des Web Service Anbieters nicht bekannt ist, jedoch die Kategorie zu der der gesuchte Web Service Anbieter gehört.
- **Green Pages** ermöglichen die Suche nach dem Web Service falls weder der Name des Web Service Anbieters noch die Kategorie, zu der er gehört, dem Benutzer bekannt ist. Mit *Green Pages* kann der Benutzer die Web Services manuell durchsuchen.
- **Service Type Registration** speichert die Informationen über die verfügbaren Web Services in der maschinenlesbaren Form. Es ermöglicht die Suche nach den Web Services für die Anwendungen.

Der *UDDI*-Ansatz bringt jedoch einige Probleme mit sich. Da es potentiell jeder im *UDDI-Verzeichnis* seine Web Services registrieren lassen kann, ist es oft unbekannt wie die Qualität des registrierten Web Services ist, wie die Bezahlung für die Benutzung des Web Services erfolgt und wer die Verantwortung für den Web Service trägt [STF+10].

WS-Inspection

WS-Inspection-Ansatz versucht die bei der *UDDI* aufgeführten Probleme zu beseitigen. *WS-Inspection* ist dokumentenbasiert. Die von einem Anbieter angebotenen Web Services werden auf der entsprechenden Web-Seite des Anbieters unter einem definiertem Dateinamen gespeichert. Der Benutzer kann auf der Web-Seite des Anbieters mit Hilfe eines Web-Browsers nach den gewünschten Web Services suchen. Als Ergebnis der Suche wird eine Liste der passenden Web Services sowie deren WSDL-Beschreibungen geliefert.

Ein WS-Inspection Dokument besteht aus beliebig vielen *Service*- und *Link*-Elementen. Ein *Service*-Element enthält die Beschreibung des Web Services, sowie die Beschreibung, wo dieser zu finden ist. Ein *Link*-Element enthält einen Verweis auf eine externe Datenquelle. Mit Hilfe von *Link*-Elementen können Hierarchien von den WS-Inspection Dokumenten aufgebaut werden, was für die Kategorisierung der Web Services und deren Verwaltung von nutzen ist [STF+10].

2.4 WS-BPEL

Web Services Business Process Execution Language (WS-BPEL) [BPE07] ist eine XML-basierte Workflow-Sprache, die unter Mitarbeit von *IBM* und *Microsoft* entwickelt wurde. *WS-BPEL* ermöglicht die Modellierung von Prozessen, deren Teilfunktionalitäten als Web Services implementiert sind. In dieser Arbeit wird *WS-BPEL 2.0* beschrieben und verwendet.

Ein grundlegender Bestandteil von BPEL sind die Aktivitäten. Diese kann man in *Basic Activities* und *Structured Activites* unterteilen. Die *Basic Activities* beschreiben elementare Schritte eines Prozesses. Die *Structured Activities* können weitere Aktivitäten enthalten und den Kontrollfluss zwischen den enthaltenen Aktivitäten definieren.

Zu den *Basic Activities* gehören:

- <receive>
- <reply>
- <invoke>
- <assign>
- <validate>
- <throw>

- <rethrow>
- <empty>
- <wait>
- <exit>
- <compensate>
- <compensateScope>
- <extensionActivity>

Zu den *Structured Activities* gehören

- <sequence>
- <flow>
- <if>
- <while>
- <repeatUntil>
- <forEach>
- <pick>
- <scope>

BPEL ist aus den Sprachen *Web Service Flow Language (WSFL)* von *IBM* und *XLANG* von *Microsoft* entstanden. Prozesse in *WSFL* werden als Graphen modelliert. Die Aktivitäten stellen dabei die Knoten dar und werden mit Links verbunden. Prozesse in *XLANG* werden als eine Folge von sequenziellen und parallelen Blöcken modelliert [WCL+05]. BPEL vereinigt diese beiden Ansätze und enthält die <flow> Aktivität um ein Prozess oder einen Teil davon als einen Graph zu modellieren und die <sequence> Aktivität, die eine sequentielle Ausführung von Aktivitäten erlaubt.

BPEL erlaubt keine Zyklen, die mit Hilfe von <flow> modelliert werden könnten. Um die wiederholte Ausführung von Aktivitäten zu unterstützen, enthält BPEL die <while>, <repeatUntil> und <forEach> Aktivitäten. <while> Aktivität erlaubt die wiederholte Ausführung einer Aktivität, solange die in <while> definierte Bedingung *wahr* ist. <repeatUntil> dagegen führt eine Aktivität solange wiederholt aus, bis die in <repeatUntil> definierte Bedingung *wahr* wird. Mit Hilfe der <forEach> Aktivität wird die enthaltene <scope> Aktivität eine bestimmte Anzahl von Iterationen wiederholt ausgeführt. Dafür wird ein Zähler benutzt. Der Startwert und Endwert des Zählers wird in der <forEach> Aktivität definiert. Es gibt dabei eine Möglichkeit die Iterationen parallel auszuführen, dies wird mit Hilfe des *parallel="yes | no"* Attributes definiert.

Die <if> Aktivität erlaubt die Ausführung von einer Aktivität aus einer definierten Menge von den Aktivitäten. Die Auswahl wird anhand von Bedingungen getroffen, die für die einzelnen Aktivitäten aus der oben genannten Menge definiert werden.

Die Aktivitäten <recevie>, <reply>, <invoke> und <pick> sind für die Kommunikation zuständig. <receive> und <reply> Aktivitäten kommen oft als ein Paar vor, um einen Synchronen Prozessaufruf zu implementieren. Außerdem kann mit dem Empfang einer Nachricht durch die <receive> Aktivität eine Prozessinstanz erstellt werden, dafür wird diese Aktivität mit dem Attribut *createInstance="yes"* versehen. Falls eine <receive> Aktivität keine Prozessinstanz erstellt (*createInstance="no"*), dann blockiert diese Aktivität den Kontrollfluss bis diese Aktivität eine Nachricht erhält. Die <invoke> Aktivität dient dem Aufruf eines Web Services. Dabei kann der Web Service sowohl synchron als auch asynchron aufgerufen werden.

Die <pick> Aktivität ist für das Reagieren auf Ereignisse vorgesehen und wartet auf eine Nachricht aus einer Menge möglichen Nachrichten. Diese Aktivität wird abgeschlossen falls eine Nachricht eingeht und eine mit dieser Nachricht assoziierte Aktivität abgeschlossen wird, oder bis eine gewisse Zeit vergeht (ein in der <pick> Aktivität definiertes Timeout). Genau so wie die <receive> Aktivität kann auch die <pick> Aktivität mit dem Attribut *createInstance="yes"* versehen werden, damit bei einer eingehenden Nachricht eine neue Prozessinstanz erstellt wird.

Die im Prozess verwendeten Daten werden in Variablen gespeichert. Mit Hilfe der <assign> Aktivität lassen sich die Variablen initialisieren und Werte von Variablen sowie den *Partner Links* kopieren. Um einen Variablenwert auf die Übereinstimmung mit der dazugehörigen XSD- bzw. WSDL-Typdefinition zu prüfen kann die <validate> Aktivität benutzt werden.

Die <throw> Aktivität löst eine Fehlerbehandlung aus, dabei muss der aufgetretene Fehler in dieser Aktivität angegeben werden, damit die Fehlerbehandlung des angegebenen Fehlers aufgerufen werden kann. Falls ein abgefangener Fehler weiter geworfen werden soll, wird die Aktivität <rethrow> benutzt.

Die <compensate> Aktivität wird benutzt, um die Kompensation aller im aktuellen <scope> enthaltenen und abgeschlossen <scope> Aktivitäten zu starten. Mit Hilfe von <compensateScope> Aktivität wird <compensationHandler> einer bestimmten abgeschlossenen <scope> Aktivität innerhalb der aktuellen <scope> Aktivität ausgeführt. Die Aktivitäten <compensate> und <compensateScope> dürfen nur innerhalb von Fault Handlern verwendet werden.

Mit Hilfe der <wait> Aktivität lässt sich der Kontrollfluss für eine bestimmte Zeit oder bis zu einem definierten Zeitpunkt anhalten. Die Information über die Zeitdauer bzw. den Zeitpunkt wird in den Kindelementen <for> bzw. <until> dieser Aktivität angegeben.

Die <empty> Aktivität besitzt keine Logik die ausgeführt werden soll und entspricht einer leeren Anweisung. Diese Aktivität kann in der <flow> Aktivität zur Synchronisierung verwendet werden oder z.B. bei einer Fehlerbehandlung, falls der aufgetretene Fehler abgefangen und ignoriert werden soll.

Die <exit> Aktivität wird benutzt um eine Prozessinstanz sofort zu beenden ohne *terminationHandler*, *faultHandler* und *compensationHandler* auszuführen.

Die <scope> Aktivität erlaubt für eine Aktivität (die Aktivität kann auch eine *Structured Activity* sein) Kontext zu definieren. Zu dem Kontext gehören *Variablen*, *PartnerLinks*, *MessageExchanges*, *CorrelationSets*, *EventHandlers*, *FaultHandlers*, *CompensationHandler* und ein *TerminationHandler*.

Mit Hilfe von <faultHandlers> Elements werden die im Prozess auftretenden Fehler abgefangen. Dabei kann für jeden Fehlertyp eine eigene Fehlerbehandlung definiert werden. Die Fehlerbehandlung ist darauf ausgerichtet, die teilweise ausgeführte Arbeit innerhalb von <scope> Aktivitäten rückgängig zu machen [BPE07]. *compensationHandler* Elemente werden benutzt um die Schritte zu definieren, welche die in der ausgeführten <scope> Aktivität erledigte Arbeit kompensieren.

terminationHandler erlauben innerhalb von <scope> Aktivitäten einen definierten Zustand zu erreichen falls eine Prozessinstanz terminiert wird.

Mit Hilfe von <eventHandlers> Elements können zu einer <scope> Aktivität Ereignisse definiert werden, die parallel zu der <scope> Ausführung verarbeitet werden. Die Ereignisse können dabei die eingehenden Nachrichten und Timeout-Ereignisse sein.

Das Element <messageExchange> wird Benutzt um die Relation zwischen Aktivitäten die eine Nachricht empfangen und <reply> Aktivitäten zu verdeutlichen. Es ist notwendig im Falle, wenn mehrere Paare von nachrichtempfangenden Aktivitäten mit <reply> Aktivitäten auftreten können, z.B. wenn mehrere Paare von <receive> <reply> Aktivitäten auf dem selben Partner Link definiert sind, dieselbe Operation verwenden und parallel ausgeführt werden.

Damit die in einem Prozessmodell eingehenden Nachrichten zu den dazugehörigen Prozessinstanzen von einem WFMS geleitet werden können, wurde in BPEL <correlationSet> Konstrukt eingeführt. Die Auswahl der zu der Nachricht gehörenden Prozessinstanz wird aufgrund des Inhalts der Nachricht getroffen. Dafür werden in den Nachrichten bestimmte Felder vorgesehen. Diese Felder enthalten Informationen, die die eindeutige Bestimmung der Prozessinstanz ermöglichen. Als Beispiel für ein solches Feld in der Nachricht kann eine Bestellnummer bei der Interaktion mit einem Online-Shop sein. Diese wird beim Kauf eines Produkts erstellt und kann für das Abbestellen des gekauften Produkts verwendet werden. Solche Felder werden in BPEL abstrakt als <property> Elemente deklariert. Damit so ein Feld innerhalb einer eingehenden Nachricht gefunden werden kann, wird ein dazugehöriger <propertyAlias> definiert. Dieses Element definiert, wie das gesuchte Feld in der Nachricht gefunden wird, beispielsweise mit Hilfe eines definierten *XPATH* Ausdrucks (Abbildung 2.5).

Bei manchen Nachrichten werden mehrere Felder benutzt, um die dazugehörige Prozessinstanz eindeutig bestimmen zu können, zu diesem Zweck werden <correlationSet> Elemente definiert. Diese enthalten eine Liste der Namen von den beteiligten <property> Elementen und werden mit Hilfe des Names von dem <correlationSet> referenziert. Die <correlationSet> Elemente werden im <correlationSets> Element einer <scope> Aktivität oder des <process> Elements definiert. Im Prozess werden die definierten <correlationSet> Elemente innerhalb von <receive>, <reply>, <invoke> Aktivitäten sowie dem <onMessage> Element der <pick> Aktivität referenziert.

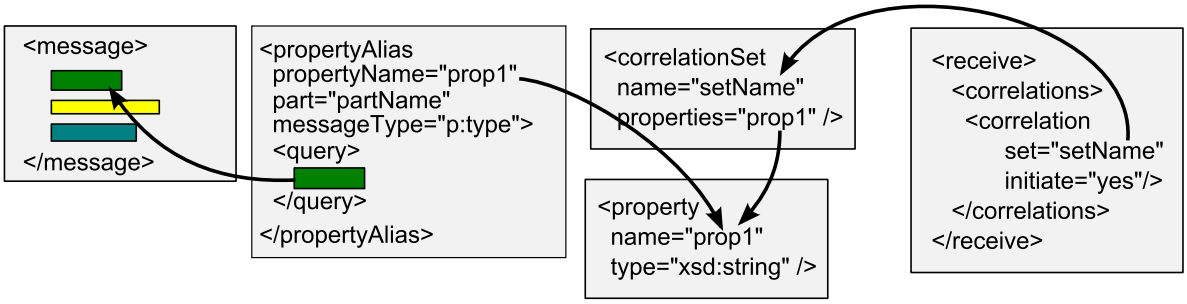

Abbildung 2.5: Zusammenhang zwischen <correlationSet>, <property>, <propertyAlias> und der Nachricht.

Weitere für diese Diplomarbeit wichtige BPEL-Konstrukte sind <partnerLink> und <partnerLinkType>. Mit Hilfe von <partnerLinkType> werden die Beziehungen zwischen den Kommunikationspartnern definiert. Ein <partnerLinkType> definiert Rollen, die ein Kommunikationspartner während der Kommunikation einnehmen kann, sowie die dazugehörigen Porttypen. Da jedoch ein Prozess mit mehreren Partnern kommunizieren kann, die die gleichen Web Services anbieten, spielen diese auch die selben Rollen. Um diese Partner zu unterscheiden wird <partnerLink> verwendet. <partnerLink> Element wird durch das Attribut *name* identifiziert und wird durch einen <partnerLinkType> charakterisiert. Des weiteren wird im <partnerLink> angegeben, welche Rolle bei der Kommunikation der Prozess und welche Rolle der Partner annimmt. Um die Verbindung mit dem Partner aufbauen zu können, wird für den jeweiligen <partnerLink> eine *endpoint reference (EPR)* benötigt. Diese kann währen des Deployments statisch zugewiesen werden, oder zur Laufzeit mit Hilfe der <assign> Aktivität. Im Gegensatz zu den anderen hier vorgestellten BPEL Elementen werden die Elemente <partnerLinkType>, <property> und <propertyAlias> in den WSDL Dateien definiert und nicht in der BPEL-Prozessbeschreibung.

BPEL unterstützt Erweiterungen, so können neue in BPEL nicht vorgesehene Aktivitäten mit Hilfe des <extensionActivity> Elements verwendet werden. <extensionActivity> Element wird als ein Umschlag (Wrapper) für die neue Aktivität benutzt [BPE07].

2.5 Java Persistence API (JPA)

Java Persistence API [KS09] ist ein Framework, das hilft Daten aus Java Objekten in relationale Datenbanken zu speichern. Dabei werden die Klassen, die zu persistierenden Daten modellieren, mit Annotationen versehen. Die Annotationen beschreiben die Abbildung von den Daten auf Datenbanktabellen.

Die Klassen werden auf Entitäten abgebildet und mit der Annotation *@Entity* markiert. Mit Hilfe der *@Id* Annotation wird ein Attribut der Klasse markiert, der die Rolle des Hauptschlüssels in der Tabelle übernehmen soll. Um die Objekte von annotierten Klassen zu persistieren wird ein *Entity Manager* benutzt.

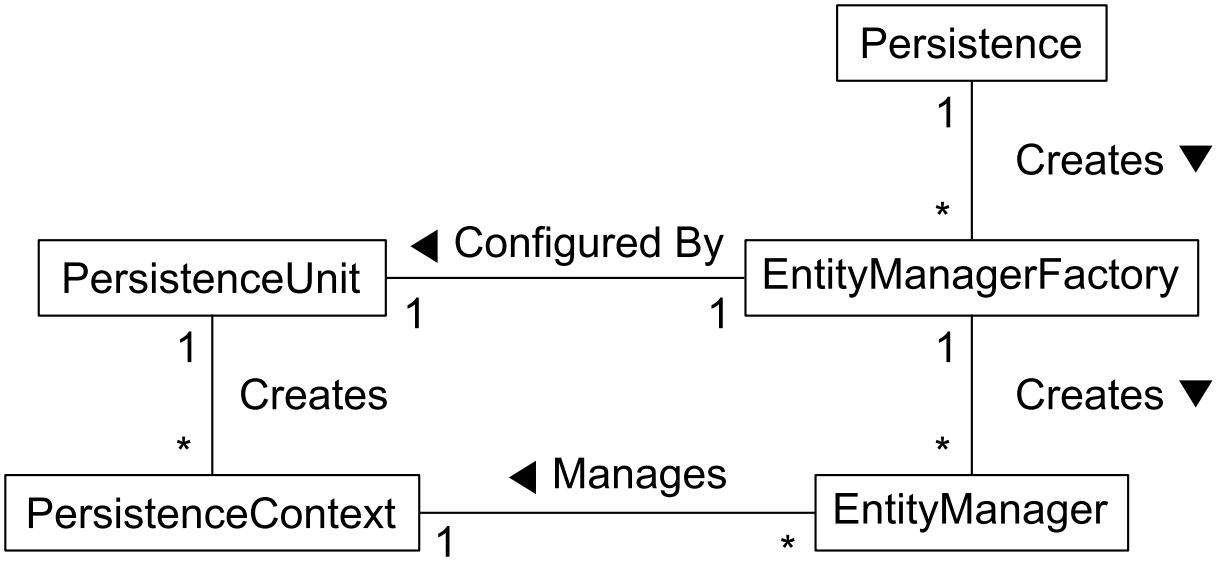

Abbildung 2.6: Beziehungen zwischen den JPA Konzepten [KS09]

Listing 2.3 EntityManagerFactory instantiierung

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("persistance unit
    name");
EntityManager em = emf.createEntityManager();
```

2.5.1 Entity Manager

Die Klasse *EntityManager* verwaltet die bekannten Entitäten und enthält Methoden um die Entitäten zu speichern, zu finden, zu aktualisieren und zu löschen. Eine Entität wird dem *EntityManager* bekannt, indem die Entität dem *EntityManager* als Parameter einer Operation übergeben wird, oder wenn *EntityManager* die Entität aus der Datenbank liest. Die Menge der dem *EntityManager* bekannten Entitäten wird als *persistence context* bezeichnet. Die Klasse *EntityManager* wird von *EntityManagerFactory* erzeugt (Abbildung 2.6). *EntityManager* Klasse braucht Informationen über die zu verwendete Datenbank und die datenmodellierende Klassen, um Daten persisitieren zu können. Diese Daten werden in der Konfigurationsdatei *persistence.xml* als eine *persistence unit* angegeben und werden durch die *EntityManagerFactory* Klasse in der *EntityManager* Instanz gesetzt.

Eine Instanz von *EntityManagerFactory* wird mit einer statischen Methode *createEntityManagerFactory* der Klasse *Persistence* erstellt, dabei wird als Parameter der Name der zu verwendenden *persistance unit* angegeben (Listing 2.3). Die *EntityManager* Instanz wird mit Hilfe der *createEntityManager()* Methode der *EntityManagerFactory* Klasse erzeugt.

Speichern einer Entität wird durch den Aufruf von *persist()* Methode von *EntityManager* initiiert. Bis diese Methode ausgeführt ist, ist die zu speichernde Entität nichts anderes

als ein Java Objekt. Falls ein Problem bei der Speicherung der Entität auftritt, wird eine *PersistenceException* geworfen.

Um eine Entität in der Datenbank zu finden, wird die Methode *find()* verwendet. Als Parameter werden dabei die Klasse der Entität sowie der Hauptschlüssel angegeben. Falls keine Entität gefunden wird, wird *null* zurückgegeben.

Mit Hilfe der Methode *remove()* wird eine Entität aus der Datenbank gelöscht. Diese Entität muss sich jedoch innerhalb von *persistence context* des *EntityManagers* befinden. Um das sicherzustellen, kann vor dem Löschen nach der entsprechenden Entität mit Hilfe der *find()* Methode in der Datenbank gesucht werden.

Um die Daten einer Entität, die sich im *persistence context* des *EntityManagers* befindet, zu verändern, können die Methoden der Entität gehöriger Klasse verwendet werden, z.B. die *setter* Methoden. Ungleich den bis jetzt beschriebenen Methoden wird für die Aktualisierung der Daten einer Entität keine Methode von *EntityManger* verwendet. Das erfordert jedoch, dass die Entität sich bereits im *persistence context* des *EntityManagers* befindet. Alle datenverändernde Operationen von *EntityManger* müssen innerhalb von einem Transaktionskontext aufgerufen werden, ansonsten wird entweder ein Fehler geworfen, oder die Änderungen werden nicht persistiert. Lediglich die *find()* Methode darf außerhalb eines Transaktionskontexts aufgerufen werden, da diese keine Daten verändert.

JPA erlaubt außerdem die Ausführung von *Queries*, diese werden jedoch nicht in *SQL*, sondern in *Java Persistence Query Language (JPQL)* definiert.

Die *Queries* in JPA können sowohl statisch, als auch dynamisch definiert werden. Statische Queries können durch Annotationen definiert werden. Diese Queries werden über deren Namen identifiziert. Die dynamisch definierten Queries werden zur Laufzeit definiert, kosten dafür aber mehr Rechenzeit.

2.5.2 Zusammengesetzte Hauptschlüssel

JPA unterstützt zusammengesetzte Hauptschlüssel. Dafür wird für den zusammengesetzten Hauptschlüssel eine separate Klasse erstellt, die mit der Annotation *@Embeddable* annotiert wird. Diese Annotation führt dazu, dass die Attribute dieser Klasse ein Teil einer Entität werden, die diese Klasse referenziert. Die Klasse, die eine Entität repräsentiert und diesen zusammengesetzten Hauptschlüssel verwendet, muss einen Attribut der oben genannten Klasse enthalten. Dieses Attribut wird mit Hilfe von der Annotation *@EmbeddedId* annotiert um zu kennzeichnen, dass dieses Attribut die Rolle des Hauptschlüssels übernimmt [KS09].

2.6 XSLT

Extensible Stylesheet Language for Transformations (XSLT) [XSL] [Tido8] ist eine flexible Sprache, die die Beschreibung von Transformationen der XML-Dokumenten in etwas Anderes wie

HTML, *PDF*, *JPEG*, oder wiederum in ein XML-Dokument erlauben. Zu diesem Zweck wird ein *XSLT style sheet* definiert, der die Regeln der XML-Transformation beschreibt. Ein *XSLT-Prozessor* übernimmt dann die Transformation des gewünschten XML-Dokuments nach den in dem *XSLT style sheet* definierten Regeln.

2.6.1 XSLT style sheet Struktur

Die *XSLT style sheets* sind XML-Dokumente, die die Regeln der Transformation beschreiben. Ein Style sheet besteht aus folgenden Elementen:

- **<xsl:stylesheet>** ist das Wurzelelement von *XSLT style sheets*, dieses definiert die Version der verwendeten XSLT, sowie den Namensraum *xsl*.
- **<xsl:output>** Element definiert den Typ des erzeugten Dokuments. Es sind folgende vier Werte erlaubt: *xml*, *html*, *xhtml* und *text*.
- **<xsl:template>** Element definiert eine Regel für die XML-Transformation. Mit Hilfe des *match* Attributes wird ein Suchmuster mit Hilfe von *XPath* [XPAb] [XPAa] definiert. Das Suchmuster beschreibt XML-Elemente, bei denen die Regel angewendet werden soll. Die Ausgabe des XML-Elements, die durch das Anwenden dieser Regel erzeugt wird, wird innerhalb des <xsl:template> Elements beschrieben.

Des Weiteren können folgende Elemente als Kinder des Elements <xsl:stylesheet> vorkommen:

- **<xsl:include>** und **<xsl:import>** Elemente werden benutzt um andere Stylesheets zu referenzieren.
- **<xsl:strip-space>** and **<xsl:preserve-space>** Elemente enthalten Listen von Elementen, bei deren Transformationen Leerräume (white spaces) entfernt bzw. beibehalten werden sollen.
- **<xsl:key>** Elemente definieren Schlüssel anhand deren bestimmte XML-Dokumentteile gefunden werden können. Die Schlüssel ähneln den Datenbankindizes.
- **<xsl:variable>** Element definiert eine Variable. Falls dieses Element als Kindelement des <xsl:stylesheet> Elements vorkommt, ist die definierte Variable global. Die Variablen dürfen in XSLT nur ein mal mit den Werten belegt werden.
- **<xsl:param>** Element definiert einen Parameter, der bei der Transformation berücksichtigt werden soll. Falls Parameter als Kindelemente des <xsl:stylesheet> Elements vorkommen, sind diese global. Auf die Werte der Parameter kann wie auf Variablen zugegriffen werden.

2.6.2 Verwendete XSLT Elemente

Die Ausgabe der Transformationsregeln in XSLT erfolgt mit Hilfe von XSLT definierten XML-Elementen. Da XSLT zahlreiche Elemente definiert, werden hier nur die in dieser Arbeit verwendeten Elemente vorgestellt.

- **<xsl:element>** Erzeugt in der Ausgabe ein XML-Element mit dem im Attribut *name* definierten Namen.
- **<xsl:copy-of>** Element kopiert in die Ausgabe durch das Attribut *select* definierte Elemente inklusive ihrer Kindelemente.
- **<xsl:value-of>** Element kopiert die Werte der durch das Attribut *select* definierten Elemente in die Ausgabe.
- **<xsl:text>** Element enthält den Text, der in die Ausgabe geschrieben werden soll.
- **<xsl:if>** Element hat die Semantik einer *if*-Anweisung. Dieses Element enthält das Attribut *test*, das die Bedingung für die Anwendung der in diesem Element enthaltenen Anweisungen definiert. Die Bedingung in dem Attribut *test* muss dabei eine boolesche Funktion darstellen und kann mit Hilfe von XPath erfolgen.
- **<xsl:apply-templates>** Element gibt die Anweisung alle Kindknoten des aktuellen Knotens zu verarbeiten. Durch das Attribut *select* lassen sich Kindknoten auswählen, die verarbeitet werden sollen.

Um die Elemente eines XML-Dokumentes auszuwählen wird in den Attributen *match* und *select* der XSLT-Elementen XPath verwendet.

2.6.3 Beispiel einer XSLT Transformation

In diesem Kapitel wird ein Beispiel der XSLT Transformation behandelt um die vorgestellten Sachverhalte zu verdeutlichen. In diesem Beispiel wird eine Log-Datei (Listing 2.4) eines Zeiterfassungswerkzeugs zu einem Bericht transformiert. Der verwendete XSLT style sheet (Listing 2.5) enthält zwei Regeln. Die erste Regel (<xsl:template>) wird auf die XML-Elemente <log> angewendet und erstellt für jedes Element ein <report> Element. Des weiteren wird in dieser Regel definiert, dass die Kindelemente mit der zweiten Regel transformiert werden und die Resultate der Transformation der Kindelemente als Kinder des <report> Elements hinzugefügt werden.

Die Zweite Regel erstellt für jedes <worker> Element ein <info> Element. In dieses Element wird Text eingefügt, der den Namen des Arbeiters sowie die geleisteten Arbeitsstunden enthält. Des weiteren wird in diesem Element vermerkt, wenn ein Arbeiter Überstunden geleistet hat. Die Ausgabe der Transformation ist in dem Listing 2.6 dargestellt.

Listing 2.4 Beispiel XML-Dokument

```
<?xml version="1.0" encoding="UTF-8"?>
<log>
	<worker>
		<name>Worker1</name>
		<worked-hours>7</worked-hours>
	</worker>
	<worker>
		<name>Worker2</name>
		<worked-hours>5</worked-hours>
	</worker>
	<worker>
		<name>Worker3</name>
		<worked-hours>10</worked-hours>
	</worker>
	<worker>
		<name>Worker4</name>
		<worked-hours>9</worked-hours>
	</worker>
</log>
```

Listing 2.5 Beispiel eines XSLT Style sheets

```
<xsl:stylesheet version='1.0'
	xmlns:xsl='http://www.w3.org/1999/XSL/Transform'
	    xmlns:xs="http://www.w3.org/2001/XMLSchema">
	<xsl:output method="xml" />

	<xsl:template match="log">
		<xsl:element name="report">
			<xsl:apply-templates select="worker" />
		</xsl:element>
	</xsl:template>
	<xsl:template match="worker">
		<xsl:element name="info">
			<xsl:value-of select="./name" />
			<xsl:text> worked: </xsl:text>
			<xsl:value-of select="./worked-hours" />
			<xsl:text> hours today. </xsl:text>

			<xsl:if test="./worked-hours &gt; 8">
				<xsl:value-of select="./worked-hours - 8" />
				<xsl:text> hour(s) overtime.</xsl:text>
			</xsl:if>
		</xsl:element>
	</xsl:template>
</xsl:stylesheet>
```

Listing 2.6 Ausgabe der Transformation

```
<report>
       <info>Worker1 worked: 7 hours today. </info>
       <info>Worker2 worked: 5 hours today. </info>
       <info>Worker3 worked: 10 hours today. 2 hour(s) overtime.</info>
       <info>Worker4 worked: 9 hours today. 1 hour(s) overtime.</info>
</report>
```

3 Konzept

3.1 Prozessfragmente

In dieser Arbeit wird der Begriff eines Prozessfragments verwendet. Unter einem Prozessfragment wird ein unvollständiger Prozess, d.h. ein Teil des Prozesses verstanden. In einem Prozessfragment im Gegensatz zu einem vollständigen Prozess dürfen Teile des Prozesses undefiniert bleiben. Prozessfragmente sollen dabei als wiederverwendbare Einheiten, die eine bestimmte Funktionalität realisieren, modelliert und verwendet werden. Damit ein Prozess ausgeführt werden kann, soll dieser aus den Prozessfragmenten zusammengesetzt werden. Dabei kann man zwischen *Buildtime Komposition* und *Runtime Komposition* unterscheiden. Diese Unterscheidung basiert auf den jeweiligen Workflow-Lebenszyklen [Tel10].

3.1.1 Buildtime Komposition

Bei der *Buildtime Komposition* werden zuerst die einzelnen Prozessfragmente modelliert. Diese können von verschiedenen Personen mit unterschiedlichen Werkzeugen modelliert werden und so das lokale Wissen über den Prozess, bzw. die wiederverwendbare Prozessteile definieren. Die modellierten Prozessfragmente werden anschließend in einem einheitlichen Format in einem Repository gespeichert. Das Verwenden eines Repository soll dabei die Wiederverwendbarkeit der enthaltenen Prozessfragmente ermöglichen. Wenn alle für den Prozess benötigten Prozessfragmente in dem Repository vorhanden sind, können diese zu einem vollständigen Prozess zusammengesetzt werden. Der so erhaltene vollständige Prozess lässt sich anschließend von einer traditionellen Workflow-Engine ausführen. Dieser Ansatz ermöglicht die Wiederverwendung von Prozessteilen. Es ist jedoch notwendig, dass alle Prozessfragmente, sowie das Wissen über den zukünftigen Prozessablauf bereits vor der Prozessausführung vorhanden sind [Tel10]. Dies ist jedoch, wie in dem Kapitel 1 beschrieben ist, nicht immer möglich. Dieser Nachteil wird mit Hilfe von *Runtime Komposition* beseitigt.

3.1.2 Runtime Komposition

Im Gegensatz zur *Buildtime Komposition* wird bei der *Runtime Komposition* erlaubt einen Prozess auszuführen, bevor alle benötigten Prozessfragmente in dem Repository vorhanden sind. Diese unvollständigen Prozesse werden zur Laufzeit durch weitere Prozessfragmente vervollständigt. Die Prozessausführung beginnt dabei mit einem Startfragment, das aus dem

Repository ausgewählt und gestartet wird. Wenn bei der Navigation durch ein Prozessfragment festgestellt wird, dass ein weiterer Prozessfragment benötigt wird, wird ein passendes Prozessfragment aus dem Repository ausgewählt und an den Prozess angeklebt, so dass die Navigation fortgesetzt werden kann. Bei diesem Ansatz entsteht der Prozess iterativ zur Laufzeit. Die Auswahl von den anzuklebenden Prozessfragmenten kann dabei abhängig von dem Kontext des Prozesses getroffen werden [Tel10]. Die Auswahl der passenden Prozessfragmente, sowie das Ankleben kann dabei sowohl manuell, als auch automatisch erfolgen. Im Rahmen dieser Arbeit wird aufgrund der Aufgabenstellung nur die *Runtime Komposition* mit manueller Fragmentenauswahl und Ankleben ausführlich betrachtet. Es wäre möglich diese Operationen mit Hilfe von Annotationen der Prozessfragmente und durch Definition von Regeln, nach denen die Komposition ablaufen soll, zu automatisieren.

3.1.3 Prozessfragmentelemente

In [EUL09] werden Elemente von Prozessfragmenten vorgestellt, die dem Modellierer erlauben die bekannten Teile des Prozesses zu modellieren, sowie Teile des Prozesses zu kennzeichnen, die dem Modellierer unbekannt sind. Die bekannten Teile des Prozesses werden wie bei einem vollständigen Prozess modelliert. In [EUL09] werden Prozesse betrachtet, die auf Graphen basieren, und somit als Prozesselemente Aktivitäten (die Knoten) und die verbindende Links (die Kanten) verwenden.

Bei der Modellierung von unbekannten Prozessteilen können drei Situationen unterschieden werden. Falls es unbekannt ist, was nach einer Aktivität folgt, wird dies mit aus dieser Aktivität ausgehendem Link modelliert. Der Link wird mit keiner weiteren Aktivität verbunden (Abbildung 3.1 a)). Falls es unbekannt ist, welche Logik vor einer Aktivität ausgeführt wird, wird dies mit einem Link, das mit der bekannten Aktivität verbunden ist, modelliert, das andere Ende des Links wird jedoch mit keiner Aktivität verbunden (Abbildung 3.1 b)). Falls eine Aktivität A vor der Aktivität B ausgeführt werden soll, und es unbekannt ist, welche Logik zwischen den Aktivitäten A und B ausgeführt werden soll, so wird dieser Bereich mit einem neuen Element *Region* modelliert. *Region* definiert dabei einen Bereich des Prozesses, in dem der Prozessablauf unbekannt ist (Abbildung 3.1 c)). Dieser Bereich soll zur Laufzeit durch entsprechende Logik ersetzt werden.

Da dieses Konzept auf Graphen basiert, muss dieses für BPEL adaptiert werden. In [SAL+10] wurden diese Konzepte benutzt um die Compliance von den Prozessen nachzuweisen. Dafür wurde eine Erweiterung von BPEL vorgestellt, die oben beschriebenen Elemente auf BPEL abbildet. Die bekannten Teile der Prozesse werden mit standard BPEL Aktivitäten modelliert, um die Fälle in BPEL modellieren zu können, bei denen ein Link nur mit einer Aktivität verbunden ist, wurden Elemente *frg:fragmentEntry* und *frg:fragmentExit* eingeführt. Das Element *frg:fragmentEntry* wird benutzt um die vor einer Aktivität unbekannte Prozesslogik zu kennzeichnen. *frg:fragmentEntry* muss mindestens einen ausgehenden Link und keine eingehende Links besitzen (Listing 3.1). Das Gegenstück zu *frg:fragmentEntry* ist *frg:fragmentExit*. *frg:fragmentExit* wird verwendet um die nach einer Aktivität folgende unbekannte Prozesslogik zu kennzeichnen. Die Aktivität wird dabei über einen Link mit *frg:fragmentExit*

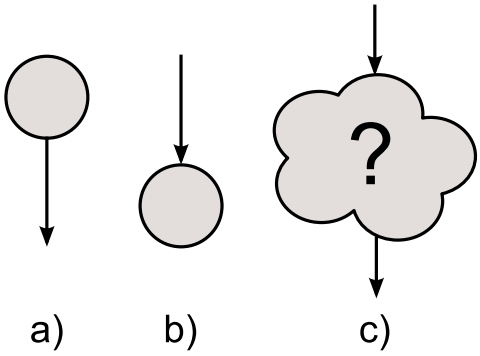

Abbildung 3.1: Fragmentelemente

Listing 3.1 *frg:fragmentEntry* [SAL+10]

```
<frg:fragmentEntry name="entryName" type="mandatory|optional">
       <bpws:sources>
              <bpws:source linkName="linkName"/>+
       </bpws:sources>
</frg:fragmentEntry>
```

verbunden. Somit muss *frg:fragmentExit* Element mindestens einen eingehenden Link und darf keine ausgehenden Links besitzen (Listing 3.2).

Die *frg:fragmentEntry* und *frg:fragmentExit* Elemente werden verwendet um zwei Prozessfragmente mit einander zu verbinden. Die dazugehörigen *frg:fragmentEntry* und *frg:fragmentExit* Elemente werden bei der Komposition durch einen Link ersetzt.

Eine unbekannte *Region* in dem Prozess wird durch *frg:fragmentRegion* modelliert. Dieser darf beliebig viele eingehenden sowie ausgehenden Links besitzen (Linsting 3.3). Dieser Konstrukt wird vor der Prozessausführung durch entsprechende Prozesslogik ersetzt.

Zusätzlich zu den in [EUL09] beschriebenen Prozesselementen wurden in [SAL+10] *frg:fragmentScope* und *frg:fragmentFlow* Elemente eingeführt (Listing 3.4). Diese Elemente wurden von den Standard-BPEL-Aktivitäten <scope> und <flow> abgeleitet und dienen der Klarheit der Semantik. Des weiteren dürfen *frg:fragmentScope* und *frg:fragmentFlow* Elemente die Elemente *frg:fragmentEntry*, *frg:fragmentExit* und *frg:fragmentRegion* enthalten. Bei der Komposition werden die *frg:fragmetScope* und *frg:fragmentFlow* entsprechend durch die <scope> und <flow> BPEL-Aktivitäten ersetzt.

Listing 3.2 *frg:fragmentExit* [SAL+10]

```
<frg:fragmentExit name="exitName" type="mandatory|optional">
       <bpws:targets>
              <bpws:target linkName="linkName"/>+
       </bpws:targets>
</frg:fragmentExit>
```

Listing 3.3 *frg:fragmentRegion* [SAL$^+$10]

```
<frg:fragmentRegion name="regionName">
       <bpws:targets>
              <bpws:target linkName="linkName"/>+
       </bpws:targets>
       <bpws:sources>
              <bpws:source linkName="linkName"/>+
       </bpws:sources>
</frg:fragmentRegion>
```

Listing 3.4 Beispiel zu *frg:fragmentScope* und *frg:fragmentFlow* [SAL$^+$10]

```
<bpws:extensionActivity>
       <frg:fragmentScope name="fragmentScopeName">
              <bpws:variables>
                     <bpws:variable .../>+
              </bpws:variables>
              <!- Other context -->
              <frg:fragmentFlow name="fragmentFlowName">
                     <bpws:links>
                            <bpws:link name="linkName" />*
                     </bpws:links>
                     <frg:fragmentEntry ...
```

Das in [SAL$^+$10] vorgestellte Konzept ist jedoch nur für die *Buildtime Komposition* vorgesehen. Die oben beschriebenen Elemente werden vor der Ausführung durch Standard-BPEL-Konstrukte ersetzt. Aus diesem Grund werden diese Konstrukte in dieser Arbeit für die *Runtime Komposition* adaptiert.

3.2 Eingeführte Aktivitäten

Da die Komposition von Prozessfragmenten zur Laufzeit stattfinden soll, können die in [SAL$^+$10] eingeführten Fragmentelemente nicht vor der Ausführung ersetzt werden. Aus diesem Grund wurden diese Elemente als BPEL erweiternde Aktivitäten übernommen. Zusätzlich wurde die *frg:fragmentSequence* Aktivität eingeführt um blockbasierte Prozessfragmentmodellierung zu unterstützen. Somit wurde BPEL um folgende Aktivitäten erweitert:

- frg:fragmentScope
- frg:fragmentRegion
- frg:fragmentFlow
- frg:fragmentSequence
- frg:fragmentEntry
- frg:fragmentExit

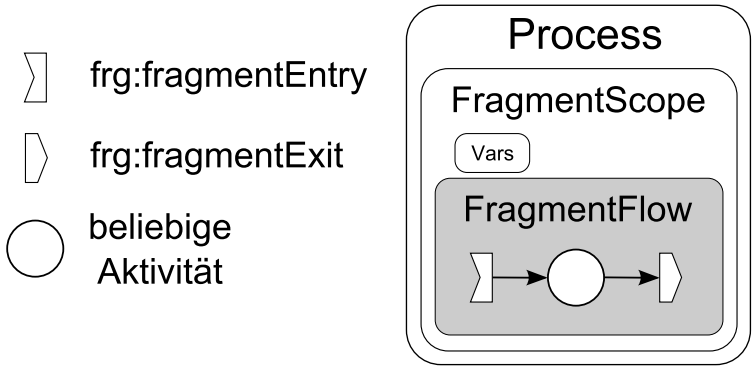

Abbildung 3.2: Legende. Auf der Linken Seite oben sind *frg:fragmentEntry* und die *frg:fragmentExit* Aktivitäten dargestellt. Der Kreis (im Bild links unten) steht stellvertretend für eine Aktivität, ohne dass diese weiter definiert ist. Im Bild Rechst ist ein mögliches Prozessfragment dargestellt.

Die in [SAL+10] vorgestellten *frg:fragmentEntry* und *frg:fragmentExit* Elemente besitzen das Attribut *type="mandatory|optional"*. Dieses Attribut wird in dieser Arbeit nicht verwendet, da die Komposition zur Laufzeit stattfindet und vom Menschen durchgeführt wird. Die Funktionalität zum Ignorieren der *frg:fragmentEntry* und *frg:fragmentExit* Elemente wird vom Menschen aufgerufen und durch die im Kapitel 3.9 beschriebene Schnittstelle ermöglicht.

Auf den Abbildungen von Prozessfragmenten werden in dieser Arbeit der Prozess sowie die Aktivitäten *frg:fragmentScope*, *frg:fragmentRegion*, *frg:fragmentFlow* und *frg:fragmentSequence* als Rechtecke mit abgerundeten Ecken und einer entsprechenden Aktivitätsbezeichnung dargestellt. Durch Kreise werden Aktivitäten dargestellt, die weiter nicht definiert sind und lediglich als Beispiel einer Prozessstruktur dienen. Die Darstellung dieser Aktivitäten sowie der *frg:fragmentEntry* und *frg:fragmentExit* Aktivitäten ist auf der Abbildung 3.2 zu sehen.

3.3 Zusammensetzung von Fragmenten

Bei der Beschreibung von Zusammensetzungen der Prozessfragmenten wird in dieser Arbeit der Begriff eines *Host-Prozessfragments* verwendet. Mit dem *Host-Prozessfragment* wird in dieser Arbeit ein Prozessfragment bezeichnet, in das ein anderes Prozessfragment eingeklebt wird.

Bei der Zusammensetzung von Prozessfragmenten wird der Startfragment iterativ um weitere Funktionalitäten (Prozessfragmente) erweitert. Aus dieser Überlegung wird die Funktionalität (die Aktivitäten und deren Kontext) des einzufügenden Prozessfragments in das Startfragment eingefügt. Das Einfügen von den Aktivitäten und deren Kontext in das Startfragment bringt die Vorteile, die mit Spezifikation von *BPEL-SPE* [BPE05] angestrebt werden, nämlich die Kontrolle über die Ausführung von Subprozessen, in diesem Fall von Prozessfragmenten. So wird es möglich, die in den eingeklebten Prozessfragmenten

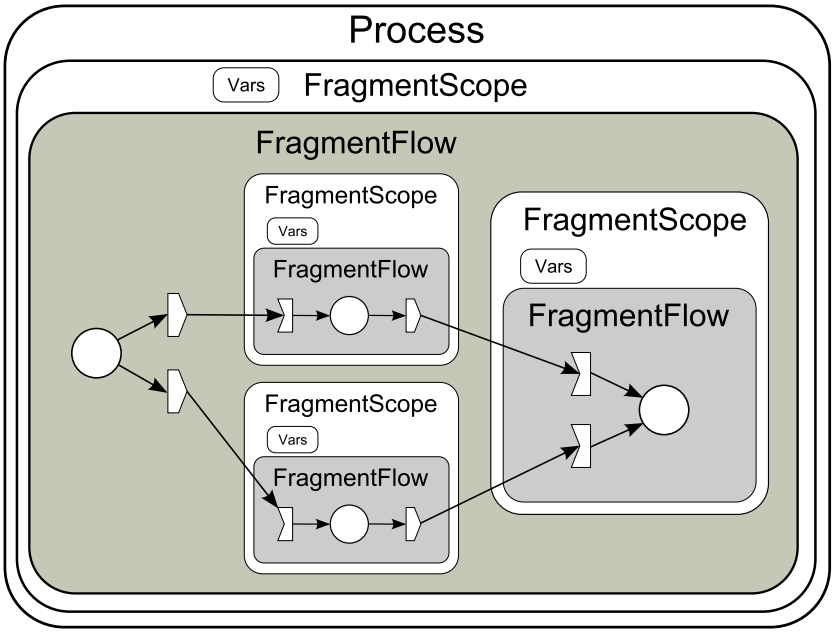

Abbildung 3.3: Kleben von Prozessfragmenten (geringere Schachtelungstiefe). Das letzte Prozessfragment wurde in die äußerste *frg:fragmentFlow* Aktivität eingeklebt.

auftretenden und nicht abgefangenen Fehler im Host-Prozessfragment abzufangen, sowie die Ausführung von den eingeklebten Prozessfragmenten zu terminieren im Fall, dass die Prozessinstanz terminiert wird. Um die Aktivitäten und den Kontext zu kapseln, wird als äußerste Aktivität in einem Prozessfragment *frg:fragmentScope* benutzt, die die Grenzen eines Prozessfragments definieren soll.

Das Kleben von Prozessfragmenten ist nur an bestimmten Stellen im Prozess erlaubt. Es ist erlaubt ein Prozessfragment in *frg:fragmentFlow* als eine neue Aktivität einzukleben, die parallel zu den anderen ausgeführt wird. In *frg:fragmentSequence* ist das Einkleben von Prozessfragmenten am Ende der Sequenz erlaubt und in *frg:fragmentRegion* darf ein Prozessfragment als eine Kindaktivität eingeklebt werden. Falls es mehrere geschachtelten *frg:fragmentFlow* *frg:fragmentSequence* Aktivitäten existieren, dann ist es erlaubt in der Hierarchie nach oben von den unverbundenen *frg:fragmentExit* Aktivitäten zu kleben. Dies erlaubt das Erzeugen von Prozessen mit geringerer Schachtelungstiefe der Aktivitäten (Abbildungen 3.3 und 3.4). In der Abbildung 3.3 wird bei der Komposition die äußerste *frg:fragmentFlow* Aktivität ausgewählt um den letzten Prozessfragment einzufügen. In der Abbildung 3.4 wurde die *frg:fragmentFlow* Aktivität des oberen Prozessfragments zum Kleben ausgewählt, was zu einer größeren Schachtelungstiefe führte. Die Zusammensetzung von Prozessfragmenten wird dabei vom Benutzer geleitet und wird durch die im Kapitel 3.9 beschriebene Schnittstelle ermöglicht.

Das Kleben von den Prozessfragmenten findet auf der Ebene von Kompilierten Prozessen statt, d.h. die beim Deployment erzeugte WFMSs interne Prozessrepresentationen werden

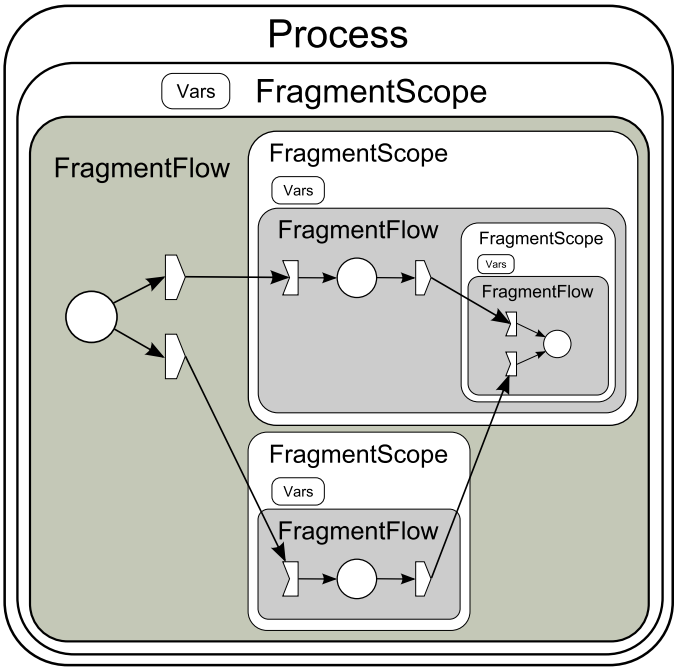

Abbildung 3.4: Kleben von Prozessfragmenten (größere Schachtelungstiefe). Das letzte Prozessfragment wurde in die *frg:fragmentFlow* Aktivität des oberen Prozessfragments eingeklebt.

geklebt. Das verhindert das wiederholte Parsen von den prozessbeschreibenden Dateien. Das Kleben von Prozessfragmenten auf der Ebene von den prozessbeschreibenden Dateien würde weitere Erweiterung von BPEL benötigen, um die Zustände der eingeführten Aktivitäten, sowie Mapping von den Variablen, Partner Links und Correlation Sets beim Kleben festzuhalten.

3.3.1 Komposition mit *frg:fragmentFlow*

Nachdem ein neues Prozessfragment in eine *frg:fragmentFlow* Aktivität eingeklebt wurde, wird *frg:fragmentScope* des eingeklebten Prozessfragments sofort als parallele Aktivität (und damit die im Prozessfragment enthaltenen *frg:FragmentEntry* Aktivitäten) ausgeführt. Die in dem eingeklebten Prozessfragment enthaltenen *frg:fragmentEntry* Aktivitäten blockieren den Kontrollfluss, bis diese jeweils mit einer *frg:fragmentExit* Aktivität verbunden werden.

Dies ist notwendig, da die *frg:fragmentEntry* Aktivität beim Mapping von Variablen, Partner Links und Correlation Sets verwendet wird (Kapitel 3.4). Dabei werden die Werte von Variablen, Partner Links und Correlation Sets aus dem Host-Prozessfragment ausgelesen und an den eingeklebten Prozessfragment übergeben. Zu diesem Zweck müssen die von der *frg:fragmentExit* Aktivität aus sichtbaren Variablen, Partner Links und Correlation Sets ausgelesen werden können. Dies ist nur möglich, falls die entsprechende <scope> bzw.

frg:fragmentScope Aktivität aktiv ist. Um das sicherzustellen, blockiert die *frg:fragmentExit* Aktivität den Kontrollfluss, bis diese mit einer *frg:fragmentEntry* Aktivität verbunden wird. Das Blockieren des Kontrollflusses bei den *frg:fragmentEntry* und *frg:fragmentExit* Aktivitäten erfolgt durch das Warten auf eine Nachricht, die beim Verbinden der *frg:fragmentExit* Aktivität mit einer *frg:fragmentEntry* Aktivität an die beteiligten Aktivitäten geschickt wird.

Beim Verbinden von einer *frg:fragmentExit* Aktivität mit einer *frg:fragmentEntry* Aktivität wird ein Link zwischen den beiden hinzugefügt. Dies ist notwendig für den Fall, wenn die *frg:fragmentFlow* Aktivität sich innerhalb einer Schleife befindet, um die richtige Reihenfolge der Ausführung, nämlich die *frg:fragmentExit* Aktivität vor der *frg:fragmentEntry* Aktivität sicherzustellen. Die *frg:fragmentExit* Aktivität muss vor der *frg:fragmentEntry* Aktivität ausgeführt werden, da innerhalb von Schleifen bei der zweiten und weiteren Iterationen die *frg:fragmentExit* Aktivität den automatischen Mapping anstoßt.

3.3.2 Komposition mit *frg:fragmentSeqeuence*

Bei der Komposition mit *frg:fragmentSequence* gibt es mehrere Stellen innerhalb der Sequenz an denen ein neues Prozessfragment eingeklebt werden könnte, nämlich am Anfang, in der Mitte, und am Ende der Sequenz. Falls das Einkleben des Prozessfragments am Anfang oder in der Mitte der Sequenz benötigt wird, entspricht dies der Semantik der *frg:fragmentRegion* Aktivität, die an diesen Stellen in der Sequenz vorhanden sein muss. Das Einkleben eines Prozessfragments am Ende der Sequenz kann auch ohne einer *frg:fragmentRegion* Aktivität erfolgen. Falls es unbekannt ist, welche Prozesslogik nach der Ausführung der Sequenz ausgeführt werden soll, wird es mit der *frg:fragmentExit* Aktivität am Ende der Sequenz modelliert (Abbildung 3.5). Somit muss es möglich sein ein Prozessfragment als letzte Aktivität der Sequenz einzukleben (Abbildung 3.6). Das eingeklebte Prozessfragment wird dann nach dem Verbinden der *frg:fragmentExit* mit der entsprechenden *frg:fragmentEntry* Aktivität ausgeführt.

3.3.3 Komposition mit *frg:fragmentRegion*

Die *frg:fragmentRegion* Aktivität kapselt einen unbekannten Teil des Prozesses. Der unbekannte Teil des Prozesses wird durch ein Prozessfragment beschrieben und zur Laufzeit in diese Aktivität als eine Kindaktivität eingefügt. Da es bei der Modellierung des einzufügenden Prozessfragments unbekannt ist, an welchen Stellen in einem Prozess es eingefügt wird, und somit unbekannt ist was vor und nach dem Prozessfragment ausgeführt wird, wird jedes Prozessfragment, abgesehen von den Startfragmenten, mit mindestens einer *frg:fragmentEntry* Aktivität und optional mit einer oder mehreren *frg:fragmentExit* Aktivitäten modelliert. Aus dieser Überlegung und der notwendigen Möglichkeit des Datenaustauschs zwischen dem Host- und dem eingeklebten Prozessfragment wird erlaubt *frg:fragmentRegion* Aktivität mit *frg:fragmentEntry* und *frg:fragmentExit* Aktivitäten zu verbinden. Die *frg:fragmentEntry* und *frg:fragmentExit* Aktivitäten müssen dabei dem in die *frg:fragmentRegion* Aktivität eingeklebten Prozessfragment gehören.

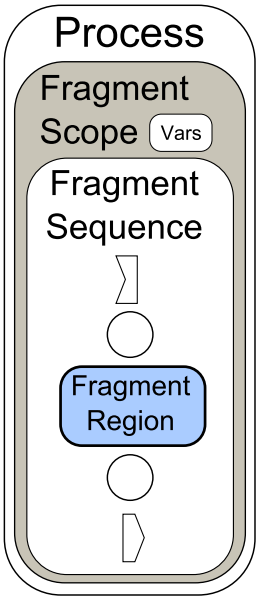

Abbildung 3.5: Die *frg:fragmentEntry* darf in einer *frg:fragmentSequence* Aktivität nur am Anfang der Sequenz vorkommen. *frg:fragmentExit* Aktivität darf nur am Ende der Sequenz vorkommen. *frg:fragmentRegion* Aktivität darf an beliebiger Stelle in der Sequenz auftreten.

3.4 Mapping

Um den Datenaustausch zwischen den Prozessfragmenten zu ermöglichen wird das Konzept des Mappings von Variablen, Partner Links und Correlation Sets eingeführt. Der Datenaustausch ähnelt dabei einem Call-by-Value Aufruf einer Methode, d.h. die Werte von bestimmten Variablen, Partner Links, bzw. Correlation Sets werden aus dem Host-Prozessfragment kopiert, und als Werte der dazugehörigen Elementen des eingefügten Prozessfragments gesetzt. Das Kopieren ermöglicht die Modellierung von Prozessfragmenten ohne das Wissen, in welchem Prozessfragment das modellierte Prozessfragment benutzt wird und ohne Konventionen für die Namensgebung der Variablen, Partner Links und Correlation Sets benutzen zu müssen.

Eine andere Möglichkeit Mapping zu realisieren wäre das Löschen von den Definitionen der zu mappenden Elemente im eingefügten Prozessfragment, so dass die Elemente des Host-Prozessfragments referenziert werden, wie es auf der Abbildung 3.7 dargestellt ist. Dies würde nicht in allen Fällen funktionieren, da es zu den Namenskollisionen kommen kann, und die Umbenennung von gleichnamigen Variablen den Benutzer verwirren würde. Außerdem kann es Fälle geben, in denen die von dem eingeklebten Prozessfragment benötigten Elemente in diesem nicht sichtbar sind (Abbildung 3.8). Ein weiterer Vorteil des Kopierens ist die Kontrolle über die Veränderungen von den Werten der Variablen, Partner Links und Correlation Sets, die von mehreren Prozessfragmenten benutzt werden. Wenn

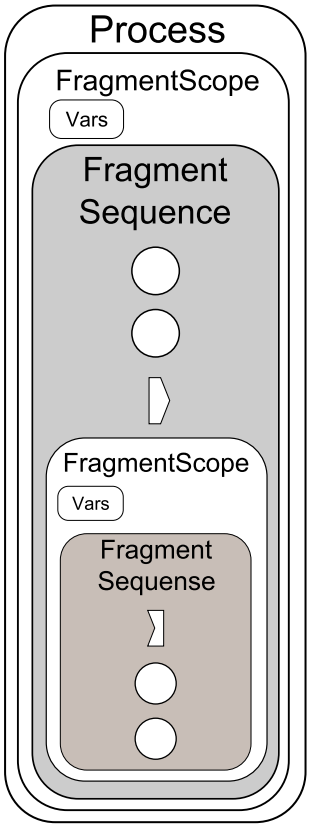

Abbildung 3.6: Das neue Prozessfragment wird am Ende der Sequenz eingefügt.

diese Änderungen beabsichtigt sind, lassen sich diese durch das Zurückkopieren mit Hilfe der *frg:fragmentRegion* Aktivität umsetzen.

Für das Kopieren von Daten über die Prozessfragmentgrenzen werden *frg:fragmentExit* und *frg:fragmentEntry* Aktivitäten benutzt. Mit Hilfe der *frg:fragmentEntry* Aktivität wird während der Modellierung definiert, welche Daten der Prozessfragment benötigt (Listing 3.5). Zur Laufzeit, nach dem Einkleben von dem Prozessfragment, muss bei dem Verbinden von den dazugehörigen *frg:fragmentExit* und *frg:fragmentEntry* Aktivitäten angegeben werden, welche Variablen, Partner Links und Correlation Sets Werte im Host-Prozessfragment enthalten, die von dem eingefügten Prozessfragment benötigt werden. Während das Kopieren von Variablen und Partner Links selbsterklärend ist, wird beim Kopieren von Correlation Sets das Initialisieren des Correlation Sets des eingeklebten Prozessfragments mit den Daten, die beim Initialisieren des Correlation Sets des Host-Prozessfragments verwendet wurden verstanden.

Das Kopieren von den Variablen, Partner Links und Correlation Sets über die Prozessfragmentgrenzen findet in zwei Schritten statt. Der erste Schritt findet dabei in der Aktivität statt, in die ein Prozessfragment eingeklebt wurde. Dieser Schritt wird beim Verbinden von einer *frg:fragmentExit* mit einer *frg:fragmentEntry* Aktivität ausgeführt. In diesem Schritt

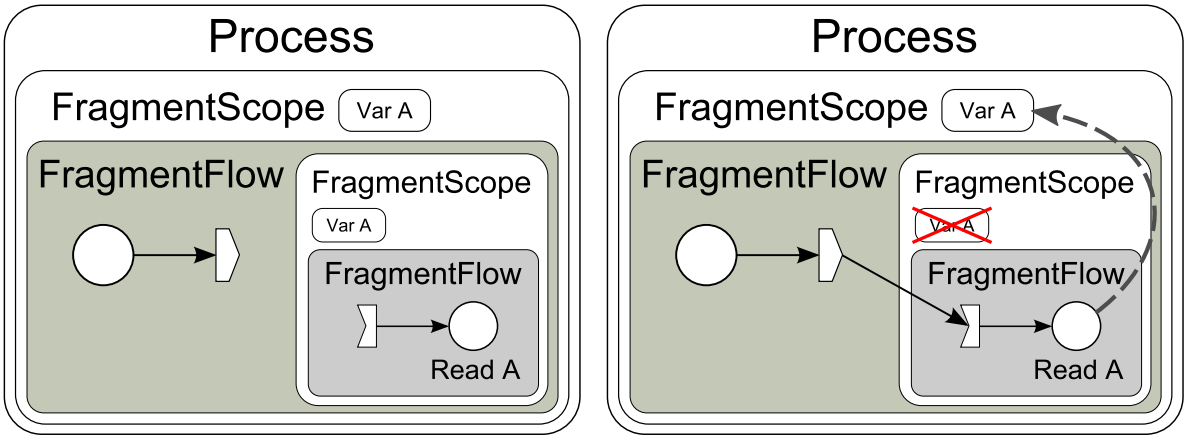

Abbildung 3.7: Variablen-Mapping durch durch Löschen der Variablendefinition in dem eingeklebten Prozessfragment

Listing 3.5 Beispiel zu *frg:fragmentEntry* Variablendefinition

```
<frg:fragmentEntry>
       <elementsToMap>?
              <variableToMap name="..."/>*
              <partnerLinkToMap name="..."/>*
              <correlationSetToMap name="..."/>*
       </elementsToMap>
</frg:fragmentEntry>
```

werden die Werte der notwendigen Variablen, Partner Links, bzw. Correlation Sets gelesen und in der Datenbank zwischengespeichert. Im zweiten Schritt werden die kopierten Werte während der Ausführung der dazugehörigen *frg:fragmentEntry* Aktivität den entsprechenden Variablen, Partner Links, bzw. Correlation Sets zugewiesen. Der Grund für das Kopieren in zwei Schritten ist die Sichtbarkeit von den beim Mapping beteiligten Elementen. Beispiel für eine Situation, die das Kopieren in zwei Schritten notwendig macht, ist auf der Abbildung 3.8 dargestellt. Die in die äußere *frg:fragmentFlow* Aktivität eingeklebten Prozessfragmente haben lokale Variablen: *Variable A* und *Variable B*. Dabei soll beim Variablen Mapping der *Variable B* der Wert der *Variable A* zugewiesen werden. Innerhalb keiner <*scope*> bzw. *frg:fragmentScope* Aktivität sind beide Variablen sichtbar, was das Kopieren des Variablenwertes in zwei Schritten erfordert.

Eine Besonderheit beim Mapping tritt bei der *frg:fragmentRegion* Aktivität auf. Diese spielt beim Mapping die Rollen sowohl von *frg:fragmentExit* als auch *frg:fragmentEntry* Aktivitäten. Bei dem Mapping von Daten aus dem Host-Prozessfragment in das eingeklebte Prozessfragment nimmt *frg:fragmentRegion* die Rolle von der *frg:fragmentExit* Aktivität ein. Dabei wird *frg:fragmentRegion* Aktivität mit der *frg:fragmentEntry* Aktivität des eingeklebten Prozessfragments verbunden und kopiert die Variablenwerte aus dem Host-Prozessfragment. Beim dem Mapping von dem eingeklebten Prozessfragment in das Host-Prozessfragment nimmt die *frg:fragmentRegion* Aktivität die Rolle der *frg:fragmentEntry* Aktivität ein. Dabei

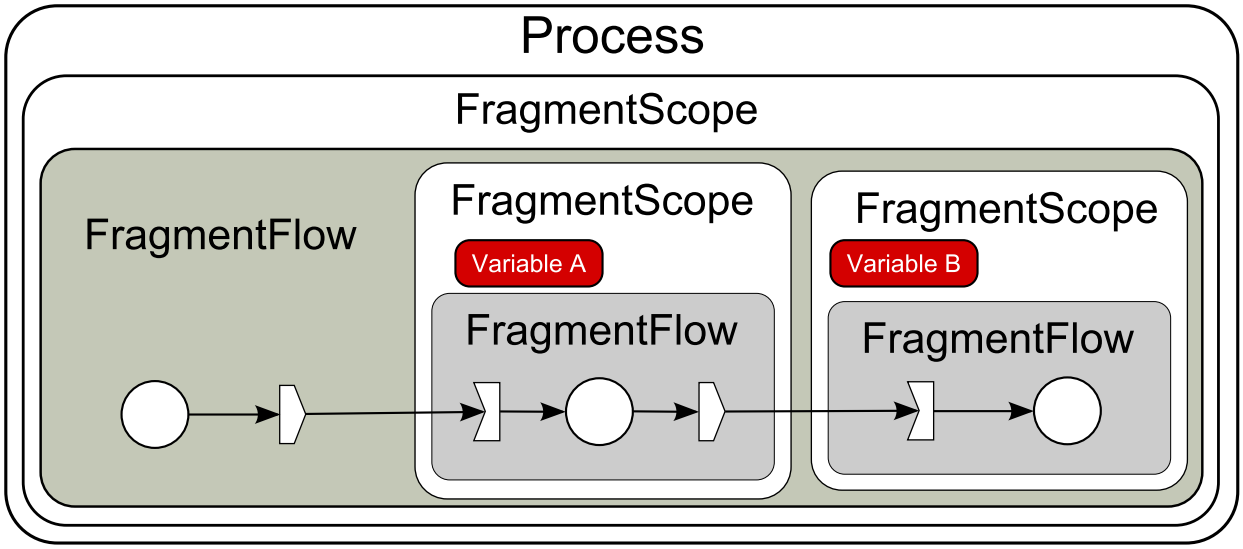

Abbildung 3.8: Sichtbarkeit der Variablen.

Listing 3.6 Beispiel zu *frg:fragmentRegion* Variablendefinition

```
<frg:fragmentRegion>
        <elementsToMap>?
                <variableToMap name="..."/>*
                <partnerLinkToMap name="..."/>*
                <correlationSetToMap name="..."/>*
        </elementsToMap>
</frg:fragmentRegion>
```

wird *frg:fragmentExit* Aktivität des eingeklebten Prozessfragments mit der *frg:fragmentRegion* Aktivität verbunden. Die *frg:fragmentRegion* Aktivität setzt die entsprechenden Werte von Elementen im Host-Prozessfragment bevor die Ausführung dieser Aktivität abgeschlossen wird.

Um die beim Mapping von dem eingefügten Prozessfragment in den Host-Prozessfragment benötigten Elemente zu definieren wird die *frg:fragmentRegion* Aktivität ähnlich der *frg:fragmentEntry* Aktivität mit dem Element <*elementsToMap*> versehen (Listing 3.6).

Falls ein Prozessfragment mehrere *frg:fragmentEntry* Aktivitäten besitzt, können die benötigten Daten in verschiedenen *frg:fragmentEntry* Aktivitäten definiert werden. Das ermöglicht die Ausführung eines Teils des Prozessfragments falls noch nicht alle Daten in dem Prozessfragment vorhanden sind, d.h. nicht alle *frg:fragmentEntry* Aktivitäten ausgeführt wurden, die die Daten kopieren (Abbildung 3.9). Dies muss aber auch die Semantik des Prozessfragments erlauben.

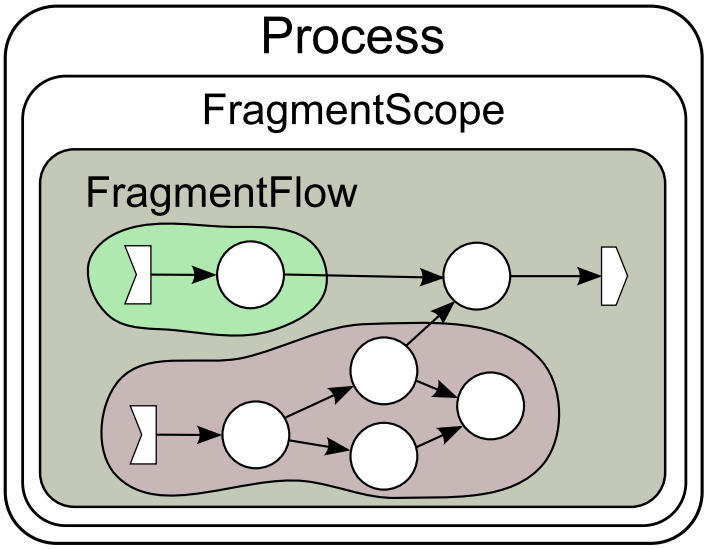

Abbildung 3.9: Teile eines Fragments benötigen unterschiedliche Variablen und können unabhängig von einander ausgeführt werden.

Listing 3.7 Beispiel eines Komplexen Datentyps für die Mediation

```
<customer name="Max Mustermann">
       <address>
              <street>Musterstrasse</street>
              <city>Musterstadt</city>
              <zip>12345</zip>
                  ...
       </assress>
       ...
</customer>
```

3.5 Mediation

Unter Mediation wird in dieser Arbeit das automatische Konvertieren von Datentypen verstanden. Da die Prozessfragmente unabhängig von einander modelliert werden, kann es zu Situationen kommen, in denen die Datentypen der ausgetauschten Daten zwischen den Prozessfragmenten kompatibel sind, jedoch nicht übereinstimmen. Beispiel für solche Situation wäre, wenn die Daten im Host-Prozessfragment als *double* vorliegen, und im eingeklebten Prozessfragment werden diese Daten als *long* benötigt. Ein anderes Beispiel für solche Situation mit Verwendung von komplexen Datentypen wäre, wenn die im Host-Prozessfragment Daten des Kunden in einer Variable des *customer* Datentyps (Listing 3.7) vorliegen, und im eingeklebten Prozessfragment nur die Kundenadresse als *address* Datentyp benötigt wird.

Damit solche Datentyp-Konvertierung automatisiert ablaufen kann, müssen Regeln definiert werden, nach denen die Konvertierung ablaufen soll. Diese Regeln werden beim Mapping

von Variablen angewendet, falls die Datentypen der beteiligten Variablen nicht übereinstimmen. Die Mediation beim Partner Link Mapping wird nicht angewendet, da die Werte von Partner Links lediglich *endpoint references* darstellen. Beim Mapping von Correlation Sets wird jedes Mal Mediation aufgerufen, da die Correlation Sets keine Typen besitzen, deren Übereinstimmung man als Kriterium für das Anwenden der Mediation verwenden könnte. Bei der Mediation der Correlation Sets wird eine Regel anhand von den beteiligten Prozessfragmenten, den Namen der Correlation Sets, sowie den Namen der Scopes, die diese Correlation Sets definieren, ausgewählt. Die Regel für die Konvertierung der Correlation Sets kann die Werte der einzelnen Properties der zu konvertierenden Correlation Sets lesen und diese auf die Properties des gewünschten Correlation Sets abbilden.

3.6 Schleifen

Bis jetzt wurden nur die linearen Konstrukte von BPEL betrachtet, BPEL enthält jedoch auch Schleifen-Konstrukte wie <while>, <repeatUntil> und <forEach> Aktivitäten. Falls innerhalb einer Schleife ein Prozessfragment eingeklebt wird und die *frg:fragmentExit* mit *frg:fragmentEntry* Aktivitäten verbunden werden, wird dies nur bei der ersten Iteration gemacht, bei den weiteren Iterationen wird der eingeklebte Prozessfragment wieder verwendet, der Mapping findet dabei automatisch statt, d.h. die gleichen Elemente werden bei jeder Iteration auf einander abgebildet, wie es bei der ersten Iteration definiert wurde. Laut BPEL-Spezifikation [BPE07, S. 104-105] dürfen keine Links die Grenzen von Schleifen überbrücken, so dürfen es auch die Verbindungen zwischen *frg:fragmentExit* und *frg:fragmentEntry* Aktivitäten nicht.

3.7 Einschränkungen bei der Komposition

Bei der Modellierung von Prozessfragmenten ist dem Modellierer oft unbekannt, welche Prozesslogik vor und welche nach dem zu modellierenden Prozessfragment ausgeführt wird. In diesem Fall wird die unbekannte Prozesslogik von dem zu modellierenden Prozessfragment durch die Aktivitäten *frg:fragmentEntry* bzw. *frg:fragmentExit* abgegrenzt. Die *frg:fragmentEntry* Aktivitäten fehlen in einem Prozessfragment nur, wenn es sich dabei um einen Startfragment handelt. Bei der Komposition von den Prozessfragmenten müssen also zu jedem einzufügenden Prozessfragment mindestens eine *frg:fragmentExit* Aktivität im Host-Prozessfragment vorhanden sein. Das Einkleben von Startfragmenten in ein Prozess widerspricht der modellierten Semantik von Prozessfragmenten.

Aus diesem Grund ist nur eine begrenzte Anzahl von Klebeoperationen innerhalb eines Prozesses möglich. Diese Anzahl gleicht der Anzahl der aktiven unverbundenen *frg:fragmentExit* Aktivitäten plus die Anzahl der aktiven *frg:fragmentRegion* Aktivitäten des Prozesses, in die kein Prozessfragment eingeklebt wurde. Ob es Klebeoperationen innerhalb eines Prozesses erlaubt sind, soll von dem WFMS kontrolliert werden.

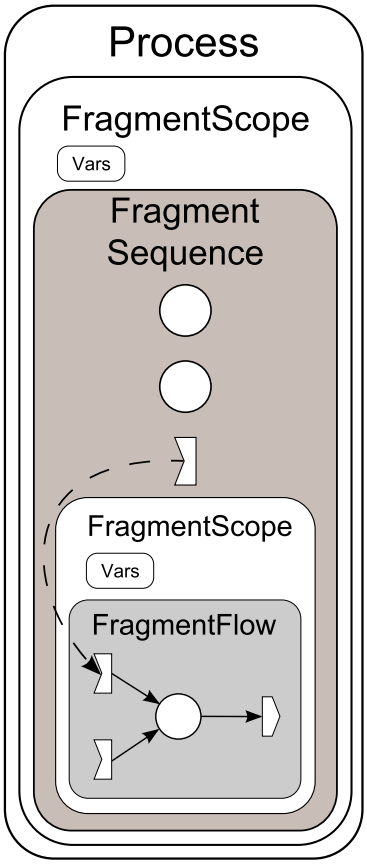

Abbildung 3.10: Der Pfeil deutet die Verbindung von *frg:fragmentExit* und *frg:fragmentEntry* Aktivitäten an, es wird dabei jedoch kein Link erstellt. Die zweite *frg:fragmentEntry* Aktivität kann nicht mehr verbunden werden ohne Zyklen im Prozess zu erzeugen.

Des weiteren kann es bei der Komposition von Prozessfragmenten zu Situationen kommen, dass bestimmte *frg:fragmentEntry* Aktivitäten nicht mehr mit *frg:fragmentExit* Aktivitäten verbunden werden können. Diese Situationen treten beim Ankleben von Prozessfragmenten mit mehreren *frg:fragmentEntry* Aktivitäten auf, falls im Host-Prozessfragment nur eine unverbundene *frg:fragmentExit* bzw. *frg:fragmentRegion* Aktivität vorhanden ist. In diesem Fall ist das Verbinden von einem *frg:fragmentExit frg:fragmentEntry* Paar möglich, alle weiteren *frg:fragmentEntry* Aktivitäten des eingeklebten Prozessfragments können nicht mehr verbunden werden ohne Zyklen zu erzeugen. Beispiel für solche Situation ist auf der Abbildung 3.10 dargestellt. In dieser Situation ist es nicht möglich die *frg:fragmentEntry* Aktivität, die sich links unten im Bild befindet, mit einer *frg:fragmentExit* Akivität zu verbinden ohne Zyklen im Prozess zu erzeugen.

3.8 Generische Architektur

Die generische Architektur eines WFMSs (Abbildung 3.11) besteht im wesentlichen aus folgenden Komponenten:

- **Navigator** ist für das Navigieren durch die Prozesse und die Ausführung der Aktivitäten zuständig.
- **Service Provider** bietet die deployten Prozessmodelle als Web Services an.
- **Service Invoker** wird benutzt um Web Services der Partner aufzurufen.
- **Scheduler** ist für die verzögerte Ausführung von Aktivitäten wie <wait> Aktivität zuständig.
- **Buildtime Datenbank** dient der Speicherung der verfügbaren Prozessmodelle.
- **Runtime Datenbank** wird für die Speicherung von Prozessinstanz-Zuständen und den Daten benutzt, die von den Prozessinstanzen verwendet und erzeugt werden.
- **Auditing Datenbank** wird für die Speicherung der Informationen über die ausgeführten Prozessinstanzen und deren Aktivitäten benutzt.
- **Management API** stellt die Funktionalitäten für die Steuerung des WFMSs zur Verfügung. Beispielsweise Deployen/Undeployen von Prozessmodellen, Terminieren von Prozessinstanzen, abfragen der Auditing-Informationen etc.

Damit ein WFMS Prozessfragmente ausführen kann, soll diese Architektur erweitert werden. Dazu muss diese um die Schnittstellen *FragmentManagement API* und *FragmentComposition API* erweitert werden (Abbilung 3.12). Die Funktionalität wird dabei in zwei unterschiedlichen Schnittstellen angeboten, da die Schnittstelle *FragmentManagement API* keine Interaktion mit der *Navigator* Komponente benötigt, sondern lediglich die Verbindung mit der *Buildtime Datenbank* um die Informationen über die verfügbaren Prozessfragmenten abfragen zu können. Die Schnittstelle *FragmentComposition API* benötigt die Verbindung mit der *Navigator* Komponente um die *Runtime Komposition* durchführen zu können. Die *Navigator* Komponente muss dabei um die in dieser Arbeit eingeführten Aktivitäten erweitert werden. Die Aktivitäten *frg:fragmentEntry*, *frg:fragmentExit* und *frg:fragmentRegion* sollen dabei ähnlich der <receive> Aktivität den Kontrollfluss blockieren können bis eine bestimmte Nachricht empfangen wird, bzw. eine Operation dieser Aktivität ausgeführt wird.

Die Schnittstelle *FragmentComposition API* leitet die Anfragen an die *Navigator* Komponente weiter. *Navigator* erzeugt dabei den Kontext der benötigten Prozessinstanz und leitet die Anfrage an die *FragmentComposition Proxy* Komponente weiter. Diese Komponente leitet die *glue(...)*, *wireAndMap(...)*, *ignoreFragmentEntry(...)* und *ignoreFragmentExit(...)* Anfragen an die entsprechenden Aktivitäten der Prozessinstanz weiter (Abbildung 3.13), und leitet die restlichen Anfragen an die Komponente *FragmentCompositionAnalyzer* weiter (Abbildung 3.14). Diese sammelt Informationen über die aktuelle Fragmentkomposition des Prozesses und benötigt keine Interaktion mit den Aktivitäten der Prozessinstanz. Des weiteren muss die *Buildtime Datenbank* erweitert werden, damit Prozessfragmente gespeichert werden können.

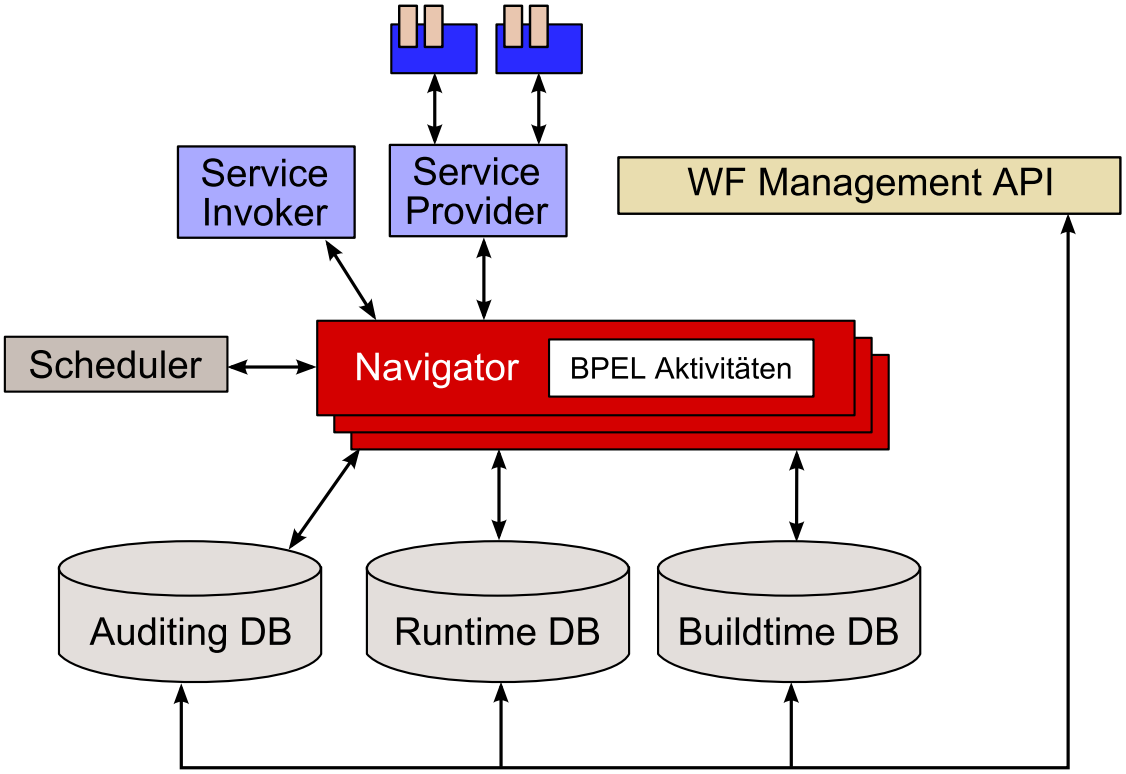

Abbildung 3.11: Generische Architektur

Damit Prozessfragmente deployt werden können, muss auch der Deployment des WFMSs erweitert werden.

Die *Mediator*-Komponente übernimmt die Aufgabe der automatischen Konvertierung von Variablen, sowie den Correlation Sets. Die Schnittstelle dieser Komponente (Listing 3.8) bietet zwei Operationen an. Die Operation *mediateVariable(...)* ist für die Konvertierung von Variablen unterschiedlicher Datentypen zuständig. Als Parameter wird der Datentyp der zu konvertierenden Variable, der gewünschte Datentyp, sowie der Wert der zu konvertierenden Variable angegeben. Die zweite Operation *mediateCorrelationSet(...)* ist für die Transformation von Correlation Sets zuständig. Wie im Kapitel 3.5 beschrieben ist, wird Mediation der Correlation Sets jedes mal beim Mapping von Correlation Sets aufgerufen. Dabei werden die Correlation Sets anhand von dem Prozessfragmentnamen, dem Namen vom Correlation Set, sowie dem Namen des Correlation Set deklarierenden Scopes unterschieden. Die zu konvertierenden Daten, sowie die Resultate der Konvertierung werden dabei durch XML DOM Objekte repräsentiert. Falls die Mediator-Komponente keine Transformation durchführen kann, weil keine entsprechende Transformationsregel definiert ist, oder die Eingabe fehlerhaft ist, wird ein Ausnahmefehler *MediationException* geworfen.

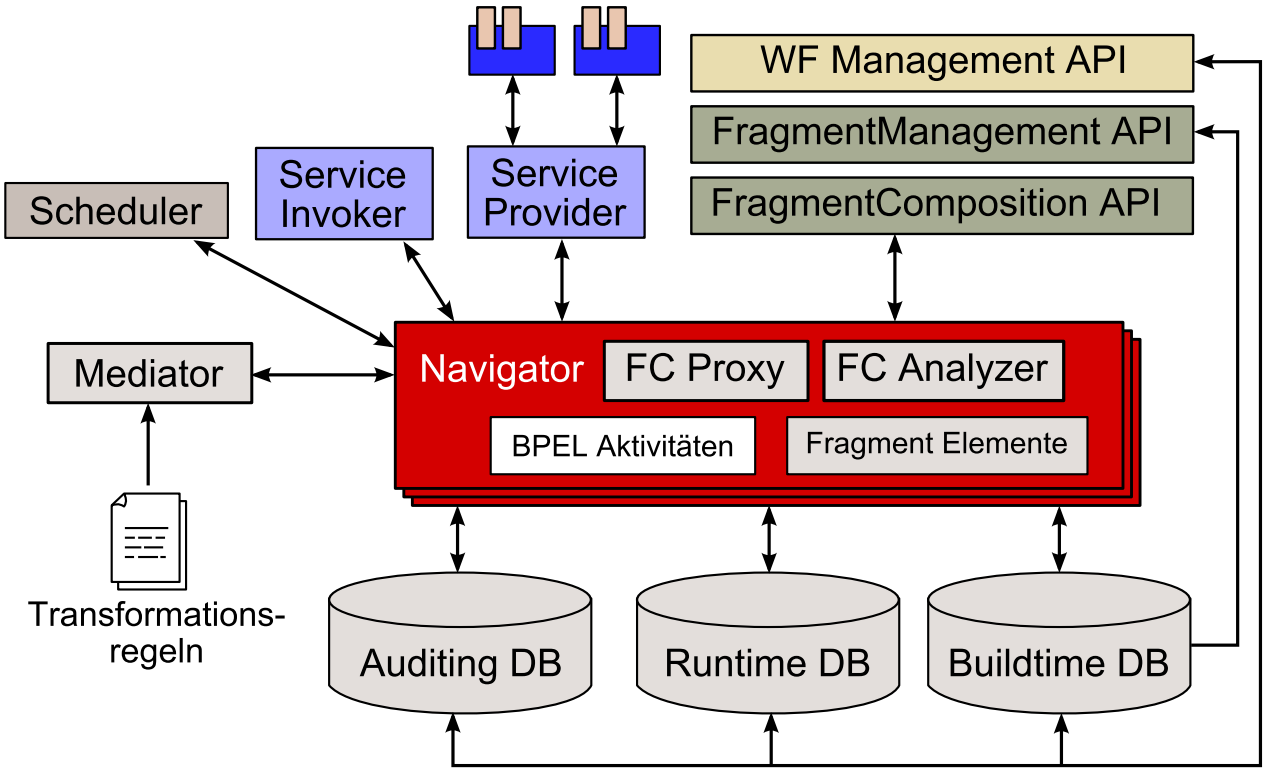

Abbildung 3.12: Erweiterte generische Architektur

Listing 3.8 Schnittstelle der *Mediator*-Komponente

```
public Node mediateVariable(QName fromDataType, QName toDataType, Node xmlData)
    throws MediationException;
public Node mediateCorrelationSet(CSetMediationInfo from, CSetMediationInfo to, Node
    xmlData) throws MediationException;

CSetMediationInfo {
       QName processName;
       String scopeName;
       String correlationSetName;
}
```

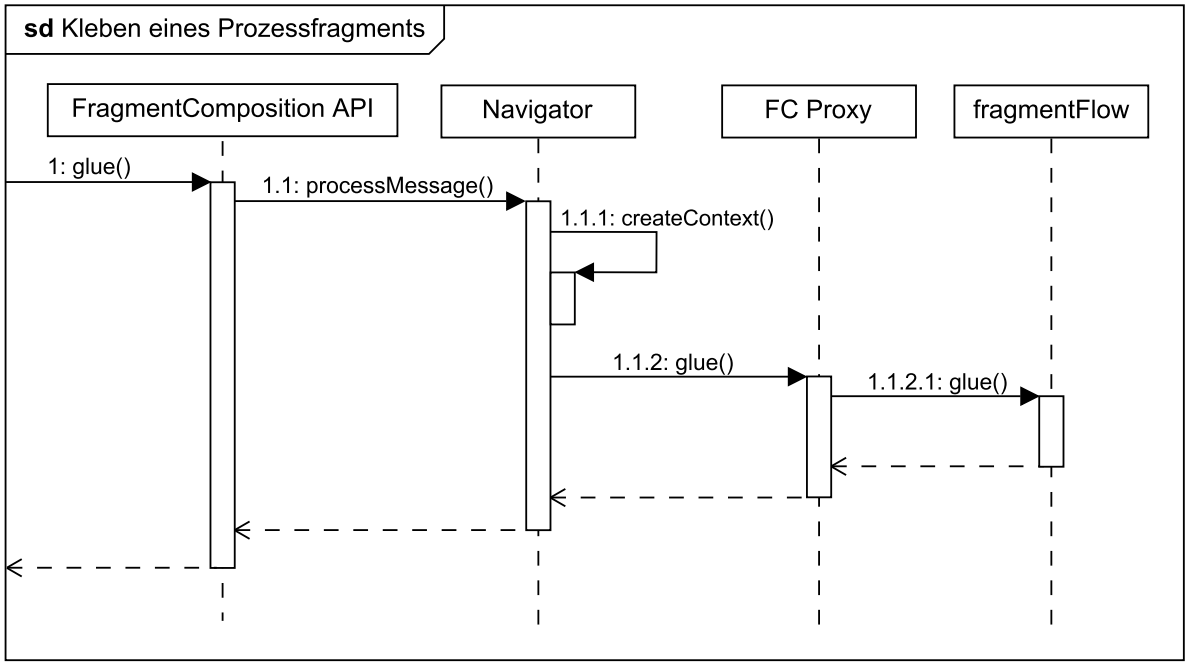

Abbildung 3.13: Aufruf einer *glue(...)* Operation des FragmentComposition API

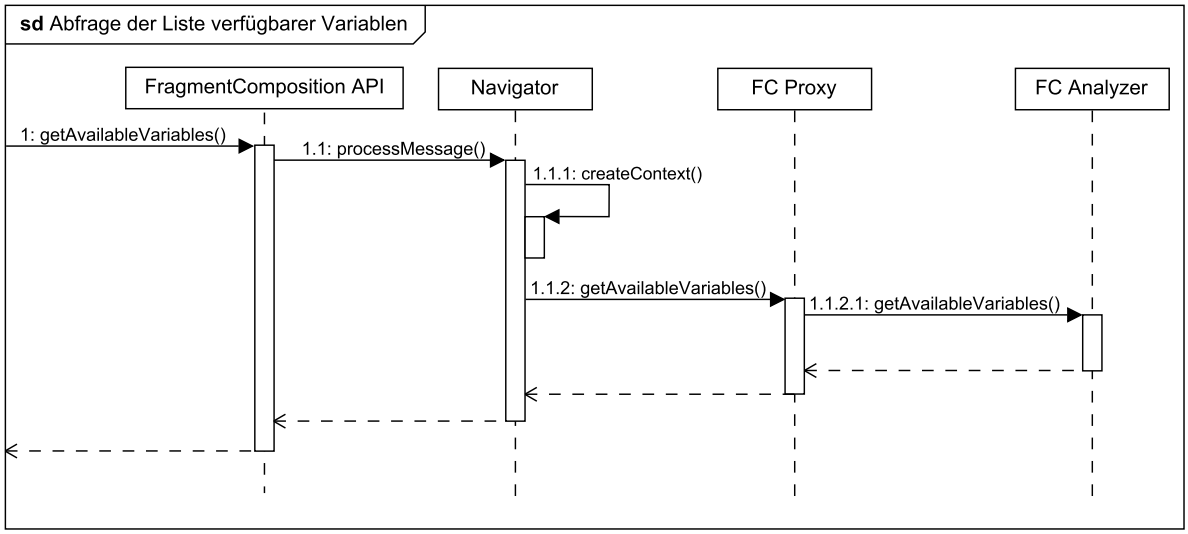

Abbildung 3.14: Aufruf einer *getAvailableVariables(...)* Operation des FragmentComposition API

Listing 3.9 FragmentComposition Schnittstelle

```
public boolean glue(Long instanceId, int containerId, String newFragmentName) throws
    FragmentCompositionException, InstanceNotFoundException;
public boolean wireAndMap(Long instanceId, int fragmentExitId, int fragmentEntryId,
    Mapping[] mappings) throws FragmentCompositionException,
    InstanceNotFoundException;
public boolean ignoreFragmentExit(Long instanceId, int fragmentExitId) throws
    FragmentCompositionException, InstanceNotFoundException;
public boolean ignoreFragmentEntry(Long instanceId, int fragmentEntryId) throws
    FragmentCompositionException, InstanceNotFoundException;
public byte[] getProcessImage(Long instanceId) throws InstanceNotFoundException;

public ActivityInfoList getFragmentContainers(Long instanceId) throws
    InstanceNotFoundException;
public ActivityInfoList getDanglingExits(Long instanceId) throws
    InstanceNotFoundException;
public ActivityInfoList getDanglingEntries(Long instanceId) throws
    InstanceNotFoundException;
public ActivityInfoList getIgnorableExits(Long instanceId) throws
    InstanceNotFoundException;
public ActivityInfoList getIgnorableEntries(Long instanceId) throws
    InstanceNotFoundException;

public VariableInfoList getVariablesToMap(Long instanceId, int elementId) throws
    InstanceNotFoundException;
StringList getPartnerLinksToMap(Long instanceId, int elementId) throws
    InstanceNotFoundException;
public StringList getCorrelationSetsToMap(Long instanceId, int elementId) throws
    InstanceNotFoundException;

public VariableInfoList getAvailableVariables(Long instanceId, int elementId) throws
    FragmentCompositionException, InstanceNotFoundException;
public StringList getAvailablePartnerLinks(Long instanceId, int elementId) throws
    FragmentCompositionException, InstanceNotFoundException;
public StringList getAvailableCorrelationSets(Long instanceId, int elementId) throws
    FragmentCompositionException, InstanceNotFoundException;
```

Listing 3.10 Datenstruktur *ActivityInfo*

```
ActivityInfo {
    String name;
    Integer id;
}
```

3.9 Komposition-API

Laut der Aufgabenstellung dieser Diplomarbeit wird die Komposition von den Prozessfragmenten von einem Menschen geleitet. Um dies zu ermöglichen muss das verwendete WFMS entsprechende Schnittstellen implementieren. Zu diesem Zweck werden die Schnittstellen *FragmentComposition* und *FragmentManagement* eingeführt (Listings 3.9 und 3.13).

Listing 3.11 Datenstruktur *VariableInfo*

```
VariableInfo {
       String name;
       QName type;
}
```

Listing 3.12 Datenstruktur *Mapping*

```
enum ElementType { VARIABLE, PARTNER_LINK, CORRELATION_SET };

Mapping {
       ElementType type;
       String fromVar;
       String toVar;
}
```

Im folgenden werden die Operationen der *FragmentComposition* Schnittstelle vorgestellt:

- Mit Hilfe der *glue(...)* Operation wird ein Prozessfragment in ein so genantes *Container* in einer Prozessinstanz eingefügt. *Container* steht dabei stellvertretend für die Aktivitäten *frg:fragmentFlow*, *frg:fragmentSequence* und *frg:fragmentRegion*.
- Die Operation *wireAndMap(...)* dient dem Verbinden von *frg:fragmentEntry* und *frg:fragmentExit*, bzw. *frg:fragmentRegion* Aktivitäten. Diese Operation benötigt die Ids der zu verbindenden Aktivitäten, sowie eine Liste mit Angaben über die beim Mapping beteiligten Elemente. Die Angaben zum Mapping erfolgen durch die Angaben von Elementen-Namen, sowie dem Typ der Elemente, die auf einander abgebildet werden sollen (Listing 3.12).
- Die Operation *ignoreFragmentExit(...)* schließt die Ausführung einer *frg:fragmentExit* Aktivität ab, ohne diese mit einer *frg:fragmentEntry* Aktivität zu verbinden. Das Ignorieren der *frg:fragmentExit* Aktivitäten ist in den Situationen nützlich, wenn nicht Alle *frg:fragmentExit* Aktivitäten mit den entsprechenden *frg:fragmentEntry* Aktivitäten verbunden werden sollen, da diese die optionalen Prozesszweige darstellen (Abbildung 3.15). In diesem Fall können solche *frg:fragmentExit* Aktivitäten ignoriert werden, d.h. deren Ausführung wird erfolgreich beendet.
- Die Operation *ignoreFragmentEntry(...)* schließt die Ausführung einer *frg:fragmentEntry* Aktivität ab, ohne diese mit einer *frg:fragmentExit* Aktivität zu verbinden. Falls die *frg:fragmentEntry* Aktivität sich in der *frg:fragmentFlow* Aktivität befindet, dann werden die Transition-Conditions der ausgehenden Links aus dieser Aktivität auf *false* gesetzt, somit wird *Dead Path Elimination* [LR00] ausgelöst. Falls die *frg:fragmentEntry* Aktivität sich in der *frg:fragmentSequence* Aktivität befindet, dann wird die *frg:fragmentEntry* Aktivität abgeschlossen und die Ausführung der Sequenz fortgesetzt.

  Das Ignorieren der *frg:fragmentEntry* Aktivitäten ist in den Situationen nützlich, wenn diese zur Synchronisation mit anderen Prozessfragmenten vorgesehen sind, werden aber in der aktuellen Zusammenstellung der Prozessfragmente nicht benötigt.

- Die Operation *getProcessImage(...)* erstellt eine graphische Darstellung des Prozesses und dient der Übersicht über die aktuelle Prozessstruktur. Diese zeigt auch die Aktivitäten Ids, die beim Aufruf bestimmter Operationen dieser Schnittstelle benötigt werden.
- Die Operation *getFragmentContainers(...)* liefert die *Containers*, die Aktiv sind und das Einfügen eines Prozessfragments erlauben.
- Die Operation *getDanglingExits(...)* liefert eine Liste von aktiven und nicht verbundenen *frg:fragmentExit*, sowie *frg:fragmentRegion* Aktivitäten, falls diese die Rolle der *frg:fragmentExit* Aktivität übernehmen. Die Aktivitäten werden dabei durch die *ActivityInfo* Elemente (Listing 3.10) dargestellt.
- Die Operation *getDanglingEntries(...)* liefert eine Liste von aktiven und nicht verbundenen *frg:fragmentEntry*, sowie *frg:fragmentRegion* Aktivitäten, falls diese die Rolle der *frg:fragmentEntry* Aktivität übernehmen.
- Die Operation *getIgnorableExits(...)* liefert die aktiven und nicht verbundenen *frg:fragmentExit* Aktivitäten, die ignoriert werden können.
- Die Operation *getIgnorableEntries(...)* liefert die aktiven und nicht verbundenen *frg:fragmentEntry* Aktivitäten, die ignoriert werden können.
- Die Operation *getVariablesToMap(...)* gibt die Variablen zurück, die in *frg:fragmentEntry* bzw. *frg:fragmentRegion* definiert sind und Mapping benötigen.
- Die Operation *getPartnerLinksToMap(...)* gibt die Partner Links zurück, die in *frg:fragmentEntry* bzw. *frg:fragmentRegion* definiert sind und Mapping benötigen.
- Die Operation *getCorrelationSetsToMap(...)* gibt die Correlation Sets zurück, die in *frg:fragmentEntry* bzw. *frg:fragmentRegion* definiert sind und Mapping benötigen.
- Die Operation *getAvailableVariables(...)* liefert die von einer Aktivität aus sichtbaren Variablen.
- Die Operation *getAvailablePartnerLinks(...)* liefert die von einer Aktivität aus sichtbaren Partner Links.
- Die Operation *getAvailableCorrelationSets(...)* liefert die von einer Aktivität aus sichtbaren Correlation Sets.

In dieser Schnittstelle werden zur Identifikation der Aktivitäten die *Aktivitäten Id* verwendet. Die Aktivitäts Id wird jeder Aktivität von dem WFMS vergeben. Jede Aktivitäts Id ist innerhalb der Prozessinstanz eindeutig. Das ermöglicht das wiederholte Einfügen von den gleichen Prozessfragmenten in den selben Prozess und vermeidet die möglichen Kollisionen, die beim Verwenden von Aktivitätennamen auftreten könnten. Die Aktivitäten Id sind auf der graphischen Darstellung der aktuellen Prozessstruktur angegeben.

Die Operationen *getFragmentContainers(...)*, *getDanglingExits(...)*, *getDanglingEntries(...)*, *getIgnorableExits(...)* und *getIgnorableEntries(...)* liefern eine Liste der Aktivitäten zurück. Die Aktivitäten werden dabei durch die Datenstruktur *ActivityInfo* (Listing 3.10) dargestellt. Die Operationen *getVariablesToMap(...)* und *getAvailableVariables(...)* liefern eine Liste von

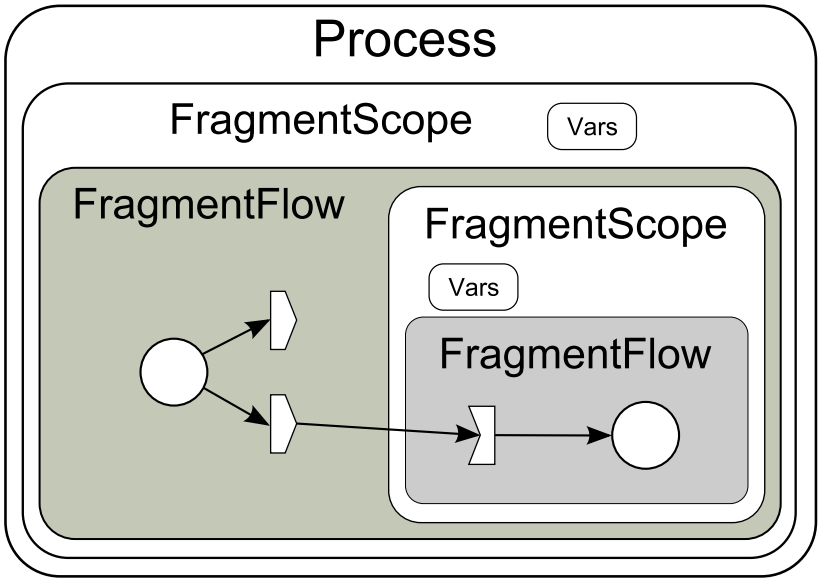

Abbildung 3.15: Eine *frg:fragmentExit* Aktivität wird mit keiner *frg:fragmentEntry* Aktivität verbunden, da diese ein optionales Prozesszweig darstellt und kann ignoriert werden.

Listing 3.13 FragmentManagement Schnittstelle

```
StringList getAvailableFragments() throws ManagementFault;
StringList getAvailableStartFragments() throws ManagementFault;
StringList getAvailableNonStartFragments() throws ManagementFault;
```

Variablen zurück. Die Variablen werden dabei durch die Datenstruktur *VariableInfo* (Listing 3.11) dargestellt. Die Operationen *getPartnerLinksToMap(...)*, *getCorrelationSetsToMap(...)*, *getAvailablePartnerLinks(...)* und *getAvailableCorrelationSets(...)* geben eine Liste von *strings* zurück, die die Namen der entsprechenden Elementen enthalten.

Im folgenden werden die Operationen der *FragmentManagement* Schnittstelle (Listing 3.13) beschrieben:

- Die Operation *getAvailableFragments()* gibt Ids aller verfügbaren Prozessfragmente zurück.
- Die Operation *getAvailableStartFragments()* gibt die Ids der verfügbaren Startfragmente zurück.
- Die Operation *getAvailableNonStartFragments()* gibt die Ids der verfügbaren Prozessfragmente zurück, die keine Startfragmente sind, und somit in andere Prozessfragmente eingeklebt werden können.

Die Operationen dieser Schnittstelle geben jeweils eine Liste von *strings* mit den entsprechenden Ids zurück.

Diese Methoden werden in einer separaten Schnittstelle definiert, da diese im Gegensatz zu den Operationen der *FragmentComposition* Schnittstelle keine Interaktion mit BPEL-Engine benötigen, sondern lediglich den Zugriff auf die *Buildtime* Datenbank.

4 Übersicht über Apache ODE

Apache ODE [ODEa] ist eine Open Source Workflow-Engine, die BPEL-Prozesse ausführen kann und in dieser Arbeit verwendet wird, um den im Kapitel 3 beschriebenen Konzept umzusetzen.

4.1 Architektur

In diesem Kapitel wird ein kurzer Überblick über die Architektur der *Apache ODE* gegeben. Die Architektur von Apache ODE (Abbildung 4.1) besteht aus folgenden Komponenten [ODEa]:

- **BPEL Compiler** erstellt aus den prozessbeschreibenden Dateien (BPEL, WSDL und XSD Dateien) eine ausführbare interne Repräsentation des Prozesses. Diese interne Repräsentation wird in eine *.cbp* Datei serialisiert. Die *.cbp* Dateien werden von der *ODE BPEL Runtime* Komponente verwendet um Prozesse auszuführen.
- **ODE BPEL Runtime** übernimmt die Ausführung von BEPL-Prozessen und enthält die Implementierung der Logik von den einzelnen BPEL-Konstrukten. Des weiteren übernimmt diese Komponente die Zuordnung von den eingehenden Nachrichten zu den entsprechenden Prozessinstanzen.
- **Jacob** Komponente befindet sich innerhalb von *ODE BPEL Runtime* Komponente und stellt die Laufzeitumgebung für die auszuführenden Arbeitseinheiten dar. Als Arbeitseinheiten wird die Logik der einzelnen Aktivitäten der *Apache ODE* umgesetzt. Es erlaubt parallele Ausführung von Arbeitseinheiten innerhalb eines Threads, sowie das Anhalten der Ausführung und Persistieren von dem Ausführungszustand der Arbeitseinheiten.
- **ODE Data Access Objects** übernehmen die Rolle der Vermittlung bei der Datenspeicherung zwischen der *ODE BPEL Runtime* Komponente und der Datenbank.
- **ODE Integration Layer** dient als Kommunikationsmittel mit der *ODE BPEL Runtime*. Mit Hilfe von *AXIS2* [AXI] als Integrationsschicht ist es möglich mit ODE über Web Services zu kommunizieren. Die *JBI* [JBI05] Implementierung von der Integrationsschicht ermöglicht die Kommunikation über *JBI Message Bus*.

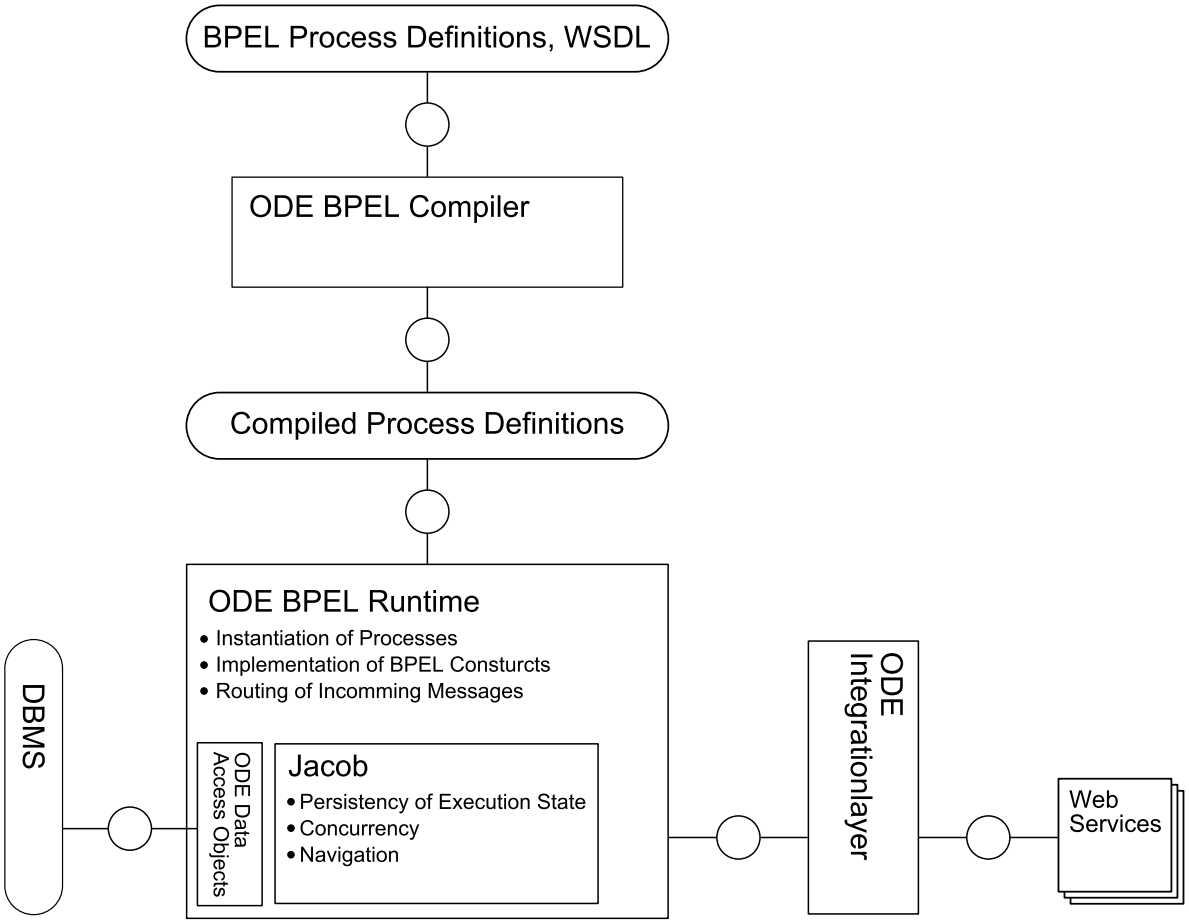

Abbildung 4.1: ODE Architektur [ODEa]

4.2 Deployment

Um einen Prozess in *Apache ODE* zu deployen wird die dazugehörige *.bpel* Prozessbeschreibung benötigt, die *.wsdl* Dateien von den angebotenen und aufzurufenden Web Services, sowie die benötigten *.xsd* Beschreibungen und dem *Deployment Descriptor*. Diese Dateien werden vor dem Deployment in eine *.zip* Detei komprimiert und anschließend deployt.

Apache ODE bietet zwei Möglichkeiten Prozesse zu deployen.

- Kopieren von der *.zip* Datei in den *processes* Ordner der deployten Apache ODE
- Durch die Operation *deploy(...)* des *Management API* [ODEa].

Als Deployment Descriptor dient für die Apache ODE die Datei mit dem Namen *"deploy.xml"*. In dieser Datei werden die Binding-Informationen für die im Prozess benutzten Partner Links angegeben. Als Wurzelelement des Deployment Descriptors dient <deploy> Element.

Listing 4.1 Deployment Descriptor Wurzelelement [ODEa]

```
<deploy>
       <process name=QName ...>*
              ...
       </process>
</deploy>
```

Listing 4.2 Service Binding im Deployment Descriptor [ODEa]

```
<provide partnerLink=NCName>
       <service name=QName port=NCName?>
</provide>
```

Für jeden Prozess in der zu deployten *.zip* Datei wird <process> Element als Kind des Wurzelelements hinzugefügt. Das Attribut *name* des <process> Elements gibt den Namen des zu deployten Prozesses an. Der Name muss dabei mit dem in der *.bpel* Datei angegebenem Prozessnamen übereinstimmen (Listing 4.1). Jedes <prozess> Element muss dabei angeben, welche Services dieser anbietet, sowie deren Binding. Dies wird mit dem Element <provide> beschrieben. Das Attribut *partnerLink* gibt den Namen des verwendeten Partner Links an. Das Kindelement <service> gibt den dazugehörigen Binding an (Listing 4.2). Die aufzurufenden Services im Prozess müssen durch das Element <invoke> im Deployment Descriptor angegeben werden. Die vollständige Schema-Beschreibung des Deployment Descriptors kann in [ddS] nachgeschlagen werden [ODEa].

4.3 Versionierung

Apache ODE unterstützt Versionierung von Prozessen. Diese Funktionalität erlaubt die Verwendung von neuen Versionen von Prozessmodellen ohne den Betrieb der veralteten Prozesse zu stören. Beim Deployment einer neuen Version des bestehenden Prozesses werden die vorhandenen Prozessinstanzen nach der alten Version des Prozessmodells ausgeführt, die neuen Prozessinstanzen werden dabei von der aktuellen Version des Prozessmodells erstellt. Das ermöglicht einen fließenden Übergang zwischen den Versionen von den Prozessmodellen ohne den Betrieb von veralteten Prozessmodellen zu stören [ODEa].

4.4 Apache ODE Channels

Die *Channels* in Apache ODE sind Schnittstellen, die Kommunikation zwischen den Aktivitäten erlauben. Die *Channels* besitzen keine Implementierung. Damit eine Reaktion auf einen Methodenaufruf eines *Cahnnel*s erfolgen kann, muss eine Aktivität einen dem *Channel* zugeordnete *ChannelListener* Schnittstelle implementieren. Der Aufruf der entsprechenden Methode des *ChannelListener*s erfolgt dabei verzögert. Es wird eine Arbeitseinheit erstellt, die die entsprechende Methode des *ChannelListener*s aufruft. Diese Arbeitseinheit wird an *Jacob*

Komponente übergeben, die die Ausführung dieser Arbeitseinheit entsprechend einplant [ODEa].

5 Umsetzung

Bei der Umsetzung des Konzepts müssen folgende Komponenten von *Apache ODE* erweitert werden:

- ODE BPEL Compiler
- ODE BPEL Runtime
- ODE Data Access Objects, sowie das Datenbankschema
- ODE Integrationlayer

5.1 Erweiterung der ODE BPEL Compiler Komponente

Damit die *Apache ODE* die im Kapitel 3.2 eingeführten Aktivitäten aus den BPEL-Prozessbeschreibungen in die interne Darstellung transformieren kann, muss *ODE BPEL Compiler* Komponente erweitert werden. *Apache ODE* unterstützt die <*extensionActivity*> nicht [ODEb], aus diesem Grund werden die neuen Aktivitäten als Standard-BPEL-Aktivitäten umgesetzt. Standard-BPEL-Aktivitäten werden in *Apache ODE* durch Java-Klassen repräsentiert. Die Benennung dieser Klassen folgt dem Schema *O + Aktivitätsname*. Für die <empty> Aktivität heißt die entsprechende Klasse *OEmpty*. Eine kompilierte Darstellung des Prozesses bildet dabei einen Baum dieser Java-Klassen mit der Wurzel *OProcess*. Jede dieser Klassen enthält alle für die Ausführung benötigten Daten. Bei der <receive> Aktivität sind das z.B. der verwendete Partner Link mit dem entsprechenden Binding sowie die verwendeten Correlation Sets.

Klassen, die in der *ODE BPEL Compiler* Komponente die Aktivitäten und den Prozess an sich repräsentieren, kann man in mehrere Gruppen unterteilen. Zunächst wird die *BPEL*-Datei geparst und ein *DOM*-Baum erstellt. Die Klassen mit dem Suffix *Activity* des Pakets *org.apache.ode.bpel.compiler.bom* dienen der XML-unabhängigen Darstellung der Aktivitäten und besitzen Methoden um auf die Eigenschaften von Aktivitäten im *DOM*-Baum zugreifen zu können. Mit Hilfe von Klassen mit dem Suffix *Generator* des *org.apache.ode.bpel.compiler* Paktes können aus diesen XML-abstrahierenden Aktivitätsklassen die Aktivitäten (Klassen mit dem Prefix *O* des Pakets *org.apache.ode.bpel.o*) in der internen Darstellung erzeugt werden (Abbildung 5.1). Diese Klassen enthalten alle für die Ausführung notwendigen Daten, enthalten jedoch keine Implementierung der Aktivitätenlogik, sondern sind eine interne Darstellung des Prozesses. Diese interne Darstellung wird nach dem Kompilieren in eine *.cbp (Compiled BPEL Process)* Datei serialisiert. Diese Datei wird von der *ODE BPEL Runtime* Komponente benutzt um den Prozess auszuführen.

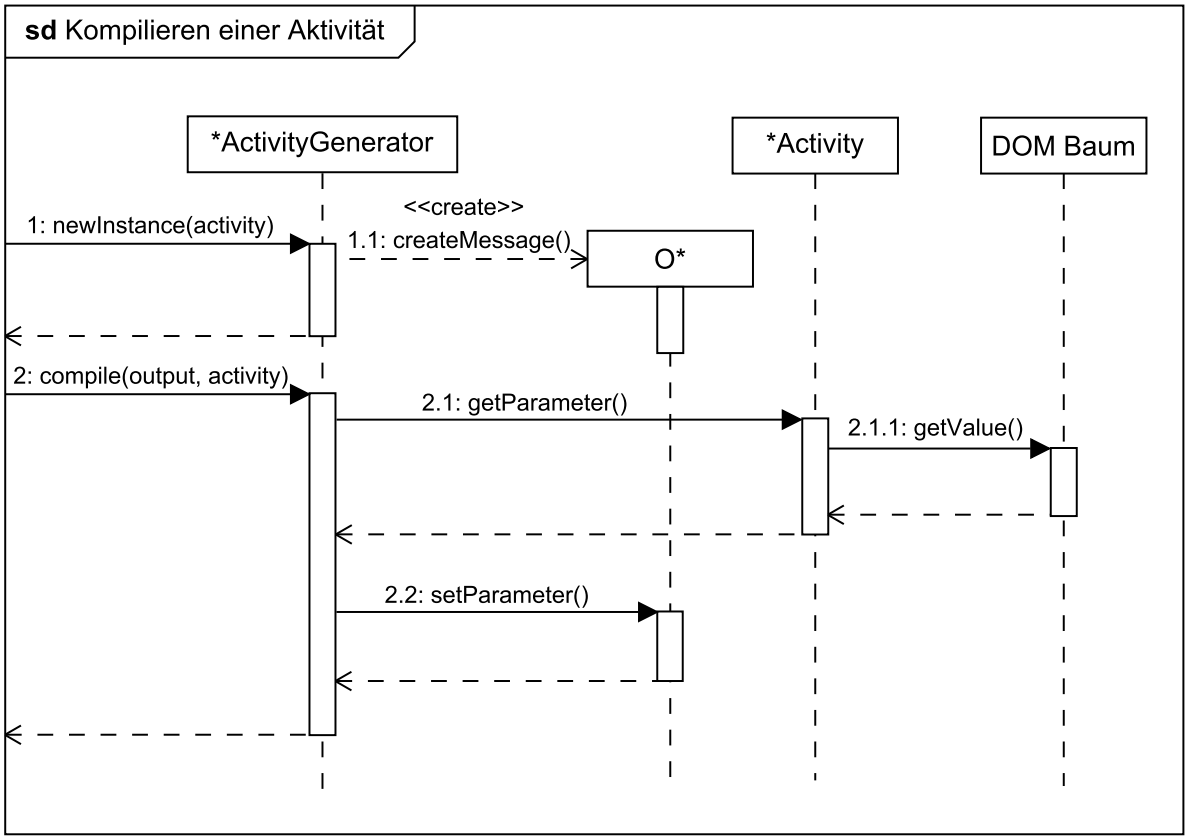

Abbildung 5.1: Kompilieren einer Aktivität. Symbol * steht stellvertretend für den Namen der zu kompilierenden Aktivität.

Um diese Komponente zu erweitern, wurden für jede Aktivität entsprechende XML-abstrahierende Klassen im Paket *org.apache.ode.bpel.compiler.bom*, die entsprechenden Aktivitätsgeneratoren im Paket *org.apache.ode.bpel.compiler*, sowie die interne Repräsentation der Aktivitäten im Paket *org.apache.ode.bpel.o* erstellt. Damit die *ODE BPEL Compiler* Komponente für die neuen BPEL-Aktivitäten entsprechende XML-abstrahierende Klassen finden kann, wurden entsprechende Assoziationen in der Klasse *BpelObjectFactory* des Pakets *org.apache.ode.bpel.compiler.bom* hinzugefügt. Des weiteren wurden die XML-abstrahierende Klassen mit den dazugehörigen Aktivitätsgeneratoren assoziiert. Dies wurde in der Klasse *BpelCompiler20* des Pakets *org.apache.ode.bpel.compiler* gemacht.

5.2 Erweiterung der ODE BPEL Runtime Komponente

5.2.1 Aktivitätenlogik

Damit die interne Repräsentation der eingeführten Aktivitäten ausgeführt werden kann, muss für jede neue Aktivität entsprechende Logik implementiert werden. Die Logik der Aktivitäten wird in Klassen des Pakets *org.apache.ode.bpel.runtime* implementiert, die von der Klasse *ACTIVITY* abgeleitet sind. Objekte dieser Klassen stellen Arbeitseinheiten dar und werden von *Jacob* Komponente ausgeführt. Um die Logik einer Aktivität auszuführen, ruft *Jacob* Komponente die *run()* Methode der Aktivitäten auf.

Um die *ODE BPEL Runtime* Komponente um die Logik neuer Aktivitäten zu erweitern, wurden entsprechende Klassen erstellt, die von der Klasse *ACTIVITY* ableiten und entsprechend die Methode *run()* implementieren. Des weiteren wurde die Klasse *ActivityTemplateFactory* des Pakets *org.apache.ode.bpel.runtime* erweitert, damit diese für die internen Repräsentationen der neuen Aktivitäten die entsprechenden Objekte der Logik implementierenden Klassen zurückgibt.

Die Logik der eingeführten Aktivitäten wird in Listings 5.1, 5.2, 5.3, 5.4 und 5.5 in Pseudocode dargestellt.

5.2.2 Zusätzliche Channels

Um die Funktionalität der Aktivitäten aufrufen zu können, werden drei neue Channel-Typen eingeführt. *FragmentCompositionChannel* (Listing 5.6) besitzt vier Operationen *glue(...)*, *wireAndMap(...)*, *ignoreFragmentEntry(...)* und *ignoreFragmentExit(...)*. Channels dieses Typs sind mit den Containers verbunden. In den Listings 5.1, 5.2 und 5.3 heißen die durch den *FragmentCompositionChannel* aufgerufene Operationen *onGlue(...)*, *onWireAndMap(...)*, *onIgnoreFragmentEntry(...)* und *onIgnoreFragmentExit(...)*.

Die Channels des Types *FragmentCompositionResponseChannel* (Listing 5.7) sind mit den aktiven und unverbundenen *frg:fragmentExit* Aktivitäten verbunden. Der Aufruf der Operation *fragmentCompositionCompleted()* dieses Channels ruft die *onFragmentCompositionCompleted()* Operation der entsprechenden *frg:fragmentExit* Aktivität (Listing 5.5) und beendet die Ausführung dieser Aktivität.

Die Channels des Types *FragmentEntryMappedChannel* (Listing 5.8) sind mit den aktiven und unverbundenen *frg:fragmentEntry* Aktivitäten verbunden. Der Aufruf der Operation *fragmentEntryMapped()* dieses Channels ruft die *onFragmentEntryMapped()* Operation der entsprechenden *frg:fragmentEntry* Aktivität (Listing 5.4), weist die beim Mapping zwischengespeicherten Daten den entsprechenden Elementen zu und beendet die Ausführung dieser Aktivität. Die Operation *ignoreEntry()* dieses Channels ruft die *onIgnoreEntry()* Operation der entsprechenden *frg:fragmentEntry* Aktivität (Listing 5.4) und beendet die Ausführung dieser Aktivität ohne Mapping durchzuführen.

Listing 5.1 Pseudocode der Logik von der *frg:fragmentFlow* Aktivität

```
onGlue(QName newFragmentName, FragmentCompositionResponse response){
       if (isGlueAllowed()) {
              OProcess toGlue = getProcess(newFragmentName);
              OFragmentScope scope = getScopeToGlue(toGlue);
              glueProcesses(thisProcess, toGlue); // Listing 5.9
              addParallelActivity(scope);
              response.returnValue(true);
       } else {
              response.returnValue(false);
       }
}
onWireAndMap(int fragmentExitId, int fragmentEntryId, Mapping[] mappings,
    FragmentCompositionResponse response){
       try {
              wireAndMap(fragmentExitId, fragmentEntryId, mappings); // Listing 5.10
              addLinkIfNeeded(fragmentExitId, fragmentEntryId);
              serializeOProcess(process);
              FragmentCompositionResponseChannel channel =
                  getFragmentExitChannel(instanceId, fragmentExitId);
              channel.fragmentCompositionCompleted();
              removeFragmentExitChannel(instanceId, fragmentExitId);
              response.returnValue(true);
       } catch (Exception e){
              response.throwException(e);
       }

}

onIgnoreFragmentExit(int fragmentExitId, FragmentCompositionResponse response){
       markExitAsIgnored(fragmentExitId);
       FragmentCompositionResponseChannel channel = getFragmentExitChannel(instanceId,
           fragmentExitId);
       channel.fragmentCompositionCompleted();
       removeFragmentExitChannel(instanceId, fragmentExitId);
       response.returnValue(true);
}

onIgnoreFragmentEntry(int fragmentEntryId, FragmentCompositionResponse response){
       markEntryAsIgnored(fragmentExitId);
       setTransitionConditionToFalse();
       FragmentEntryMappedChannel channel = getFragmentEntryChannel(instanceId,
           fragmentEntryId);
       channel.ignoreEntry();
       removeFragmentEntryChannel(instanceId, fragmentEntryId);
       response.returnValue(true);
}

execute(){
       createActivityInstances(parallelActivities);
       addFragmentCompositionChannel(instanceId, flowActivityId, new
           FragmentCompositionChannel());
}
```

Listing 5.2 Pseudocode der Logik von der *frg:fragmentSequence* Aktivität

```
onGlue(QName newFragmentName, FragmentCompositionResponse response){
       if (isGlueAllowed()){
              OProcess toGlue = getProcess(newFragmentName);
              OFragmentScope scope = getScopeToGlue(toGlue);
              glueProcesses(thisProcess, toGlue); // Listing 5.9
              addActivityIntoSequence(scope);
              response.returnValue(true);
       } else {
              response.returnValue(false);
       }
}
onWireAndMap(int fragmentExitId, int fragmentEntryId, Mapping[] mappings,
    FragmentCompositionResponse response){
       try {
              wireAndMap(fragmentExitId, fragmentEntryId, mappings); // Listing 5.10
              serializeOProcess(process);
              FragmentCompositionResponseChannel channel =
                  getFragmentExitChannel(instanceId, fragmentExitId);
              channel.fragmentCompositionCompleted();
              removeFragmentExitChannel(instanceId, fragmentExitId);
              response.returnValue(true);
       } catch (Exception e){
              response.throwException(e);
       }
}

onIgnoreFragmentExit(int fragmentExitId, FragmentCompositionResponse response){
       markExitAsIgnored(fragmentExitId);
       FragmentCompositionResponseChannel channel = getFragmentExitChannel(instanceId,
           fragmentExitId);
       channel.fragmentCompositionCompleted();
       removeFragmentExitChannel(instanceId, fragmentExitId);
       response.returnValue(true);
}

onIgnoreFragmentEntry(int fragmentEntryId, FragmentCompositionResponse response){
       markEntryAsIgnored(fragmentEntryId);
       FragmentEntryMappedChannel channel = getFragmentEntryChannel(instanceId,
           fragmentEntryId);
       channel.ignoreEntry();
       removeFragmentEntryChannel(instanceId, fragmentEntryId);
       response.returnValue(true);
}

execute(){
       if (firstTimeInvoked){
              addFragmentCompositionChannel(instanceId, sequenceActId, new
                  FragmentCompositionChannel());
       }
       OActivity activity = getNextActivityInSequence();
       createActivityInstance(activity);
}
```

Listing 5.3 Pseudocode der Logik von der *frg:fragmentRegion* Aktivität

```
onGlue(QName newFragmentName, FragmentCompositionResponse response){
       if (!regionHasChild()){
              OProcess toGlue = getProcess(newFragmentName);
              OFragmentScope scope = getScopeToGlue(toGlue);
              glueProcesses(thisProcess, toGlue); // Listing 5.9
              setRegionChild(scope);
              response.returnValue(true);
       } else { response.returnValue(false); }
}
onWireAndMap(int fragmentExitId, int fragmentEntryId, Mapping[] mappings,
    FragmentCompositionResponse response){
       try {
              wireAndMap(fragmentExitId, fragmentEntryId, mappings); // Listing 5.10
              serializeOProcess(process);
              OActivity activity = getActivity(fragmentExitId);
              if (activity is frg:fragmentExit){
                     FragmentCompositionResponseChannel channel =
                         getFragmentExitChannel(instanceId, fragmentExitId);
                     channel.fragmentCompositionCompleted();
                     removeFragmentExitChannel(instanceId, fragmentExitId);
              } else if (activity is frg:fragmentRegion){
                     FragmentEntryMappedChannel channel =
                         getFragmentEntryChannel(instanceId, fragmentExitId);
                     channel.fragmentEntryMapped();
                     removeFragmentEntryChannel(instanceId, fragmentEntryId);
              }
              createActivityInstance(regionChild);
              response.returnValue(true);
       } catch (Exception e){ response.throwException(e); }
}
onIgnoreFragmentExit(int fragmentExitId, FragmentCompositionResponse response){
       // Die Semantik von frg:fragmentRegion Aktivität verbietet es.
       response.returnValue(false);
}
onIgnoreFragmentEntry(int fragmentEntryId, FragmentCompositionResponse response){
       // Die Semantik von frg:fragmentRegion Aktivität verbietet es.
       response.returnValue(false);
}
onCompleteActivity(){
       List<MappingInfo> mappingInfos = getMappingInfo(instanceId, activityId);
       assignVariableValues(mappingInfos);
       assignPartnerLinkValues(mappingInfos);
       assignCorrelationSetValues(mappingInfos);
       removeMappings(instanceId, activityId);
       Activity_Completed();
}
execute(){
       if (!isRegionExitMapped()){
              // Nur einemal hinzufüghen, falls es in einer Schleife ausgeführt wird
              addFragmentCompositionChannel(instanceId, regionActId(), new
                  FragmentCompositionChannel());
       } else {
              // frg:fragmentRegion ist in einer Schleife und wurde schon verbunden.
              onWireAndMap(regionId, fragmentEntryId, mappings, new DummyResponse());
       }
}
```

Listing 5.4 Pseudocode der Logik von der *frg:fragmentEntry* Aktivität

```
onFragmentEntryMapped(){
        List<MappingInfo> mappingInfos = getMappingInfo(instanceId, activityId);
        assignVariableValues(mappingInfos);
        assignPartnerLinkValues(mappingInfos);
        assignCorrelationSetValues(mappingInfos);
        removeMappings(instanceId, activityId);
        Activity_Completed();
}

onIgnoreEntry(){
        Activity_Completed();
}

execute(){
        FragmentEntryMappedChannel channel = new FragmentEntryMappedChannel()
        if (fragmentEntryIsMapped()){
                // Prozessfragment wurde am Ende einer frg:fragmentSequence Aktivität eingefügt
                // und wurde mit einer frg:fragmentExit Aktivität verbunden
                // bevor frg:fragmentEntry Aktivität aktiviert werden konnte.
                channel.fragmentEntryMapped();
        } else {
                addFragmentEntryMappedChannel(instanceId, entryActivityId(), channel);
        }
}
```

5.2.3 Kleben von Prozessfragmenten

Das Kleben von Prozessfragmenten kann mit dem Pseudocode aus dem Listing 5.9 dargestellt werden. Als erstes wird die kompilierte Darstellung von dem einzuklebenden Prozessfragment abgefragt, und in diesem die *frg:fragmentScope* Aktivität gefunden, es ist die Kindaktivität des Prozess-Scopes. Abhängig davon welche Aktivität die Rolle des Containers spielt, wird die gefundene *frg:fragmentScope* Aktivität im Falle der *frg:fragmentFlow* Aktivität als eine parallele Aktivität, im Falle der *frg:fragmentSequence* als letzte Aktivität, und im Falle der *frg:fragmentRegion* Aktivität als eine Kindaktivität in die kompilierte Repräsentation des Host-Prozesses eingefügt. In der Wurzel der kompilierten Prozessrepräsentationen werden die benutzten Datentypen referenziert, die fehlenden Datentypen im Host-Prozess müssen dabei hinzugefügt werden.

Die geklebte Prozessrepräsentation muss in *.cbp* Datei serialisiert werden, damit die Änderungen des Prozessmodells beim Neustart der *Apache ODE* erhalten bleiben. Da jedoch zur Laufzeit nicht nur die Kompilierte Repräsentation des Prozesses benötigt wird, sondern auch die WSDL und XSD Dateien um die Web Services anzubieten bzw. aufzurufen, müssen die WSDL und XSD Dateien des Host-Prozessfragments mit neuen WSDL und XSD Dateien vervollständigt werden. Dafür werden die WSDL und XSD Dateien des einzufügenden Prozessfragments in den Ordner des Host-Prozessfragments kopiert. Falls Namenskollisionen auftreten wird ein neuer Name für die Datei generiert, indem eine Zahl am Ende des Dateinamens angehängt wird. Falls der Name der Datei geändert wurde, werden alle Referenzen

Listing 5.5 Pseudocode der Logik von der *frg:fragmentExit* Aktivität

```
execute(){
        addFragmentExitChannel(instanceId, fragmentExitId, new
            FragmentCompositionResponseChannel())

        if (isExitWired(exit)) {
                // Die Aktivität ist bereits mit einer frg:fragmentEntry verbunden,
                // d.h. es wird innerhalb einer Schleife ausgeführt
                int containerId = findEnclosingFragmentContainer(process, exitId);
                FragmentCompositionChannel channel = getFragmentCompositionChannel(instanceId,
                    containerId);
                if (isExitIgnored(exit)){
                        // frg:fragmentExit wurde vorher ignoriert, also wird es nochmal
                            gemacht.
                        fcChannel.ignoreFragmentExit(exitId);
                } else {
                        // sonst wurde wireAndMap ausgeführt
                        fcChannel.wireAndMap(exitId, exit.fragmentEntryId, exit.mappings);
                }
        }
}
onFragmentCompositionCompleted(){
        int fragmentEntryId = getEntryIdExitIsMappedTo();
        OActivity activity = findActivity(fragmentEntryId);
        if (activity is frg:fragmentEntry){
                FragmentEntryMappedChannel channel = getFragmentEntryChannel(instanceId,
                    fragmentExitId);
                channel.fragmentEntryMapped();
                removeFragmentEntryChannel(instanceId, fragmentEntryId);
        } else {
                // frg:fragmentExit ist mit frg:fragmentRegion verbunden
        }
        Activity_Completed();
}
```

Listing 5.6 FragmentComposition Channel

```
        glue(QName newFragmentName, FragmentCompositionResponse response);
        wireAndMap(int fragmentExitId, int fragmentEntryId, Mapping[] mappings,
            FragmentCompositionResponse response);
        ignoreFragmentExit(int fragmentExitId, FragmentCompositionResponse response);
        ignoreFragmentEntry(int fragmentEntryId, FragmentCompositionResponse response);
```

Listing 5.7 FragmentCompositionResponse Channel

```
        fragmentCompositionCompleted();
```

Listing 5.8 FragmentEntryMappedChannel

```
        fragmentEntryMapped();
        ignoreEntry();
```

Listing 5.9 Pseudocode der Operation glue(...)

```
OActivity container;
OProcess toGlue = getProcessToGlue(processname);
OFragmentScope scope = getFragmentScope(toGlue);
container.addChild(scope);
copyDeclaredTypesFrom(toGlue);
serializeOProcess(hostOProcess);
mergeXSDFiles();
mergeWSDLFiles();
mergeDeploymentDescriptors();
reloadHostProcess();
```

auf diese Datei aktualisiert. Falls beide an der Komposition beteiligten Prozessfragmente den gleichen Web Service referenzieren kommt bei der Komposition dazu, dass derselbe Web Service durch zwei WSDL Dateien beschrieben wird. Um das zu verhindern wird beim Kopieren von WSDL Dateien darauf geachtet, welche Web Services in dem Host-Prozessfragment bereits definiert sind. Falls der Web Service bereits existiert, wird aus der WSDL Datei des einzufügenden Prozessfragments die Definition dieses Web Services entfernt. Da es jedoch sein kann, dass die WSDL Datei des einzufügenden Prozessfragments weitere Web Services oder weitere Datentypen definiert, die in dem Prozessfragment benötigt werden, wird der übrig gebliebene Inhalt in den Ordner des Host-Prozessfragments geschrieben.

Zuletzt sollen die vom einzufügenden Prozessfragment benutzten und angebotenen Web Services im Host-Prozessfragment verfügbar gemacht werden. Dazu müssen die Bindings der Web Services aus dem Deployment Descriptor des einzufügenden Prozessfragments in das Deployment Descriptor des Host-Prozessfragments kopiert werden. Damit die Bindings der Web Services des eingefügten Prozessfragments gelesen und angewendet werden, muss das Host-Prozessfragment neu geladen werden, dabei wird die serialisierte Repräsentation des Prozesses gelesen, sowie die Einstellungen aus dem aktualisierten Deployment Descriptor.

5.2.4 Verbinden von *frg:fragmentExit* und *frg:fragmentEntry* Aktivitäten

Bei dem Aufruf von der Operation *wireAndMap(...)* werden die beteiligten *frg:fragmentExit* und *frg:fragmentEntry* bzw. *frg:fragmentRegion* Aktivitäten anhand ihrer Id identifiziert. Diese Aktivitäten werden in dem Prozess gefunden und überprüft, ob für alle von der *frg:fragmentEntry* bzw. *frg:fragmentRegion* Aktivität verlangten Variablen, Partner Links und Correlation Sets entsprechende Abbildungen (Mappings) angegeben sind. Falls alle Daten beim Mapping angegeben wurden, werden die Werte der angegebenen Variablen, Partner Links und Correlation Sets gelesen und in der Datenbank gespeichert. Danach wird die *frg:fragmentExit* Aktivität als verbunden markiert und die Id der dazugehörigen *frg:fragmentEntry* Aktivität gespeichert. Die dazugehörige *frg:fragmentEntry* Aktivität wird ebenfalls als verbunden markiert. Des weiteren werden die Informationen über den Mapping in der *frg:fragmentExit* Aktivität gespeichert. Diese Informationen sowie die Id der dazugehörigen *frg:fragmentEntry* Aktivität werden bei dem Automatischen Mapping innerhalb von Schleifen verwendet.

Listing 5.10 Pseudocode der Operation wireAndMap(...)

```
if (variableMappingOk()){
        readAndStoreVariables(mapping);
        markFragmentExitAsMappedTo(entryId);
        setFragmentExitMappings(mapping);
        markFragmentEntryAsMapped();
} else {
        throw new Exception("Not all Elements are mapped!");
}
```

Listing 5.11 FragmentCompositionResponse

```
FragmentCompositionResponse{
        returnValue(Object value);
        throwException(Exception e);
}
```

Damit die *frg:fragmentExit* Aktivität abgeschlossen wird, wird die Operation *fragmentCompositionCompleted()* des entsprechenden *Channels* aufgerufen. Die hier beschriebene Logik wird in dem Listing 5.10 als Pseudocode dargestellt. Nach dem die *frg:fragmentExit* Aktivität ausgeführt wurde, werden bei der Ausführung der dazugehörigen *frg:fragmentEntry* Aktivität die gespeicherten Werte von den Variablen, Partner Links und Correlation Sets gelesen und als Werte der entsprechenden Elementen zugewiesen. Mapping bei der *frg:fragmentRegion* Aktivität verläuft nach dem ähnlichen Muster, da diese Aktivität die Rollen von *frg:fragmentExit* und *frg:fragmentEntry* Aktivitäten beim Mapping übernimmt.

5.2.5 Ausführung der Logik der eingeführten APIs

Damit die Nachrichten von den Web Services der *FragmentManagement API* und *FragmentComposition API* von den richtigen Prozessinstanzen verarbeitet werden können, wird anhand von der Instanz Id im Inhalt der Nachricht die richtige Prozessinstanz ausgewählt. Anhand des Namens des Web Services wird entschieden, ob eine Nachricht an die *FC Proxy* Komponente (siehe Kapitel 3.8) übergeben wird, oder ob diese von dem Prozess selbst behandelt werden soll. Beim Aufruf von den Operationen des Web Services der *FragmentComposition API* Schnittstelle wird eine gleichnamige Methode der Klasse *FragmentCompositionAPIImpl* aufgerufen. Um die Operationen *glue(...)*, *wireAndMap(...)*, *ignoreFragmentEntry(...)* und *ignoreFragmentExit(...)* der Aktivitäten aufrufen zu können, werden die *Channels* benutzt. Der Aufruf einer Operation eines *Channels* wird dabei als eine Arbeitseinheit von der *Jacob* Komponente verarbeitet. Da die *Channels* keine Rückgabeparameter erlauben, wird bei den Aufrufen dieser Operationen außer den für die Komposition der Prozessfragmenten notwendigen Informationen noch ein Objekt der Klasse *FragmentCompositionResponse* (Listing 5.11) übergeben. Diese Klasse erlaubt das Senden des Ergebnisses der aufgerufenen Operation an den Web Service aufrufenden Partner.

5.2.6 FC Analyser

Die Operationen *getFragmentContainers(...)*, *getDanglingExits(...)*, *getDanglingEntries(...)*, *getVariablesToMap(...)*, *getPartnerLinksToMap(...)*, *getCorrelationSetsToMap(...)*, *getIgnorableEntries(...)*, *getIgnorableExits(...)*, *getAvailableVariables(...)*, *getAvailablePartnerLinks(...)*, *getAvailableCorrelationSets(...)* und *getProcessImage(...)* verändern den Prozessablauf nicht und dienen lediglich dem Abfragen des aktuellen Zustands der Prozesskomposition. Aus diesem Grund werden diese Operationen durch analysieren der kompilieren Prozessdarstellung umgesetzt.

In diesem Abschnitt wird nur auf die Operationen *getFragmentContainers(...)* und *getProcessImage(...)* näher eingegangen, da die Umsetzung der restlichen Operationen keine komplexen Algorithmen erfordert.

Die Operation *getFragmentContainers(...)* liefert die aktiven *frg:fragmentRegion*, *frg:fragmentFlow* und *frg:fragmentSequence* Aktivitäten, die das Einkleben der neuen Prozessfragmenten erlauben. Dafür wird die Menge von *frg:fragmentRegion* Aktivitäten gebildet, in die noch kein Prozessfragment eingeklebt wurde. In diese Menge werden die *frg:fragmentFlow* Aktivitäten hinzugefügt, die sich in dem Prozessbaum über den unverbundenen *frg:fragmentExit* Aktivitäten befinden. Diese Menge wird anschließend mit den *frg:fragmentSequence* Aktivitäten vervollständigt, die am ende der Sequenz eine unverbundene *frg:fragmentExit* Aktivität enthalten. Um nur die Aktiven Container zurückzugeben wird diese Menge mit der Menge der aktiven Containers geschnitten. Dies ist notwendig um nur die Aktiven Containers zu erhalten. Zurückgeben der Menge der Aktiven Containers wäre dabei inkorrekt, da die *frg:fragmentRegion* Aktivitäten jeweils nur eine Klebeoperation erlauben und nach dem Kleben noch Aktiv sind bis diese durch Verbinden mit einer *frg:fragmentExit* Aktivität abgeschlossen werden. Des weiteren wird geprüft, ob die Anzahl der eingeklebten Prozessfragmente kleiner ist als die Anzahl der möglichen Klebeoperationen. Falls dies der Fall ist, wird die oben beschriebene Menge von Containern zurückgegeben, sonst wird eine leere Menge zurückgegeben, da keine Klebeoperationen mehr möglich sind.

Die Operation *getProcessImage(...)* liefert die graphische Darstellung von dem Prozessmodell. Der Aufbau des Bildes erfolgt dabei rekursiv. Zuerst wird rekursiv die Größe der graphischen Darstellung jeder Aktivität ermittelt und deren Position auf der Zeichenfläche bestimmt. Danach wird die entsprechend große Zeichenfläche erstellt um den gesamten Prozess abbilden zu können. Schließlich werden die graphische Darstellungen der Aktivitäten gezeichnet.

Die Resultate dieser Operationen werden ähnlich den *glue(...)*, *wireAndMap(...)*, *ignoreFragmentEntry(...)* und *ignoreFragmentExit(...)* Operationen mit Hilfe der Klasse *FragmentCompositionResponse* zurückgegeben.

5.3 Erweiterung der ODE Data Access Objects Komponente

Damit zu jeder Aktivität der entsprechende *Channel* gefunden werden kann, wird die Abbildung *{Prozessinstanz Id, Aktivitäts Id} -> {Channel, Channel Typ}* in der Datenbank gespeichert.

Listing 5.12 SQL Ausdruck zum Erstellen der Tabelle für Speicherung der Channels

```
CREATE TABLE IF NOT EXISTS ‘ode_channel_selector‘ (
       ‘PROCESS_IID‘ bigint(20),
       ‘ELEMENT_ID‘ bigint(20),
       ‘CHANNEL‘ varchar(255) DEFAULT NULL,
       ‘CHANNEL_TYPE‘ bigint(20),
       PRIMARY KEY (‘PROCESS_IID‘, ‘ELEMENT_ID‘)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

Listing 5.13 SQL Ausdruck zum Erstellen der Tabelle für den Mapping

```
CREATE TABLE IF NOT EXISTS ‘ode_element_mapping‘ (
  ‘PROCESS_IID‘ bigint(20),
  ‘ACTIVITY_ID‘ bigint(20),
  ‘ELEMENT_ID‘ bigint(20),
  ‘MAPPING_DATA‘ text,
  PRIMARY KEY (‘PROCESS_IID‘, ‘ACTIVITY_ID‘, ‘ELEMENT_ID‘)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

Des weiteren muss Mapping persistent gehalten werden. Dafür wird die Abbildung *{Prozessinstanz Id, Activity Id, Element Id} -> {Element-Wert}* verwendet. Zu diesem Zweck werden zusätzliche *Data Access Objects* Benötigt. *Data Access Objects* werden in *Apache ODE* mit Hilfe von *Java Persistence API* umgesetzt. Die *{Prozessinstanz Id, Aktivitäts Id}* und *{Prozessinstanz Id, Activity Id, Element Id}* Tupeln spielen dabei die Rolle der Hauptschlüssel der entsprechenden Tabellen. Entsprechend dieser Erweiterung muss das Datenbankschema, wie in Listings 5.12 und 5.13 gezeigt ist, erweitert werden.

Dabei werden nicht die Channels selbst, sondern deren Ids gespeichert. *Jacob* Komponente erlaubt das Auffinden von dem Channel nach seiner Id, was genutzt wird, nachdem die Channel Id aus der Datenbanktabelle abgefragt wird.

5.4 Mediator-Komponente

Die Mediator-Komponente ist für die Konvertierung von Variablen, sowie Correlation Sets benötigt. Diese Komponente soll um neue Transformationsregeln erweiterbar sein, ohne den Quellcode neu kompilieren zu müssen. Zu diesem Zweck wird XSLT 2.0 (Kapitel 2.6) eingesetzt. Die Transformationsregeln für die Variablen werden dabei in der Datei *var_mediator.xslt* und für die Correlation Sets in der Datei *cset_mediator.xslt* beschrieben. Diese lassen sich bei bedarf erweitern, ohne den Quellcode kompilieren zu müssen.

5.4.1 Variable Mediation

Im Listing 5.14 wird der Inhalt der *var_mediator.xslt* Datei mit dem Beispiel einer Regel der Konvertierung des *boolean* Datentyps zum *integer* Datentyp gezeigt. Diese enthält zwei globale Parameter.

Listing 5.14 Beispiel einer *var_mediator.xslt* Datei mit der Regel der Konvertierung von *boolean* Datentypen zum *integer* Datentypen.

```
<xsl:stylesheet version='1.0'
       xmlns:xsl='http://www.w3.org/1999/XSL/Transform'
           xmlns:xs="http://www.w3.org/2001/XMLSchema">
       <xsl:variable name="integerType" as="xs:QName"
              select="QName('http://www.w3.org/2001/XMLSchema','integer')" />
       <xsl:variable name="booleanType" as="xs:QName"
              select="QName('http://www.w3.org/2001/XMLSchema','boolean')" />

       <xsl:param name="from" as="xs:QName" />
       <xsl:param name="to" as="xs:QName" />
       <xsl:output method="xml" />

       <xsl:template match="/">
              <xsl:if test="$from = $booleanType and $to = $integerType">
                     <xsl:element name="temporary-simple-type-wrapper">
                            <xsl:if test="xs:boolean(/)">
                                   <xsl:text>1</xsl:text>
                            </xsl:if>
                            <xsl:if test="xs:boolean(/) != true()">
                                   <xsl:text>0</xsl:text>
                            </xsl:if>
                     </xsl:element>
              </xsl:if>
       </xsl:template>
</xsl:stylesheet>
```

Listing 5.15 Interne Darstellung des Wertes einer *boolean* Variable mit dem Wert *true* in ODE

```
<temporary-simple-type-wrapper>true</temporary-simple-type-wrapper>
```

- **from** Parameter gibt an, in welchem Datentyp die zu konvertierenden Daten vorliegen.
- **to** Parameter gibt an, in welchen Datentyp die Daten konvertiert werden sollen.

Der Wert der zu transformierenden Variable wird als XML dem XSLT Transformator übergeben.

Der im Listing 5.14 definierte Template wird auf das Wurzelelement der zu transformierenden XML Daten angewendet. Falls der *from* Parameter gleich *xsd:boolean* und der *to* Parameter gleich *xsd:integer* ist, wird die Regel angewendet. Dabei wird ein <temporary-simple-type-wrapper> Element erstellt, das als Wrapper bei der Speicherung von einfachen Datentypen in ODE verwendet wird. Als Wert dieses Elements wird '1' gesetzt falls der Wert des Wurzelelements (des <temporary-simple-type-wrapper> Elements Listing 5.15) gleich 'true' ist, und '0' sonst. Somit würde im Listing 5.15 dargestellte Variable zu der im Listing 5.16 dargestellten Variable transformiert.

Listing 5.16 Interne Darstellung des Wertes einer *integer* Variable mit dem Wert *1* in ODE

```
<temporary-simple-type-wrapper>1</temporary-simple-type-wrapper>
```

Listing 5.17 XML Darstellung eines initialisierten Correlation Sets

```
<?xml version="1.0" encoding="UTF-8"?>
<CorrelationSet>
       <property name="q0:property1"
           xmlns:q0="http://example.com/bpel/process">Wert1</property>
       <property name="q0:property2"
           xmlns:q0="http://example.com/bpel/process">Wert2</property>
</CorrelationSet>
```

5.4.2 Correlation Set Mediation

In ODE werden die Werte von Correlation Sets als eine Liste von Strings gespeichert. Um Correlation Set transformieren zu können werden diese Werte mit den dazugehörigen Properties als XML dargestellt, durch die Mediator-Komponente transformiert und wieder in eine Liste von Strings umgewandelt. Die XML Darstellung eines initialisierten Correlation Sets ist im Listing 5.17 dargestellt. Das Wurzelelement <CorrelationSet> enthält die <property> Elemente, die für Properties des zu transformierenden Correlation Sets stehen. Diese werden durch das Attribut *name* identifiziert. Wert der Property wird dabei als Wert des <property> Elements angegeben.

Der Inhalt der *cset_mediator.xslt* Datei, die zur Konvertierung von Correlation Sets verwendet wird, ist im Listing 5.18 dargestellt. Diese enthält sechs globale Parameter

- **fromProcess** Parameter gibt an, aus welchem Prozessfragment der Wert des Correlation Sets transformiert wird.
- **toProcess** Parameter gibt an, welchem Prozessfragment der Correlation Set gehört, in deren Darstellung die Daten transformiert werden sollen.
- **fromCSetName** Parameter gibt den Namen des zu transformierenden Correlation Sets an.
- **toCSetName** Parameter gibt den Namen des Correlation Sets an, zu dem die Daten transformiert werden sollen.
- **fromScopeName** Parameter gibt den Namen des Scopes an, der den Correlation Set definiert, der im *fromCSetName* Parameter angegebenen ist.
- **toScopeName** Parameter gibt den Namen des Scopes an, der den Correlation Set definiert, der im *toCSetName* Parameter angegebenen ist.

Die zu transformierenden Daten werden wie im Listing 5.17 gezeigt dargestellt.

Listing 5.18 Beispiel einer *cset_mediator.xslt* Datei.

```
<xsl:stylesheet version='1.0'
       xmlns:xsl='http://www.w3.org/1999/XSL/Transform'
           xmlns:xs="http://www.w3.org/2001/XMLSchema">

       <xsl:variable name="fragment1" as="xs:QName"
              select="QName('http://example.com/bpel/process','fragment1')" />
       <xsl:variable name="fragment2" as="xs:QName"
              select="QName('http://example.com/bpel/process','fragment2')" />

       <xsl:variable name="property2" as="xs:QName"
              select="QName('http://example.com/bpel/process','property2')" />

       <xsl:param name="fromProcess" as="xs:QName" />
       <xsl:param name="toProcess" as="xs:QName" />

       <xsl:param name="fromCSetName" as="xs:string" />
       <xsl:param name="toCSetName" as="xs:string" />

       <xsl:param name="fromScopeName" as="xs:string" />
       <xsl:param name="toScopeName" as="xs:string" />
       <xsl:output method="xml" />

       <xsl:template match="/CorrelationSet">
              <xsl:element name="CorrelationSet">
                     <xsl:apply-templates select="./property" />
              </xsl:element>
       </xsl:template>
       <xsl:template match="property">
              <xsl:if test="$fromProcess = $fragment1 and $toProcess = $fragment2 and
                  $fromCSetName = 'corr_set1' and $toCSetName = 'corr_set2'">
                     <xsl:if test="resolve-QName(./@name, .) = $property2">
                            <xsl:copy-of select="." />
                     </xsl:if>
              </xsl:if>
              <xsl:if test="$fromCSetName = $toCSetName">
                     <xsl:copy-of select="." />
              </xsl:if>
       </xsl:template>
</xsl:stylesheet>
```

Listing 5.18 enthält eine Transformationsregel. Diese Regel weist XSLT-Prozessor an die Property mit dem QName *{http://example.com/bpel/process}property2* zu kopieren falls Correlation Set mit dem Namen *corr_set1* des Prozessfragments *{http://example.com/bpel/process}fragment1* auf den Correlation Set mit dem Namen *corr_set2* des Prozessfragments *{http://example.com/bpel/process}fragment2* abgebildet wird. Sowie alle Properties unabhängig davon, welche Prozessfragmente beteiligt sind zu kopieren, falls die Namen der Correlation Sets übereinstimmen.

Listing 5.19 Web Service

```
AxisConfiguration axisConf;
FragmentManagement _fragmentMgmt;
...
WSDLReader wsdlReader = WSDLFactory.newInstance().newWSDLReader();
File fcmwsdlFile = new File(rootpath + "/fcmapi.wsdl");
Definition fcdef = wsdlReader.readWSDL(fcmwsdlFile.toURI().toString());

AxisService fmService = ODEAxisService.createService(
              axisConfig, FM_SERVICE_NAME, FM_PORT_NAME, FM_AXIS2_NAME,
              fcdef, new DynamicMessageReceiver<FragmentManagement>(_fragmentMgmt));
axisConfig.addService(fmService);
```

5.5 Erweiterung der ODE Integrationlayer Komponente

Zugriff auf die Services des *Management API* der *Apache ODE* wird durch die *ODE Integration Layer* ermöglicht. Entsprechend werden auch die *FragmentManagement API* und *FragmentComposition API* Schnittstellen angeboten. In dieser Diplomarbeit wird AXIS2 als *ODE Integration Layer* Komponente verwendet, somit werden diese Schnittstellen als Web Services angeboten.

Die *Management API* von *Apache ODE* wird mit Hilfe der Klasse *ManagementService* als Web Services angeboten. Diese muss auch erweitert werden um die Schnittstelle *FragmentManagement API* als Web Service anzubieten. Zu diesem Zweck wird die dazugehörige WSDL Datei eingelesen und wie in Listsing 5.19 gezeigt ein Web Service erstellt. Die Logik des Web Services wird in diesem Listing durch die Klasse *FragmentManagement* implementiert. Beim Aufruf dieses Web Services wird mit Hilfe von Reflections die Methode der Klasse *FragmentManagement* aufgerufen, die den gleichen Namen wie die aufgerufene Web Service Operation trägt. Dieser Ansatz passt für die Schnittstelle *FragmentManagement API*, da diese keine Interaktion mit den Aktivitäten benötigt und deren Operationen sofort ausgeführt werden können.

Die Nachrichten für die Web Services, die von den Prozessen angeboten werden, werden mit der Klasse *ODEService* empfangen und an die *ODE BPEL Runtime* Komponente weitergeleitet. Um die zwischen den Komponenten *ODE Integrationlayer* und *ODE BPEL Runtime* ausgetauschten Nachrichten zu repräsentieren, wird die Schnittstelle *MessageExchange* benutzt. Diese ermöglicht den Zugriff auf die Daten über die aufgerufene Web Service Operation, sowie auf die empfangene Nachricht. Damit diese Daten im Falle eines Absturzes nicht verloren gehen, werden diese persistiert. Ähnlich werden auch die Nachrichten von dem Web Service der *FragmentComposition API* Schnittstelle an die *ODE BPEL Runtime* Komponente übergeben.

5.6 Werkzeug für die Fragmentenkomposition

Um die Zusammensetzung von Prozessfragmenten benutzerfreundlich zu gestalten, wird ein Werkzeug benötigt, das die *FragmentManagement API* und die *FragmentComposition API* benutzt und deren Funktionalität über graphische Benutzeroberfläche anbietet. Die *Apache ODE* besitzt eine Web-Schnittstelle, die dem Benutzer die Verwaltung der *Apache ODE* durch die Verwendung der *Management API* erlaubt. Aus diesem Grund wurde die Funktionalität der eingeführten Schnittstellen in diese Web-Schnittstelle integriert (Abbildung 5.2). Die *Management API* der *Apache ODE* ist in die Web-Schnittstelle wie folgt integriert (Abbildung 5.3). Die *Apache ODE* besitzt statische *HTML-Seiten*, die mit Hilfe eines Browsers abgerufen werden können. Die HTML-Seiten referenzieren *JavaScript*, der die Funktionalität der *Management API* durch die entsprechenden Web Service Aufrufe nutzt. Die von den Web Services erhaltene Daten werden durch *JavaScript* in die *HTML-Seiten* im Browser eingefügt. Dieses Prinzip wurde auch bei der Integration von den *FragmentManagement API* und *FragmentComposition API* in die Web-Schnittstelle verwendet. Zu diesem Zweck wurde eine neue HTML-Seite erstellt, die die Daten der Fragmentkomposition darstellt, diese referenziert den *JavaScript* für die Aufrufe von Web Services der *FragmentManagement API* und *FragmentComposition API*. Dieser fügt anschließend die empfangenen Daten in die HTML-Seite ein.

5.7 Erstellung von Prozessinstanzen

Bei der Verwendung von Prozessfragmenten ist für jede Prozessinstanz ein separates Prozessmodell notwendig. Das folgt aus der Tatsache, dass zwei Prozessinstanzen des gleichen Startfragments durch Ankleben unterschiedlicher Prozessfragmenten unterschiedliche Prozessmodelle bilden.

Eine Möglichkeit dieses Problem zu lösen ist in [Tel10] vorgestellt. Dabei wird bei dem Ankleben eines Prozessfragments ein neues Prozessmodell erstellt und die laufende Prozessinstanz auf das neue Prozessmodell migriert. Eine andere Möglichkeit ist bei dem Erstellen einer Prozessinstanz automatisch eine neue Kopie von dem Startfragment zu deployen und diese Kopie als Prozessmodell für die neue Prozessinstanz zu verwenden. Diese kann zur Laufzeit auf der Ebene der internen Repräsentation des Prozesses durch Ankleben von Prozessfragmenten geändert werden, ohne die Prozessinstanzmigration durchführen zu müssen. Der letzte Ansatz wurde in dieser Diplomarbeit angewendet. Um eine Kopie des Startfragments bei der Erstellung einer neuen Prozessinstanz zu deployen, wird Web Service der *Management API* der *Apache ODE* verwendet. Dazu müssen die zu dem Startfragment gehörigen Dateien zu einem *ZIP-Archive* komprimiert werden und mit *Base64* kodiert an den Web Service geschickt werden. Anschließend wird das deployte Prozessmodell abgefragt und für die Erstellung der Prozessinstanz benutzt.

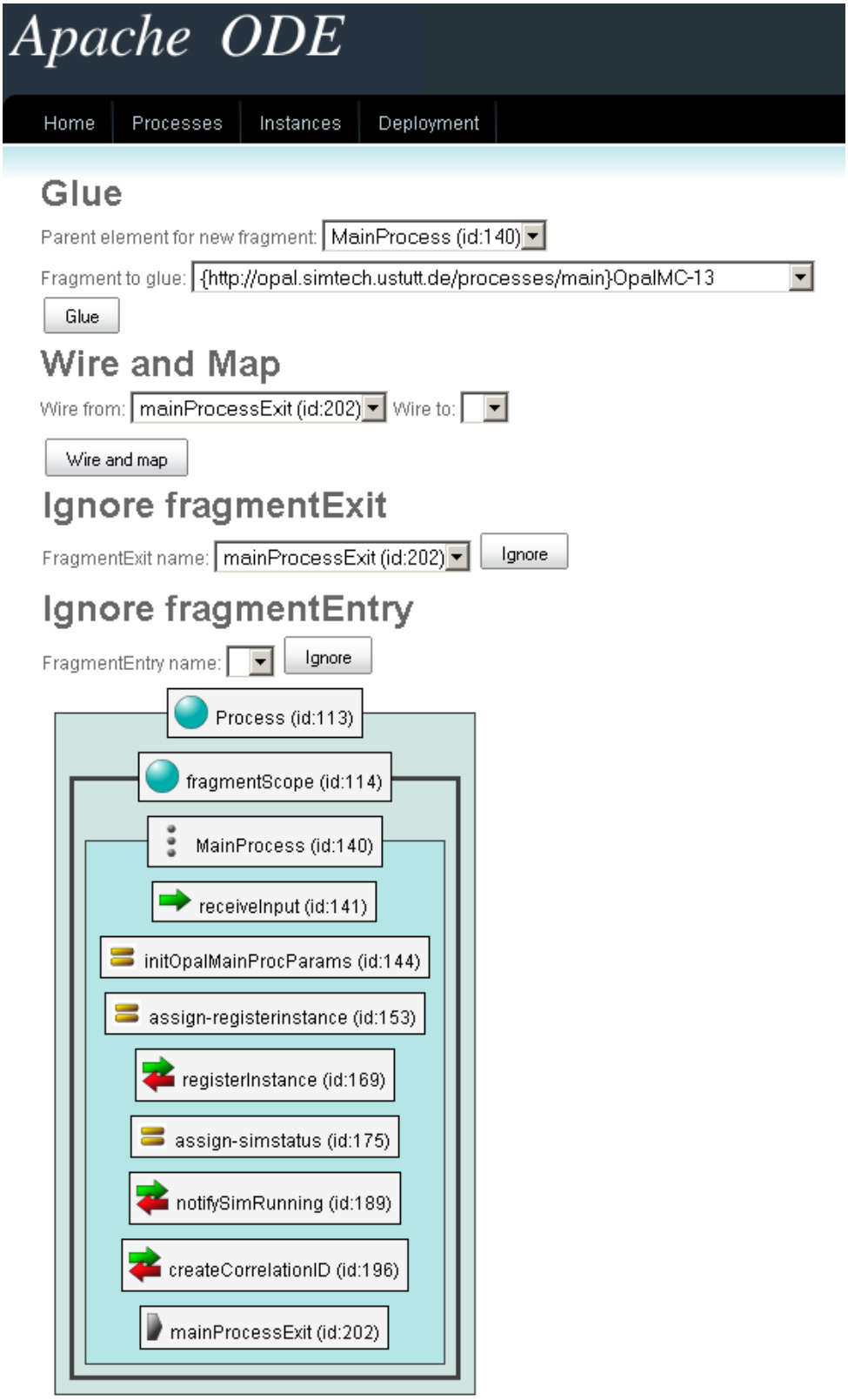

Abbildung 5.2: Werkzeug für die Fragmentenkomposition

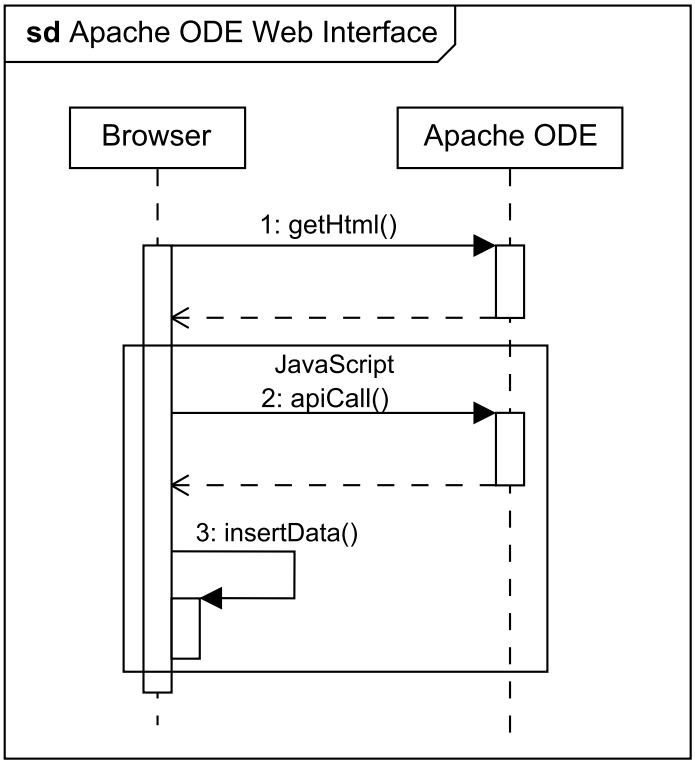

Abbildung 5.3: Apache ODE Web Interface

6 Anwendungsbeispiel

Um das in dieser Arbeit vorgestellte Konzept sowie seine Umsetzung zu prüfen, wurde ein wissenschaftlicher Workflow für die Festkörpersimulation aus [Hot10] verwendet. Dieser Prozess wurde in Prozessfragmente aufgeteilt, die zur Laufzeit zusammengesetzt wurden um vollständigen Prozess zu bilden. In folgenden Abschnitten wird auf den verwendeten Prozess näher eingegangen, sowie auf die erzeugten Prozessfragmente und deren Zusammensetzung.

6.1 Ziel der Festkörpersimulation

Die mit Workflow-Technologie realisierte Simulation erlaubt strukturelle Veränderungen eines metallischen Festkörpers über längere Zeiträume zu simulieren. Die Simulation selbst erfolgt mit Hilfe von Fortran77 Anwendungen, diese werden durch Web Service Aufrufe gestartet. Die Workflow-Technologie kommt dabei zum Einsatz, um Festkörpersimulationen inklusive Vor- und Nachbereitung automatisiert durchführen zu können [Hot10].

6.2 Überblick über die Simulationsanwendung

In diesem Kapitel wird ein kurzer Überblick über die Funktionsweise der Simulationsanwendung geboten.

6.2.1 Aufbau und Funktionsweise der Simulationsanwendung

Die Simulationsanwendung besteht aus folgenden Fortran Anwendungen:

- OpalBCC
- OpalABCD
- OpalMC
- OpalCLUS
- OpalXYZR

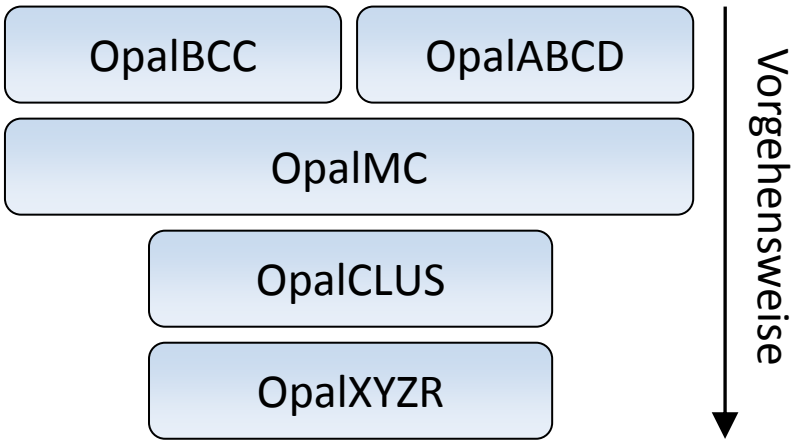

Abbildung 6.1: Aufbau der Simulationsanwendung [Hot10].

Die Reihenfolge der Ausführung von diesen Anwendungen ist auf der Abbildung 6.1 dargestellt.

Die Anwendungen *OpalBCC* und *OpalABCD* werden benutzt, um die Konfigurationsdateien für die Simulation zu erstellen. Die *OpalBCC* Anwendung ist für das Erstellen der Kristallgitters zuständig. Die *OpalABCD* erstellt die Energiekonfigurationen für die Simulation. Diese generierten Kristallgitter- und Energiekonfigurationen werden als Ausgangsdaten benutzt um die Simulation durchzuführen. Die Simulation selbst erfolgt mit Hilfe eines Monte-Carlo Algorithmus, dieser wurde in der *OpalMC* Anwendung implementiert. Um den Verlauf der Simulation analysieren zu können, erlaubt diese Anwendung mit einer bestimmten Frequenz Snapshots, also den Zustand der Simulation (die Positionen aller Atome im Kristallgitter) zu speichern. Diese Snapshots werden von den Anwendungen *OpalCLUS* und *OpalXYZR* für die Analyse benutzt. Die *OpalCLUS* Anwendung ist für das Auffinden von Clustern von Atomen sowie der Identifizierung der Atomen innerhalb eines Clusters zuständig. Die *OpalXYZR* Anwendung berechnet die Radien der gefundenen Cluster im Kristallgitter [Hot10].

6.2.2 Opal Manager

Opal Manager stellt das Bindeglied zwischen dem Benutzer und der Simulationsanwendung dar und hat die Aufgabe die Simulationen zu erstellen und zu verwalten. *Opal Manager* erlaubt auch die zentralisierte Speicherung von den Kristallgitter und Energiekonfigurationen, die beim Start einer Simulation ausgewählt werden können.

6.2.3 Ressourcen Management

Die verwendete Simulation besteht aus rechenintensiven Anwendungen, die bei paralleler Ausführung den Rechner überlasten können. Dies kann vor allem dann auftreten, wenn meh-

rere Simulationen parallel ausgeführt werden. Um dem entgegenzuwirken wurde *Ressourcen-Manager* Komponente entwickelt. Diese koordiniert die Aufrufe der rechenintensiven Web Services und verhindert somit die Rechnerüberlastung. Um eine Anwendung über Web Service im Simulations-Workflow aufzurufen, ist es notwendig den gewünschten Service zu akquirieren. Dies ermöglicht die Kontrolle über die Last der für die Simulation verwendeten Rechnern. Wenn ein benötigter Service und somit die Rechenkapazität frei wird, wird EPR des angeforderten Services als Resultat des Akquirierens asynchron zurückgegeben.

Zusätzlich zu dieser Aufgabe übernimmt *Ressourcen-Manager* die Aufgabe der zentralen Datenspeicherung. Die Operationen zum Lesen und Schreiben von Dateien werden von dem *Ressourcen-Manager* über Web Service Aufrufe ermöglicht [Hot10].

6.2.4 Akquirieren eines Services

Als Beispiel des Akquirierens eines Services kann der auf der Abbildung 6.2 dargestellte Prozessausschnitt verwendet werden. Als Erstes in diesem Ausschnitt wird eine Correlation Id beim Ressourcen Manager angefordert. Diese wird bei der Korrelation der asynchronen Operationen des Ressourcen Managers verwendet. Als nächstes wird die Nachricht der Service-Anfrage erzeugt, in dieser Nachricht wird die Correlation Id angegeben. Anschließend wird die Operation *acquireService* des Ressourcen Managers aufgerufen um den gewünschten Service zu akquirieren. Da als Rückgabe dieser Operation unterschiedliche Nachrichten empfangen werden sollen, wird *pick* Aktivität benutzt. Falls die Nachricht *aquireServiceCallback* empfangen wird, wird der Service auf die erhaltene EPR gebunden und kann benutzt werden. Wenn der Service nicht mehr benötigt wird, muss er mit Hilfe der *releaseService* Operation des Ressourcen Managers freigegeben werden.

6.3 Prozesse der Simulationsanwendung

Die Simulationsanwendung besteht aus zwei Prozessen. Der erste Prozess *OpalMainProc* führt die eigentliche Simulation durch, der Prozess *OpalSnapProcLink* analysiert die erstellten Snapshots. Diese Prozesse werden auf der Abbildung 6.3 schematisch dargestellt.

6.3.1 Haupt-Prozess

Der Haupt-Prozess beginnt mit dem Initialisieren des Prozesses, was die Registrierung der Prozessinstanz bei *OpalManager* und Abfragen einer Correlation Id einschließt, die bei Kommunikation mit der Prozessinstanz benutzt wird. Die Ausführung von den *OpalBCC* und *OpalABCD* Anwendungen ist in den Prozess nicht eingeschlossen, da nicht für jede Simulation eine neue Konfiguration benötigt wird. Nach der Initialisierung des Prozesses wird *OpalMC* Dienst Akquiriert und aufgerufen, anschließend wartet Haupt-Prozess auf die Rückgabe vom *OpalMC* Dienst. Als Rückgabe können ein Snapshot für die Nachbearbeitung, oder eine Nachricht sein, die den Abschluss der Simulation signalisiert. Um mehrere

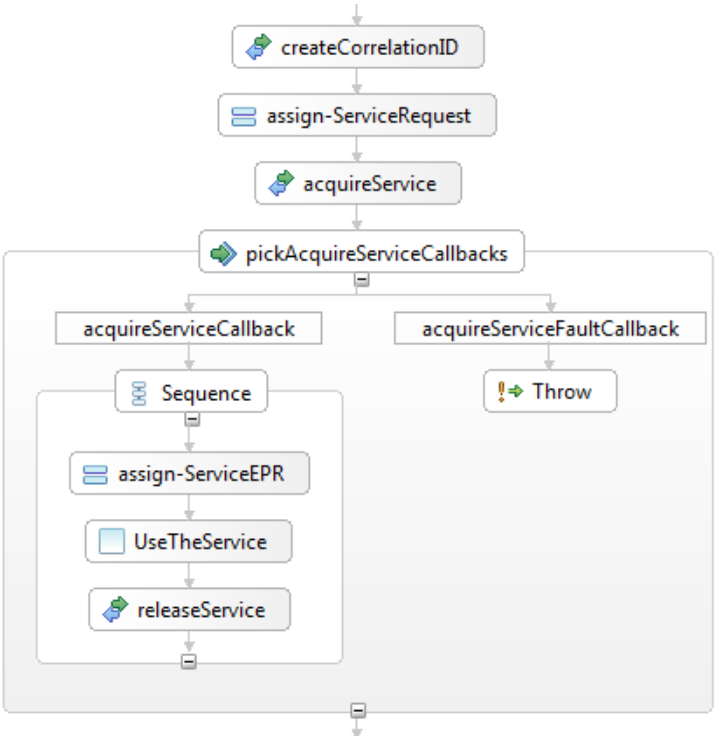

Abbildung 6.2: Beispiel des Akquirierens eines Services in BPEL [Hot10].

Nachrichten vom *OpalMC* Dienst empfangen zu können, befindet sich die entsprechende *pick* Aktivität innerhalb einer Schleife. Für jeden erhaltenen Snapshot wird ein Nachbereitungsprozess gestartet. In der nächsten Schleife des Haupt-Prozesses wird auf das Ende der Nachbereitung aller Snapshots gewartet. Wurden alle Snapshots verarbeitet, so wird die Visualisierung der verarbeiteten Snapshots durchgeführt.

6.3.2 Nachbereitungsprozess

Der Nachbereitungsprozess akquiriert den *OpalCLUS* Dienst und ruft diesen auf um den Snapshot zu analysieren. Nachdem der *OpalCLUS* Dienst die Datenverarbeitung abschließt, wird der Dienst wieder freigegeben und der Hauptprozess über den Abschluss der Verarbeitung benachrichtigt.

6.4 Aufteilung des Prozesses in Prozessfragmente

Im Folgenden wird die Aufteilung des Haupt-Prozesses in Prozessfragmente erläutert. Der Nachbereitungsprozess wird in Prozessfragmente nicht aufgeteilt, da dieser Prozess für

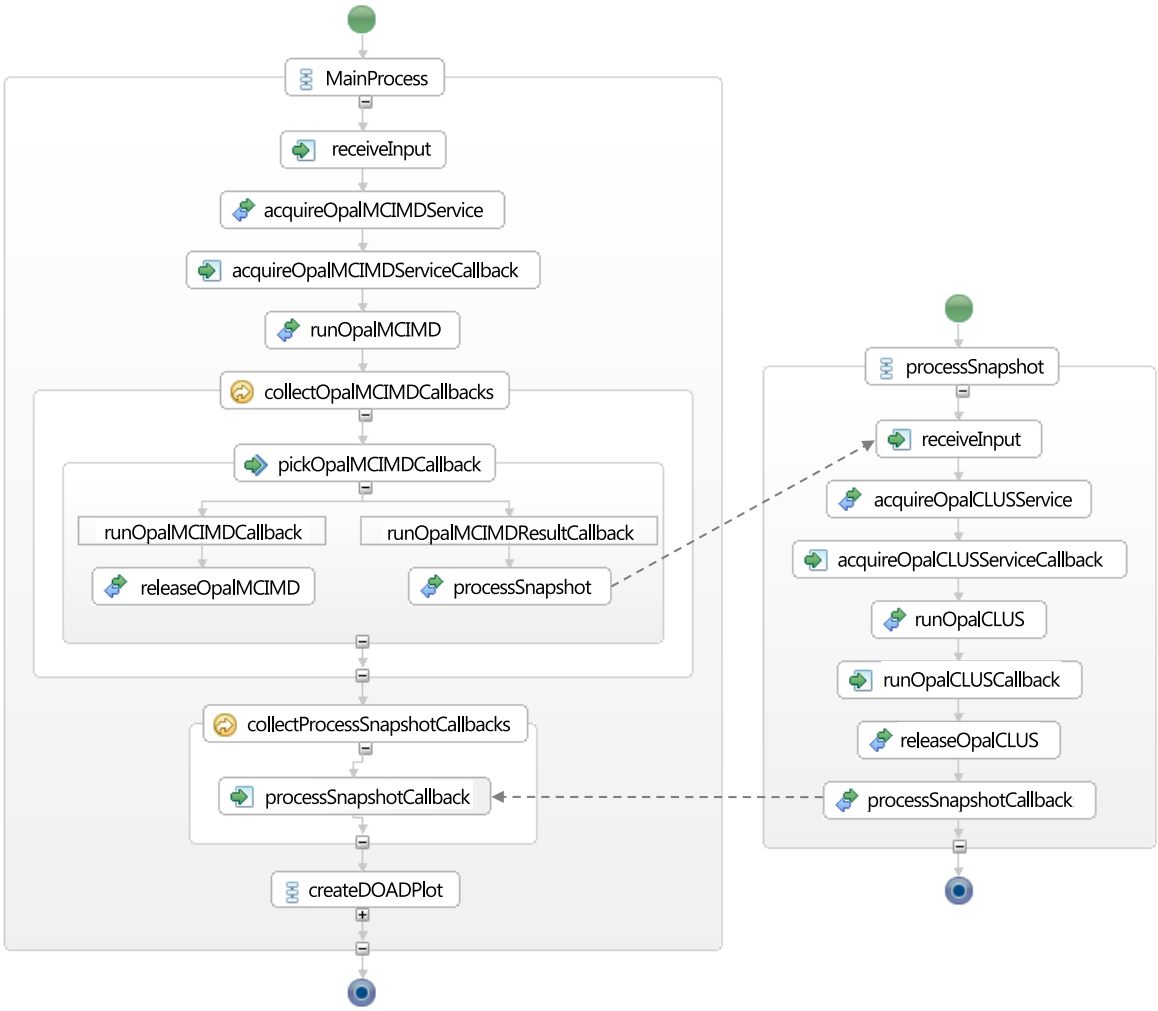

Abbildung 6.3: Schematische Darstellung der Simulationsprozessen [Hot10].

jeden Snapshot aufgerufen wird, und man für jeden Snapshot den Nachbereitungsprozess neu aus Prozessfragmenten zusammensetzen müsste.

Der Haupt-Prozess besteht aus drei logischen Blöcken die nacheinander ausgeführt werden: Prozessinitialisierung, Ausführung der Monte-Carlo Simulation und Visualisierung. Entsprechend dieser Einteilung wird der Haupt-Prozess in Prozessfragmente aufgeteilt. Zusätzlich wurde der Block von Monte-Carlo Simulation in zwei Prozessfragmente aufgeteilt. Im ersten Prozessfragment der Monte-Carlo Simulation wird *OpalMC* Dienst akquiriert, und im zweiten wird dieser Aufgerufen. Somit ergibt die Aufteilung des Simulationsprozesses, wie es auf der Abbildung 6.4 gezeigt ist.

Das Startfragment der Simulationsanwendung ist für das Initialisieren der Simulation zuständig und ist auf der Abbildung 6.5 dargestellt. Dieser Prozessfragment registriert die

Prozessinstanz beim *OpalManager*, und benachrichtigt diesen über den Start der Simulation. Anschließend fragt dieses Prozessfragment die Correlation Id beim *Ressourcen Manager* ab. Um an dieses Prozessfragment einen anderen Prozessfragment ankleben zu können wurde eine *frg:fragmentExit* Aktivität am Ende der Sequenz hinzugefügt.

Das *OpalMC* Prozessfragment (Abbildung 6.6) akquiriert den *OpalMC* dienst und enthält Anweisungen für die Fehlerbehandlung. Im Falle der erfolgreichen Akquirierung des Dienstes wird das weitere Vorgehen des Prozesses nicht spezifiziert und durch eine *frg:fragmentRegion* Aktivität erlaubt die entsprechende Logik einzufügen. Um dieses Prozessfragment in andere Prozessfragmente einfügen zu können, wurde am Anfang der Sequenz die *frg:fragmentEntry* Aktivität und am Ende der Sequenz die *frg:fragmentExit* Aktivität hinzugefügt. Die *frg:fragmentEntry* Aktivität definiert folgende Elemente die Mapping benötigen:

- Variable *corrIDResponse*: Diese Variable speichert die im Startfragment abgefragte Correlation Id, diese wird bei der Kommunikation mit dem *Ressourcen Manager* benutzt.
- Variable *runOpalMainProcRequest*: Diese Variable speichert die für die Simulation wichtigen Daten, wie Kontext Id, Anzahl der gewünschten Snapshots etc.
- Correlation Set *corrOpalManager*: Dieser Correlation Set wird benötigt, um die Nachricht für den Abbruch der Simulation korrelieren zu können.

Der *OpalMCCallback* Prozessfragment (Abbildung 6.7) ruft den *OpalMC* Dienst auf und empfängt die Snapshots der Simulation von diesem Dienst. Falls ein Snapshot empfangen wird, wird der Prozess für die Snapshot-Verarbeitung gestartet. Das Prozessfragment wartet bis die Ausführung des *OpalMC* Dienstes abgeschlossen ist und alle Snapshots verarbeitet wurden. Um diese Arbeitsschritte durchführen zu können, werden die von der *frg:fragmentEntry* Aktivität definierten Daten benötigt:

- Variable *runOpalMainProcRequest*: Diese Variable speichert die für die Simulation wichtigen Daten, wie Kontext Id, Anzahl der gewünschten Snapshots etc.
- Variable *corrIDResponse*: Diese Variable speichert die im Startfragment abgefragte Correlation Id, diese wird bei der Kommunikation mit dem *Ressourcen Manager* benutzt.
- Variable *ServiceTicketID*: Diese Variable speichert eine Ticket Id, die beim Akquirieren des Dienstes erstellt wird. Diese Id wird in der Nachricht für den Aufruf des *OpalMC* Services benötigt.
- Partner Link *OpalMCLink11*: Dieser Partner Link wird benutzt, um den *OpalMC* dienst Aufzurufen.
- Correlation Set *corrResourceManager*: Dieser Correlation Set wird benutzt um auf die Nachrichten im Fehlerfall reagieren zu können.
- Correlation Set *corrOpalManager*: Dieser Correlation Set wird benutzt, um die Nachricht über den Abbruch der Simulation korrelieren zu können.

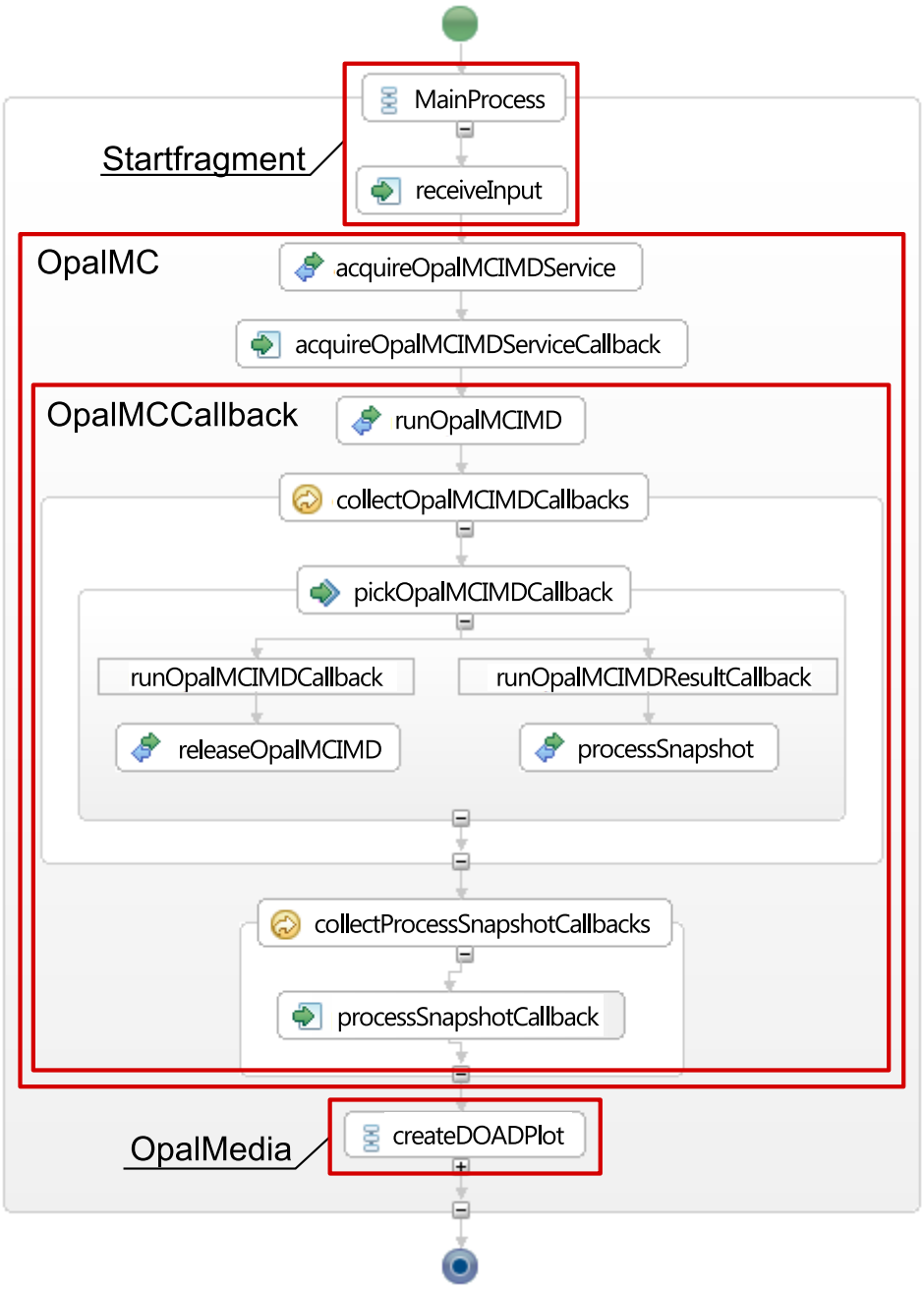

Abbildung 6.4: Aufteilung des Prozesses in Prozessfragmente.

Abbildung 6.5: Startfragment der Simulationsanwendung.

Der *OpalMedia* Prozessfragment (Abbildung 6.8) akquiriert und ruft den *OpalMedia* Dienst auf, um die Daten der Snapshots zu visualisieren. Zuletzt wird *OpalManager* über den Abschluss des Simulationsprozesses benachrichtigt. Die *frg:fragmentEntry* Aktivität am Anfang der Sequenz dieses Prozessfragments definiert folgende Elemente, die Mapping benötigen:

- Variable *corrIDResponse*: Diese Variable speichert die im Startfragment abgefragte Correlation Id, diese wird bei der Kommunikation mit dem *Ressourcen Manager* benutzt.
- Variable *ctxID*: Diese Variable speichert die Kontext Id der Simulation, diese wird benutzt, um auf die Dateien der Simulation zugreifen zu können.
- Variable *callbackBaseURL*: Diese Variable speichert URL des Servers (z.B. *http://localhost:8080/*), auf dem der Haupt-Prozess läuft und wird benutzt, um die Callback EPR zu erzeugen, die für das Akquirieren eines Services benutzt wird.
- Variable *simID*: Diese Variable speichert die Simulations Id und wird benutzt, um den Status der Simulation an *OpalManager* zu melden.
- Correlation Set *corrOpalManager*: Dieser Correlation Set wird benutzt, um die Nachricht über den Abbruch der Simulation empfangen zu können.

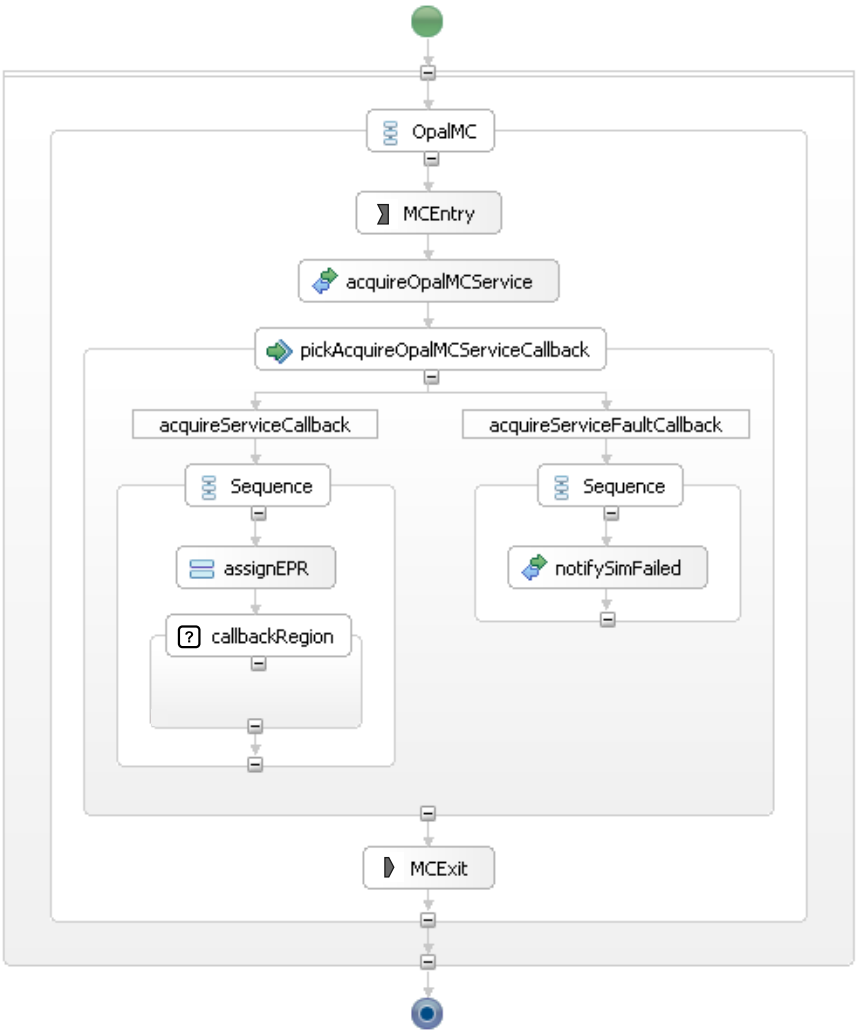

Abbildung 6.6: OpalMC Prozessfragment der Simulationsanwendung.

6.5 Zusammensetzung von Prozessfragmenten zur Laufzeit

Für das Starten einer Simulation wurde in [Hot10] die Anwendung *OpalClientApplication* entwickelt. Diese Anwendung wird weiterhin benutzt, da die Aufteilung des Prozesses in Prozessfragmente für diese Anwendung transparent ist. Beim Starten einer neuen Simulation wird das Startfragment des Haupt-Prozesses instantiiert und ausgeführt. Der Kontrollfluss wird dabei an der *frg:fragmentExit* Aktivität am Ende der Sequenz angehalten und es besteht die Möglichkeit ein Prozessfragment einzukleben. Die Prozessfragmente abgesehen von dem Startprozessfragment können zum beliebigen Zeitpunkt deployt werden, diese müssen jedoch vor dem Einkleben deployt sein.

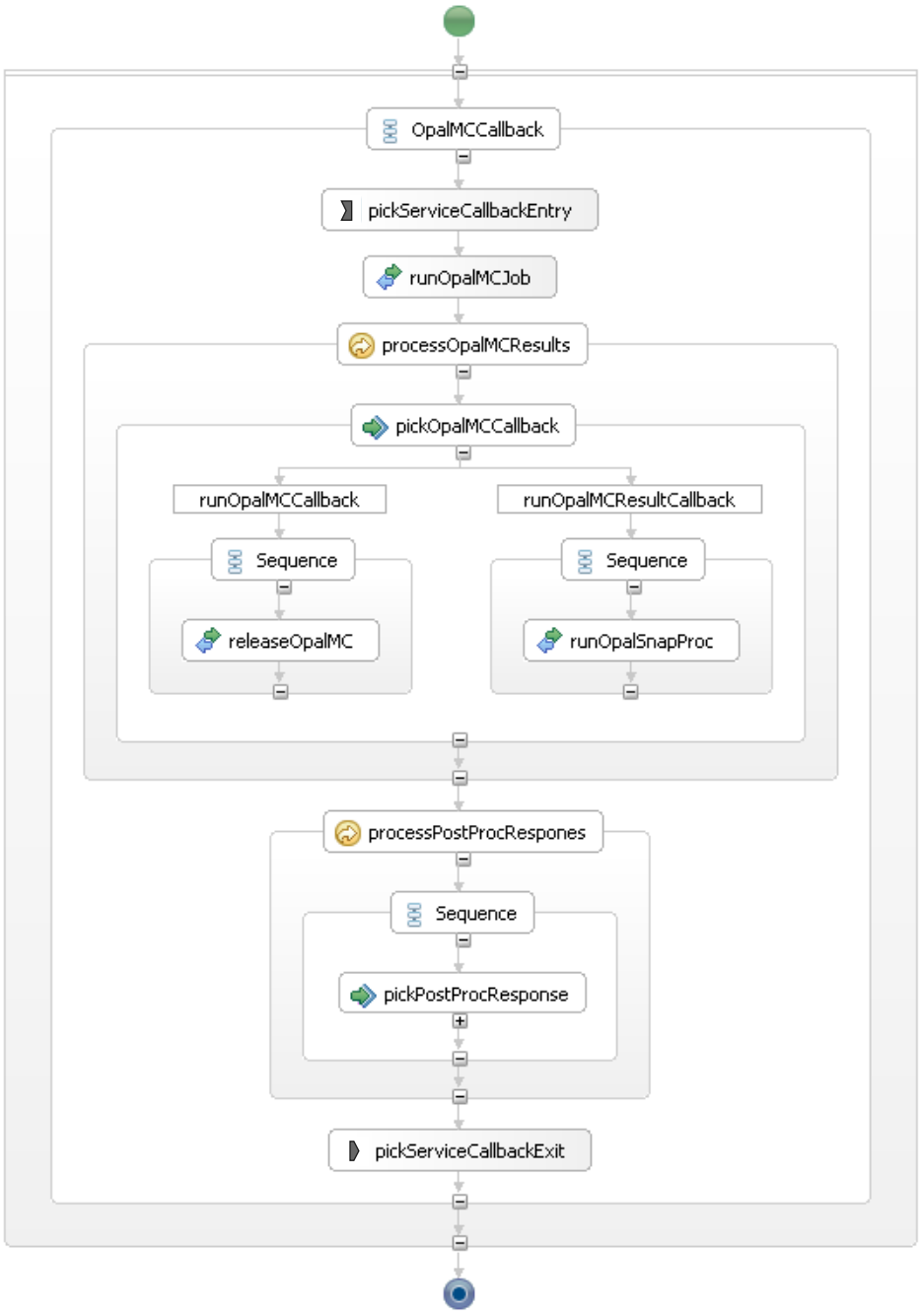

Abbildung 6.7: OpalMCCallback Prozessfragment der Simulationsanwendung.

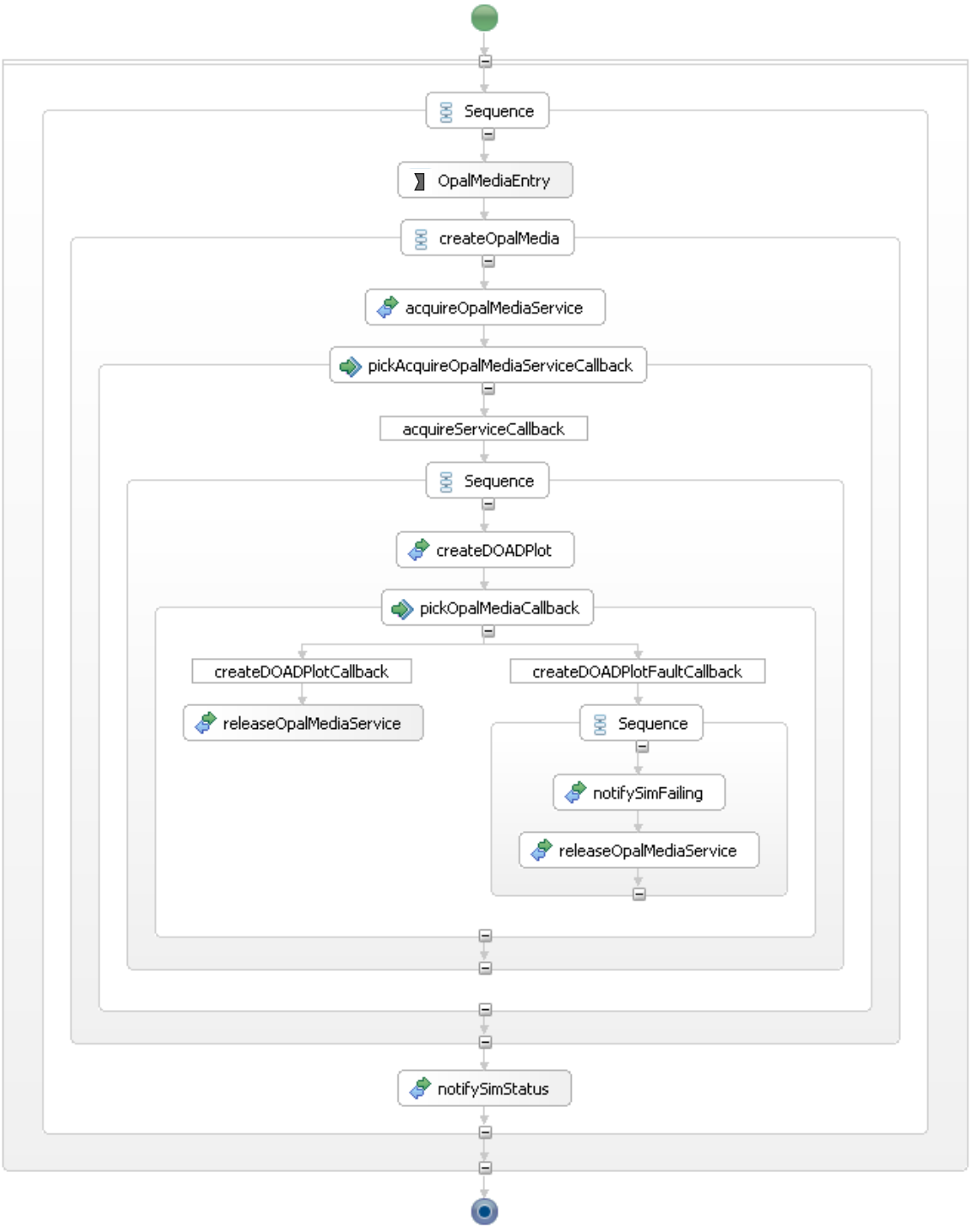

Abbildung 6.8: OpalMedia Prozessfragment der Simulationsanwendung.

Abbildung 6.9: Liste der Prozessinstanzen von Apache ODE

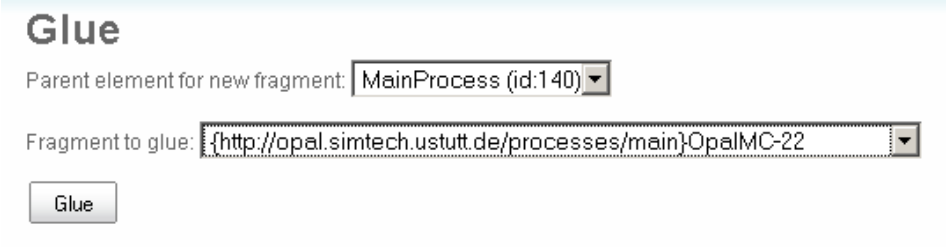

Abbildung 6.10: Einkleben von dem *OpalMC* Prozessfragment.

Um die Komposition der Prozessfragmente durchzuführen, kann Web-Interface von ODE benutzt werden. Dafür soll die erstellte Prozessinstanz in der Liste der Prozessinstanzen des Web-Interfaces gefunden werden und auf den Button *Composition* gedrückt werden (Abbildung 6.9). Somit gelangt man zur Maske für die Komposition von Prozessfragmenten (Abbildung 5.2).

Da nach der Initialisierung des Prozesses die Monte-Carlo Simulation ausgeführt werden soll, muss in die *frg:fragmentSequence* Aktivität mit dem Namen *MainProcess* das *OpalMC* Prozessfragment eingeklebt werden (Abbildung 6.10).

Nachdem das Prozessfragment eingeklebt wurde, müssen die *frg:fragmentExit* Aktivität mit dem Namen *mainProcessExit* und die *frg:fragmentEntry* Aktivität mit dem Namen *MCEntry* verbunden werden (Abbildung 6.11). Dabei müssen die Variablen *corrIDResponse*, *runOpalMainProcRequest* und der Correlation Set *corrOpalManager* gemappt werden. Um diesen Vorgang übersichtlich zu halten, wurden die dazugehörigen Variablen in den Prozessfragmenten gleich genannt. Somit müssen die gleichen Namen von Variablen und Correlation

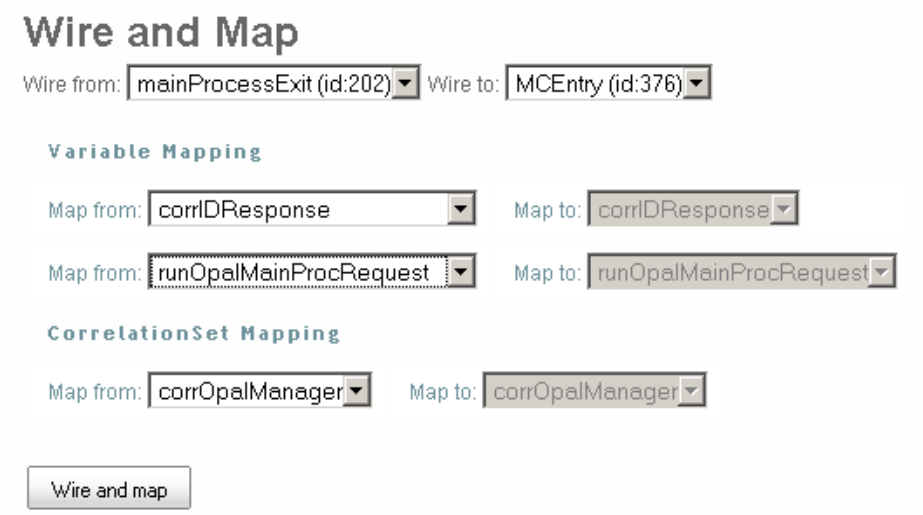

Abbildung 6.11: Verbinden von der *frg:fragmentExit* Aktivität des Startfragments und der *frg:fragmenEntry* Aktivität des *OpalMC* Prozessfragments.

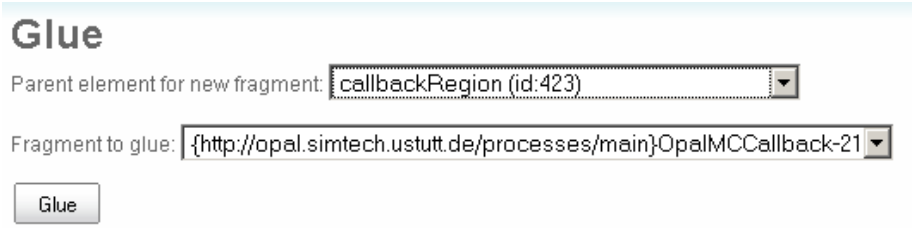

Abbildung 6.12: Einkleben von dem *OpalMCCallback* Prozessfragment.

Sets beim Mapping angegeben werden. Eine Ausnahme bilden die Partner Links, da das Vorhandensein von zwei Partner Links mit dem gleichen Namen innerhalb eines Prozesses bei der Apache ODE zu Problemen führt, wurde bei den gleichen Partner Links am Ende des Namens eine Zahl angehängt.

Nachdem die oben genannten Aktivitäten verbunden wurden, wird *OpalMC* Service akquiriert und man muss das *OpalMCCallback* Prozessfragment in die *frg:fragmentRegion* mit dem Namen *callbackRegion* einkleben (Abbildung 6.12).

Nach dem Einleben müssen die *frg:fragmentRegion* Aktivität mit dem Namen *callbackRegion* und die *frg:fragmentEntry* Aktivität mit dem Namen *pickServiceCallbackEntry* verbunden werden (Abbildung 6.13). Beim Mapping soll man, wie oben beschrieben ist, die gleichnamigen Elemente auswählen. Die Ausnahme dabei bilden die Variable mit dem Namen *ServiceTicketID* und der Partner Link mit dem Namen *OpalMCLink11*. Die Variable *ServiceTicketID* existiert in dem Prozessfragment *OpalMC* nicht. Aus diesem Grund soll beim Mapping dieser Variable die Variable mit dem Namen *AcquireServiceResponse* ausgewählt werden. Diese

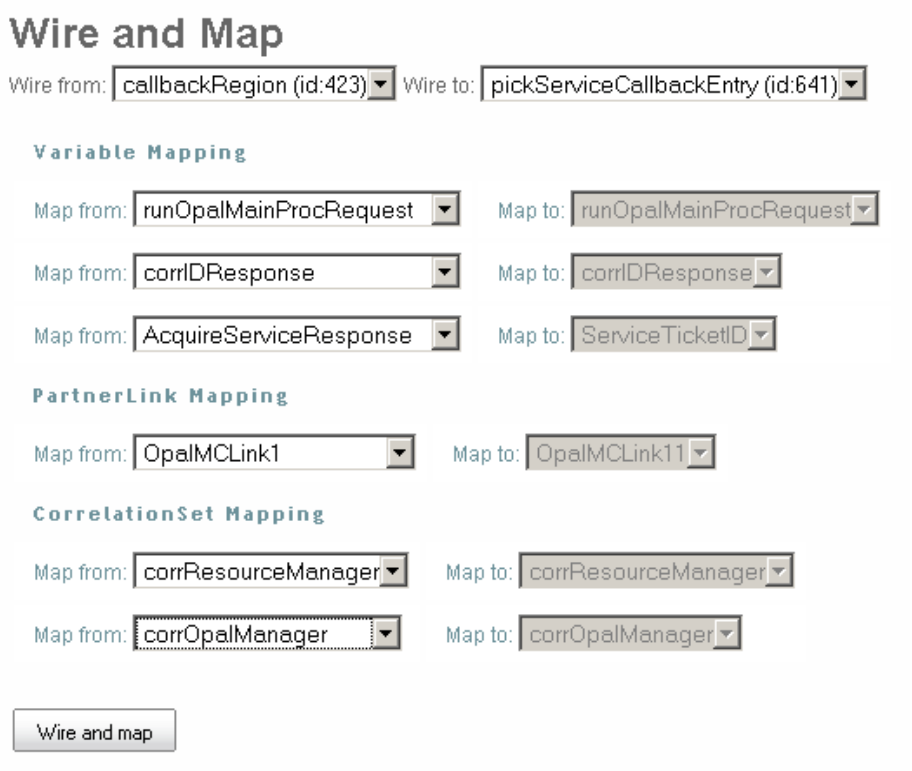

Abbildung 6.13: Verbinden von der *frg:fragmentRegion* Aktivität des Prozessfragments *OpalMC* und der *frg:fragmentExit* Aktivität des *OpalMCCallback* Prozessfragments.

enthält die benötigte Ticket Id. Die Konvertierung von den Datentypen wird dabei durch Mediation automatisch durchgeführt. Für den Mapping des Partner Links mit dem Namen *OpalMCLink11* soll der Partner Link mit dem Namen *OpalMCLink1* ausgewählt werden. Bei diesem Vorgang muss man jedoch beachten, dass der erhaltene Ticket nur eine gewisse Zeit gültig ist. Ist der Ticket abgelaufen, so muss der Simulationsprozess neu gestartet werden.

Nachdem die *frg:fragmentRegion* Aktivität mit der *frg:fragmentEntry* Aktivität verbunden wurden, beginnt die Ausführung der Monte-Carlo Simulation. Dies kann einige Zeit in Anspruch nehmen. Nachdem die Simulation durchgeführt wurde, und alle Snapshots analysiert wurden, muss man die Web-Seite aktualisieren um die aktuellen Daten in der Maske anzeigen zu lassen. Anschließend müssen die *frg:fragmentExit* Aktivität mit dem Namen *pickServiceCallbackExit* und die *frg:fragmentRegion* Aktivität mit dem Namen *callbackRegion* verbunden werden (Abbildung 6.14). Bei dieser Operation wird kein Mapping benötigt.

Als nächstes muss das Prozessfragment mit dem Namen *OpalMedia* in die *frg:fragmentSequence* Aktivität mit dem Namen *fragmentSequence-activity-line-214* eingeklebt werden (Abbildung 6.15). Zuletzt müssen die *frg:fragmentExit* Aktivität mit dem Namen *mcSuccessExit* und die *frg:fragmentEntry* Aktivität mit dem Namen *OpalMediaEntry* verbunden werden (Abbildung

Abbildung 6.14: Verbinden von der *frg:fragmentExit* Aktivität des Prozessfragments *OpalMC-Callback* und der *frg:fragmentRegion* Aktivität des *OpalMC* Prozessfragments.

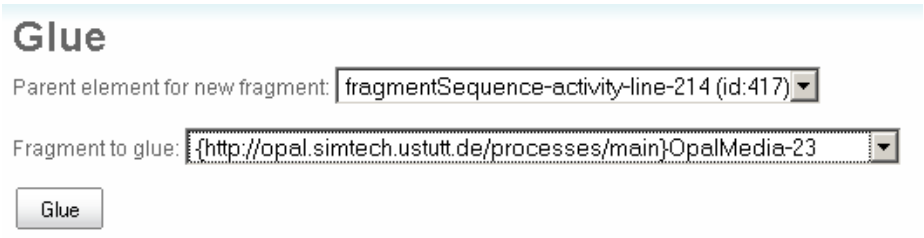

Abbildung 6.15: Einkleben von dem *OpalMedia* Prozessfragment.

6.16). Beim Mapping sollen die gleichnamigen Elemente ausgewählt werden. Die Ausnahme bilden die Variablen *ctxID*, *simID* und *callbackBaseURL*. Für den Mapping dieser Variablen soll die Variable mit dem Namen *runOpalMainProcRequest* angegeben werden. Die Konvertierung der Datentypen wird automatisch von der Mediator Komponente durchgeführt.

Wenn man nun die Liste der Simulationen in der *OpalClientApplication* Anwendung aktualisiert, wird der Zustand der aktuellen Simulation als *finished* angegeben, was einen über die erfolgreiche Durchführung der Simulation informiert.

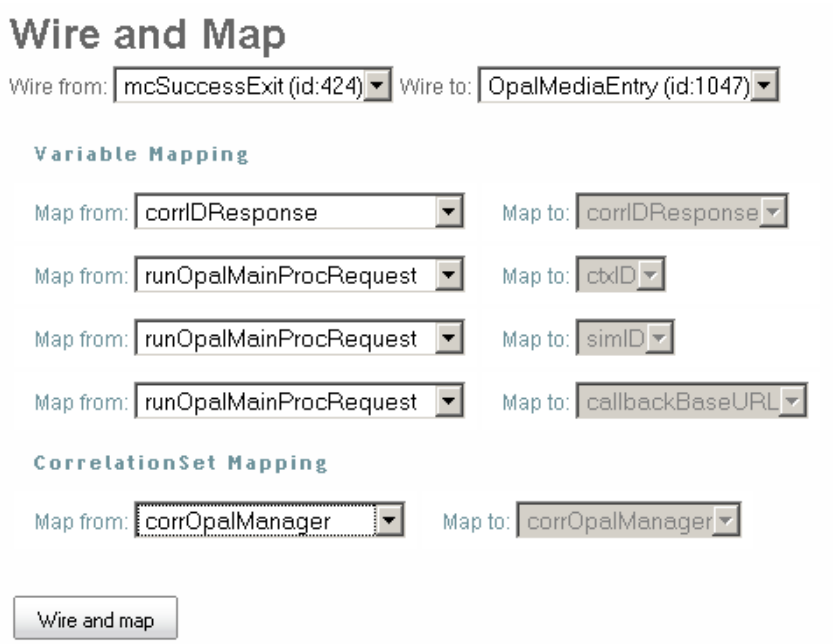

Abbildung 6.16: Verbinden von der *frg:fragmentExit* Aktivität des *OpalMC* Prozessfragments und der *frg:fragmenEntry* Aktivität des *OpalMedia* Prozessfragments.

7 Zusammenfassung und Ausblick

In dieser Diplomarbeit wurde BPEL um die Aktivitäten *frg:fragmentScope*, *frg:fragmentFlow*, *frg:fragmentSequence*, *frg:fragmentRegion*, *frg:fragmentExit* und *frg:fragmentEntry* erweitert. Diese Erweiterung ermöglicht die Definition von unvollständigen Prozessen, den Prozessfragmenten. Weiterhin wurde ein Konzept für die dynamische Zusammensetzung von Prozessfragmenten zur Laufzeit erarbeitet. Für den Datenaustausch zwischen den zusammengesetzten Prozessfragmenten wurde Konzept des Mappings von Variablen, Partner Links und Correlation Sets eingeführt. Da es bei dem Datenaustausch zwischen den zusammengesetzten Prozessfragmenten zu den Fällen kommen kann, bei den die auszutauschende Daten unterschiedliche Datentypen haben, wurde Mediation Funktionalität vorgestellt, die mit Hilfe von XSLT Transformationen die Datentypkonvertierung ermöglicht. An dem Beispiel einer generischen Architektur der Workflow Management Systemen wurden die für die Umsetzung dieses Konzeptes benötigten Erweiterungen gezeigt.

Um das Konzept auf die möglichen Schwachstellen zu prüfen, wurde das bestehende WFMS, die Apache ODE [ODEa], entsprechend erweitert. Als Beispielprozess wurde ein wissenschaftlicher Workflow aus der Diplomarbeit [Hot10] verwendet. Dieser Prozess wurde in Prozessfragmente aufgeteilt und zur Laufzeit wieder zusammengesetzt.

Ein Erkenntnis der Aufteilung des Prozesses und deren Zusammensetzung zur Laufzeit war, dass es ungünstig ist, die zusammengehörige *invoke* und *receive* bzw. *pick* Aktivitäten in unterschiedliche Prozessfragmente aufzuteilen, da die durch *receive* bzw. *pick* Aktivität zu empfangene Nachricht vor dem Einkleben des entsprechenden Prozessfragmentes an das WFMS geschickt und von dem WFMS ignoriert werden kann.

Ausblick

Das in dieser Diplomarbeit entwickelte Konzept beschränkt sich auf die manuelle Runtime Komposition, dieses Konzept kann jedoch erweitert werden, um die Komposition abhängig von dem Kontext des Prozesses automatisch durchführen zu können. Weiterhin kann das in dieser Arbeit vorgestellte Konzept um die in der Arbeit [ELU10] vorgestellten Transaktionskonzepte erweitert werden.

Außerdem kann der in dieser Arbeit auf Basis von Apache ODE entwickelte Prototyp, um die Prüfung der Korrektheit der Prozessfragmente erweitert werden. So muss z.B. sichergestellt werden, dass die *frg:fragmentEntry* und *frg:fragmentExit* Aktivitäten sich nicht innerhalb von Standard-BPEL-Aktivitäten befinden. Weiterhin ist es in dem Prototyp möglich durch Verbinden von *frg:framgentExit* und *frg:fragmentEntry* Aktivitäten im Prozess Zyklen zu erzeugen,

sowie diese Aktivitäten über die Schleifengrenzen hinweg zu verbinden. Bei dem Verbinden von *frg:fragmentExit* und *frg:fragmentEntry* Aktivitäten soll das WFMS die Komposition auf diese Situationen prüfen und wenn notwendig die Komposition verhindern.

Das in dieser Arbeit vorgestellte Konzept wurde mit Hilfe eines wissenschaftlichen Workflows überprüft, dies reicht jedoch im allgemeinen nicht, um über Praxistauglichkeit des Konzeptes eine Aussage zu treffen. Aus diesem Grund werden weitere Erfahrungen aus der Praxis benötigt, um den ausgearbeiteten Konzept zu beurteilen und möglicherweise zu verbessern.

Literaturverzeichnis

[ATEA06] M. Adams, Ter, D. Edmond, W. van der Aalst. Worklets: A Service-Oriented Implementation of Dynamic Flexibility in Workflows. pp. 291–308. 2006. doi: 10.1007/11914853_18. URL http://dx.doi.org/10.1007/11914853_18. (Zitiert auf Seite 10)

[AXI] Apache Software Foundation. Apache Axis2/Java. URL http://axis.apache.org/axis2/java/core/. (Zitiert auf Seite 57)

[BPE05] WS-BPEL Extension for Sub-processes – BPEL-SPE, 2005. URL http://xml.coverpages.org/BPEL-SPE-Subprocesses.pdf. (Zitiert auf den Seiten 9 und 37)

[BPE07] Web Services Business Process Execution Language Version 2.0, 2007. URL http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf. (Zitiert auf den Seiten 14, 21, 24, 25 und 46)

[CF04] C. Courbis, A. Finkelstein. Towards an Aspect Weaving BPEL engine. In Y. Coady, D. H. Lorenz, editors, *the Third AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*. Lancaster, United Kingdom, 2004. (Zitiert auf Seite 10)

[ddS] Apache ODE deployment descriptor Schema. URL http://svn.apache.org/viewvc/ode/trunk/bpel-schemas/src/main/xsd/dd.xsd?view=markup. (Zitiert auf Seite 59)

[ELU10] H. Eberle, F. Leymann, T. Unger. Transactional Process Fragments - Recovery Strategies for Flexible Workflows with Process Fragments. In *Proceedings of APSCC 2010*, pp. 1–8. IEEE Xplore, 2010. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2010-100&engl=0. (Zitiert auf den Seiten 9, 10, 11 und 97)

[EUL09] H. Eberle, T. Unger, F. Leymann. Process Fragments. In R. Meersman, T. Dillon, P. Herrero, editors, *On the Move to Meaningful Internet Systems: OTM 2009, Part I*, volume 5870 of *Lecture Notes in Computer Science*, pp. 398–405. Springer, 2009. doi: 10.1007/978-3-642-05148-7_29. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2009-73&engl=0. (Zitiert auf den Seiten 10, 34 und 35)

[HBR08] A. Hallerbach, T. Bauer, M. Reichert. Issues in Modeling Process Variants with Provop. In *Business Process Management Workshops*, pp. 56–67. 2008. (Zitiert auf Seite 10)

[Hot10] S. Hotta. Ausführung von Festkörpersimulationen auf Basis der Workflow Technologie, 2010. (Zitiert auf den Seiten 81, 82, 83, 84, 85, 89 und 97)

[JBI05] Java Business Integration (JBI) 1.0, 2005. URL http://www.jcp.org/aboutJava/communityprocess/final/jsr208/index.html. (Zitiert auf Seite 57)

[KS09] M. Keith, M. Schincariol. *Pro JPA 2: Mastering the Java Persistence API*. Apress, Berkely, CA, USA, 1st edition, 2009. (Zitiert auf den Seiten 25, 26 und 27)

[LR00] F. Leymann, D. Roller. *Production Workflow - Concepts and Techniques*. PTR Prentice Hall, 2000. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=BOOK-2000-01&engl=0. (Zitiert auf den Seiten 13, 14, 15 und 53)

[ODEa] Apache Software Foundation. Apache Orchestration Director Engine (ODE). URL http://ode.apache.org/. (Zitiert auf den Seiten 11, 57, 58, 59, 60 und 97)

[ODEb] Apache Software Foundation. WS-BPEL 2.0 Specification Compliance of Apache ODE. URL http://ode.apache.org/ws-bpel-20-specification-compliance.html. (Zitiert auf Seite 61)

[SAL+10] D. Schumm, T. Anstett, F. Leymann, D. Schleicher, S. Strauch. Essential Aspects of Compliance Management with Focus on Business Process Automation. In *Proceedings of the 3rd International Conference on Business Processes and Services Computing (BPSC) – INFORMATIK 2010, 27-28 September 2010, Leipzig, Germany*, volume P-177 of *Lecture Notes in Informatics (LNI)*, pp. 127–138. Gesellschaft für Informatik e.V. (GI), 2010. (Zitiert auf den Seiten 10, 34, 35, 36 und 37)

[SOA07] SOAP Version 1.2 Part 0: Primer (Second Edition), 2007. URL http://www.w3.org/TR/2007/REC-soap12-part0-20070427/. (Zitiert auf Seite 17)

[STF+10] E. Sebastian, A. Hilliger von Thile, M. Flehmig, P. Tröger, B. Rudolph, B. Stumm, M. Lipp, P. Sauter, J. Vajda, W. Dostal, M. Jeckle. *Service-orientierte Architekturen mit Web Services: Konzepte - Standards - Praxis*. Spektrum Akademischer Verlag, 2010. (Zitiert auf den Seiten 15, 16, 18, 19 und 21)

[TDGS06] I. J. Taylor, E. Deelman, D. B. Gannon, M. Shields. *Workflows for e-Science: Scientific Workflows for Grids*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. (Zitiert auf Seite 9)

[Tel10] S. Telezhnikov. Prozessfragmente: Metamodell und Ausführung, 2010. (Zitiert auf den Seiten 10, 11, 33, 34 und 77)

[Tid08] D. Tidwell. *XSLT: Mastering XML Transformations*. O'Reilly, Sebastopol, CA, 2. edition, 2008. (Zitiert auf Seite 27)

[WCL+05] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, D. F. Ferguson. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005. (Zitiert auf den Seiten 18, 19 und 22)

[WOd09] I. Wassink, M. Ooms, P. V. van der. Designing Workflows on the Fly Using e-BioFlow. In L. Baresi, C.-H. Chi, J. Suzuki, editors, *Service-Oriented Computing*, Lecture Notes in Computer Science 5900, pp. 470–484. Berlin, 2009. URL http://doc.utwente.nl/68648/. (Zitiert auf Seite 9)

[WSA04] Web Services Architecture, 2004. URL http://www.w3.org/TR/ws-arch/. (Zitiert auf Seite 16)

[WSD01] Web Services Description Language (WSDL) 1.1, 2001. URL http://www.w3.org/TR/wsdl. (Zitiert auf den Seiten 16 und 17)

[WSD07] Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language, 2007. URL http://www.w3.org/TR/wsdl20/. (Zitiert auf Seite 17)

[XPAa] XML Path Language (XPath) 2.0 (Second Edition). URL http://www.w3.org/TR/xpath20/. (Zitiert auf Seite 28)

[XPAb] XML Path Language (XPath) Version 1.0. URL http://www.w3.org/TR/xpath/. (Zitiert auf Seite 28)

[XSL] XSL Transformations (XSLT) Version 2.0. URL http://www.w3.org/TR/xslt20/. (Zitiert auf Seite 27)

Alle URLs wurden zuletzt am 13.07.2011 geprüft.

Erklärung

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

(Alex Hummel)