Institut für Parallele und Verteilte Systeme
Abteilung Anwendersoftware
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3344

Erweiterung und Evaluation eines Prototyps für eine enge Integration zwischen Datenbank- und Workflow-Engines

Christian Ageu

Studiengang:	Informatik
Prüfer:	PD Dr. rer. nat. habil. Holger Schwarz
Betreuer:	Dipl.-Inf. Peter Reimann
begonnen am:	17. Mai 2012
beendet am:	7. Dezember 2012
CR-Klassifikation:	D.2.11, H.2.3, H.2.4, H.2.8, H.4.1

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

1 Einführung

Durch die Wandlung von Information zu einem wirtschaftlichen Gut haben informationsverarbeitende Prozesse einen hohen Stellenwert erlangt. Seit Anbruch des Informationszeitalters in den 1970er Jahren sind die Ansprüche an Kommunikationsnetzwerke und elektronischer Datenverarbeitung stark gestiegen. Der von diesen Ansprüchen vorangetriebene wissenschaftliche und damit einhergehende technologische Fortschritt hat im Laufe der letzten Jahrzehnte hochspezialisierte und damit sehr effiziente Lösungen und Technologien auf den entsprechenden Sektoren hervorgebracht. Auch in Wissenschaft und Forschung kommt der Informationsverarbeitung auf vielen Fachgebieten eine immer größer werdende Bedeutung zu.

Um Informationen in Form von Daten auf Datenträgern zu speichern und wieder abrufen zu können, wurden *Datenbankverwaltungsysteme* (engl.: *database management systems*, DBMS) entwickelt, die teilweise seit über 30 Jahren bestehen, weiterentwickelt und dadurch verbessert werden. Solche, mittlerweile hochkomplexe Systeme ermöglichen es heutzutage große Datenmengen verhältnismäßig einfach, sicher und äußerst effizient zu verwalten und zu verarbeiten [KE06].

Zur elektronischen Verarbeitung von Informationen wurde das Prinzip der *Geschäftsprozesse* (engl.: *business processes*) entwickelt, nach dem Abläufe modelliert und ausgeführt werden können. Geschäftsprozesse beschreiben eine Folge von Einzeltätigkeiten, die schrittweise ausgeführt werden, um ein geschäftliches oder betriebliches Ziel zu erreichen [LR00]. Der Teil eines Geschäftsprozesses, der auf einem Computer ausgeführt wird, wird als *Workflow* bezeichnet, da der Gesamtablauf als fließende Hintereinanderausführung kleinerer Aktionen realisiert wird. Workflows werden zur Beschreibung von Vorgängen im wirtschaftlichen Umfeld, aber auch bei computergestützten Simulationen in Wissenschaft und Forschung, eingesetzt [TDG07]. Dabei werden verschiedene Teilprozesse in ein Ablaufschema überführt, welches den Gesamtprozess als Ausführungsgraph dieser Teilprozesse modelliert. Der so entstandene Gesamtprozess kann wiederum als gesonderter Arbeitschritt aufgefasst und als Teilprozess in größere Abläufe integriert werden. Durch dieses Konzept können einmal fertiggestellte Workflows in anderen Workflows wiederverwendet werden. Zur Modellierung, Ausführung und Ausführungsüberwachung solcher Workflows wurden *Workflowmanagementsysteme* (WfMS) enwickelt, die die Umsetzung dieser Aufgaben auf einem Computer ermöglichen.

1.1 Motivation

Um Daten während einer Workflowausführung persistent zu halten, verwenden nahezu alle WfMS entweder eigene (integrierte) oder extern angebundene Datenbanksysteme (DBS). Ein DBS setzt sich aus einer Datenbank und einem darauf operierenden Datenbankmanagementsystem zusammen. Während einer Workflowausführung werden in der Datenbank unter anderem die Variableninhalte der verwalteten Prozesse abgelegt. Die Daten in den Variablen eines Workflows repräsentieren die Informationen, die von ihm verarbeitet werden. Das Laden, Verarbeiten und Speichern dieser Daten stellt den produktiven Teil eines informationsverarbeitenden Geschäftsprozesses dar. Da solche Daten je nach Einsatzgebiet sehr groß werden können, kann die Verwaltung der Variableninhalte durch das WfMS die Ausführung eines Workflows negativ beeinflussen oder sogar zur Überlastung des Systems und damit dem Scheitern der Ausführung führen.

Um in Zukunft große Datenmengen innerhalb von Workflows effizienter und zuverlässiger handhaben zu können, ist die Idee entstanden, die Funktionen des DBS auch intensiv für die Workflowausführung einzusetzen, anstatt damit nur die Persistenz sicherzustellen [RSM11]. Dabei sollen datenintensive Operationen, auf die das DBS spezialisiert ist, dem WfMS abgenommen werden und es damit entlasten. Die Ziele sind dabei schnelleres Laufzeitverhalten, geringerer Speicherverbrauch, höhere Stabilität sowie Skalierbarkeit der Datenzugriffe während der Workflowausführung.

Im Rahmen vorangegangener Arbeiten [Wag11],[RSM11] wurden bereits Konzepte zur Verbesserung der Workflowausführung durch Integration des DBS bei der Workflowausführung erarbeitet und entsprechende Funktionalitäten prototypisch umgesetzt sowie evaluiert. Dabei sind sowohl einige Aspekte der Integration unbehandelt geblieben, als auch durch die prototypische Umsetzung neue Problemstellungen identifiziert worden. So kam es beispielsweise zu ungewünschten Leistungseinbußen durch die gemischte Datenverarbeitung im DBS einerseits und im WfMS andererseits. In dieser Arbeit sollen nun einige der unbehandelten Aspekte und Problemstellungen untersucht, sowie Lösungsansätze zur weiteren Verlagerung von Datenoperationen in das DBS umgesetzt und evaluiert werden.

1.2 Konventionen und rechliche Hinweise

Begriffe, die im weiteren Verlauf der Arbeit abgekürzt werden, werden bei der erstmaligen Verwendung ausgeschrieben und die verwendete Abkürzung dahinter, innerhalb runder Klammern, angegeben. Aus dem englischen übersetzte Fachbegriffe werden bei ihrer erstmaligen Verwendung zusätzlich mit ihrer englischen Bezeichnung in Klammern angegeben, um den Bezug auf die englischsprachige Literatur zu ermöglichen.

In dieser Arbeit kam Software zum Einsatz, die nicht öffentlich zur Verfügung steht und für die Lizenzen erworben werden müssen. Für die Evaluation des in dieser Arbeit entstandenen Prototyps wurde eine entsprechende Lizenz erworben. Dies betrifft insbesondere:

- IBM DB2 V10.1 Advanced Enterprise Server Edition

Diese Arbeit enthält eine Datenbank-Auswertung. Der Autor dieser Arbeit hat die Vorbereitung und Ausführung dieser Auswertung mit besonderer Vorsicht durchgeführt. Trotzdem kann der Autor mögliche Fehler, die hierbei entstanden sind, nicht ausschließen. Aus diesem Grund übernimmt der Autor keine Verantwortung für die Korrektheit und Vollständigkeit der gesamten Auswertung und der daraus geschlossenen Erkenntnisse.

1.3 Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Grundlagen: Hier werden grundlegende informationstechnische Begriffe erläutert und die in der Arbeit verwendeten Technologien vorgestellt.

Kapitel 3 – Nutzung von Datenbanken in WfMS: Hier werden durch vorangegangene Arbeiten hervorgebrachte Konzepte, Umsetzungen und Ergebnisse zur stärkeren Integration von Datenbanken in WfMS vorgestellt, beschrieben und analysiert.

Kapitel 4 – Konzeptionelle Erweiterungen: Hier werden die in dieser Arbeit entwickelten konzeptionellen Erweiterungen des bestehenden Systems vorgestellt.

Kapitel 5 – Architektur von Apache ODE: Hier werden die Teile der Softwarearchitektur der Workflowengine Apache ODE beschrieben, die für diese Arbeit relevant sind. Außerdem werden die durch die Vorarbeiten bereits implementierten Veränderungen an der Architektur vorgestellt.

Kapitel 6 – Implementierung der konzeptionellen Erweiterungen: Hier werden implementierungstechnische Umsetzungen der in Kapitel 4 vorgestellten Erweiterungen im Detail beschrieben.

Kapitel 7 – Evaluation des erweiterten Prototyps: Hier werden Aufbau, Durchführung und Ergebnisse von Messungen an den Umsetzungen der Erweiterungen vorgestellt.

Kapitel 8 – Zusammenfassung und Ausblick: Hier wird die Arbeit zusammengefasst. Es werden abschließende Kommentare abgegeben und auf weitere Themen hingewiesen, die sich im Verlauf der Arbeit aufgetan haben, jedoch nicht behandelt wurden.

2 Grundlagen

In diesem Kapitel werden grundlegende informationstechnische Begriffe erläutert und Technologien vorgestellt, die bei der Modellierung und Ausführung von Workflows im allgemeinen eine wichtige Rolle spielen oder im weiteren Verlauf dieser Arbeit benötigt werden. Dabei werden die ersten Probleme und Konflikte vorgestellt, für die im Rahmen dieser Arbeit Lösungen gefunden werden sollen.

2.1 Extensible Markup Language

Die e**X**tensible **M**arkup **L**anguage [xmla], kurz XML, ist eine weit verbreitete *Markup-Sprache* zur Darstellung von hierarchisch strukturierten Daten in Textform. Mit Hilfe von Markup-Sprachen lassen sich in einem Dokument neben Informationen auch deren Metainformationen bzw. Annotierungen speichern.

Durch den hohen Verbreitungsgrad der Sprachen existieren viele Werkzeuge, mit der sich eine XML-basierte Datenbasis realisieren lässt. Sowohl die in dieser Arbeit verwendete Beschreibungssprache für Workflows *BPEL* (s. Abschnitt 2.4), als auch alle betrachteten Workflowdaten verwenden XML als Grundlage zur Speicherung und zum Austausch von Informationen. In den folgenden Unterabschnitten werden Aufbau und Verarbeitung von XML-Dokumenten näher beschrieben.

2.1.1 Dokumentstruktur

In XML werden Informationen von sogenannten *Tags* eingeschlossen. Tags beinhalten die zu der von ihnen umschlossenen Information gehörenden Metainformationen und liefern den namen eines Elements. Tags werden in XML durch spitze Klammern gekennzeichnet (`<TAG>`), wobei schließende Tags einen zusätzlichen Schrägstrich vor dem Tagnamen besitzen (`</TAG>`). In einem öffnenden Tag können zusätzlich Attribute des Elements definiert werden. Solche Definitionen haben die Form `<TAG ATTRIBUTNAME=‘‘WERT‘‘>`. Im Gegenssatz zu anderen Markup-Sprachen, wie zum Beispiel HTML, sind in XML Tag- und Attributnamen sowie deren Bedeutung für die Informationsverarbeitung nicht vordefiniert. Diese können für jedes XML-Dokument in einer separaten Spezifikationsdatei definiert werden. Hierfür steht eine Reihe von Spezifikationssprachen zur Verfügung, zwei der gebräuchlichsten sind dabei *Document Type Definition (DTD)* [xmla] sowie das neuere *XML Schema* [xmlb].

Ein XML-Dokument besteht aus drei Teilen:

```xml
<?xml version="1.0" encoding="UTF-8"?>

<Diplomarbeit>
  <Autor>
    <Name>
      Christian Ageu
    </Name>
  </Autor>

  <Dokument type="Diplomarbeit">

    <Titel Sprache="en">
      Extension and evaluation of a prototype for a tight integration of database and
          workflow engines
    </Titel>

    <Titel Sprache="de">
      Erweiterung und Evaluation eines Prototyps für eine enge Integration zwischen
          Datenbank- und Workflow-Engines
    </Titel>

    Diese Arbeit befasst sich mit Workflow- und Datenbanksystemen.
    ...
  </Dokument>
</Diplomarbeit>
```

Listing 2.1: Beispiel XML-Datei „diplomarbeit.xml“

- **Präambel (optional)**: Hier können u.a. die verwendete XML-Version sowie die Zeichenkodierung des Dokuments angegeben werden.
- **Schema (optional)**: Hier können im Dokument verwendete Schemainformationen (DTD bzw. XML Schema) angegeben werden.
- **Wurzelelement**: Ein wohlgeformtes XML-Dokument besteht aus genau einem Wurzelelement, welches durch ein öffnendes und das entsprechende schließende Tag gekennzeichnet ist.

Ein Element kann Attribute, Textinhalt sowie beliebig geschachtelte, weitere Unterelemente beinhalten. Dadurch entsteht eine Baumstruktur, die durch bestehende und standardisierte Techniken verarbeitet werden kann, z.B. anhand des *Document Object Model* (DOM) [dom]. Bei der Repräsentation mit Hilfe von XML werden Daten üblicherweise als Textinhalt in den Blattknoten eines Dokuments abgespeichert, wobei der Pfad vom Wurzelknoten zum Blattknoten die Bezeichnung eines Datums liefert. Listing 2.1 zeigt das Beispiel einer wohlgeformten XML-Datei.

2.1.2 Verarbeitung

XML bietet umfangreiche Möglichkeiten zur Verwaltung, Abfrage und Manipulation von Daten. Mit XQuery [xqu] steht eine mächtige Sprache zur Abfrage bzw. Transformation von

XML-Daten zur Verfügung. Dabei wird XPath [xpa] verwendet, um auf einzelne Elemente eines XML-Dokuments zuzugreifen.

XPath

Bei *XPath* handelt es sich um eine Abfragesprache für XML-Dokumente. Mit ihrer Hilfe lassen sich Teile eines XML-Dokuments adressieren, indem dessen Bestandteile als unterschiedliche Knoten aufgefasst werden. Dabei werden unter anderem Text-, Attribut- und Unterknoten unterschieden, die auf Anfrage durch ein XPath-Verarbeitungsprogramm zurückgegeben werden können.

Ein XPath-Ausdruck setzt sich aus einem oder mehreren sogenannten *Lokalisierungsschritten* zusammen, die den Pfadausdrücken des UNIX-Betriebssystems ähneln. Mit Hilfe definierter *Achsen* kann in Lokalisierungsschritten durch ein XML-Dokument navigiert und ein bestimmter Knoten zurückgegeben werden. Ein Lokalisierungsschritt hat die Form

```
/Achse::Knotentest[Prädikat][...]
```

Mögliche Achsen sind dabei unter anderem *child* (alle Unterknoten), *parent* (alle Elternknoten), *self* (der Knoten selbst) und *attribute* (Attributknoten). Ein Knotentest schränkt die Elementauswahl einer Achse ein. Durch Angabe von Prädikaten kann die Auswahl noch weiter eingeschränkt werden. So wählt beispielsweise der Lokalisierungsschritt

```
/child::Titel[1]
```

den ersten Unterknoten mit den Namen „Titel“ aus und gibt diesen zurück. Dies entspricht im Beispieldokument dem englischen Titel der Diplomarbeit. Die Pfadselektion durch XPath-Ausdrücke bildet die Grundlage für verschiedene XML-Technologien, unter anderem das als nächstes vorgestellte XQuery. Für detailiertere Informationen über XPath und seine Funktionen sei auf die XPath-Spezifikation [xpa] verwiesen.

XQuery

Mit XQuery [xqu] wurde eine Abfragesprache für XML-Dokumente entwickelt, die zusätzlich zur Funktionalität von XPath noch weitere, komplexere Auswertungen von XML-Dokumenten ermöglicht. Insbesondere lassen sich mit ihrer Hilfe mehrere Knoten aus verschiedenen Dokumenten gleichzeitig verarbeiten und beispielsweise aggregieren. Dabei werden Operationen auf XML-Daten ähnlich der Abfragesprache SQL für relationale Datenbanken eingeführt. Grundlage einer XQuery-Abfrage bildet ein sog. *FLWOR*-Ausdruck (For, Let, Where, Order by, Return). Beispielsweise liefert

```
for $x in doc("diplomarbeit.xml")/Diplomarbeit
let $d := $x/Dokument
where $d/@type="'Diplomarbeit"']
order by $x/Titel
return {$x/Titel/text()}
```

die alphabetisch sortierten Textknoten aller Elemente mit dem Namen „Titel“. Mit *for* wird zunächst der zu verarbeitende Dokumententeil an eine Variable *x* gebunden. Mittels der *let*-Klausel können weitere Variablen für die Verwendung innerhalb des Ausdrucks definiert werden. Durch Angabe der *where*-Klausel wird eine Selektion der Elemente durchgeführt. Durch *order by* werden die selektierten Knoten nach ihrem Titel sortiert. Die *return*-Anweisung gibt schließlich die Textinhalte der sortierten Knoten als Resultat des FLWOR-Ausdrucks zurück. Es ist zu beachten, dass die XQuery-Spezifikation keine Möglichkeit zur Manipulation von bestehnden XML-Dokumenten vorsieht. Dies ist nur über einen Umweg möglich, indem bestehende Dokumente durch neu erzeugte XML-Dokumente vollständig überschrieben werden.

2.2 SOA und Webservices

Um den Anforderungen komplexer, heterogener Geschäftsumgebungen gerecht zu werden, wurden Technologien entwickelt, die die Verwaltung, Pflege und Integration von Geschäftsprozessen standardisiert und vereinfacht haben. In den folgenden Unterabschnitten weden Konzepte vorgestelt, auf denen sich diese Technologien stützen.

2.2.1 Service Oriented Architecture

Die Service Oriented Architecture (SOA) ist ein Architekturmuster, bei dem Geschäftsprozesse mit Hilfe von Diensten realisiert werden. Anstatt einen Gesamtprozess als Ganzes zu implementieren, wird versucht, ihn in kleinere Teilprozesse aufzuspalten. Ein so gekapselter Teilprozess bildet einen *Dienst*, der isoliert ausgeführt eine bestimmte Aufgabe erfüllt. Mehrere Dienste zusammen bilden bei koordinierter (*orchestrierter*) Ausführung den Gesamtprozess.

Einer SOA liegt ein System aus *Konsument*, *Anbieter* und einem *Verzeichnis* zugrunde. Besonderes Merkmal dieser Architektur ist die lose Kopplung der Dienste an die aufrufenden Prozesse: Ein Konsument ist nicht an einen bestimmten Dienst gebunden, sondern findet über das Verzeichnis anhand einer Beschreibung einen Dienst, der die für ihn benötigte Funktionalität zur Verfügung stellt. Der besondere Vorteil bei dieser Architektur liegt in der Wiederverwendbarkeit der einzelnen Dienste. Die Aufteilung von Geschäftsprozessen in kleinere, gekapselte Dienste bietet auch eine leichtere Überschaubarkeit für den Entwickler und damit einfachere Erstellung und Pflege der einzelnen (Teil-)Prozesse.

2.2.2 Webservices

Das Konzept der *Webservices* wird durch eine Reihe von Standards des *W3C*[1] (World Wide Web Consortium) sowie von *OASIS*[2] (Organization for the Advancement of Structured Information Standards) beschrieben. Webservices bieten eine Möglichkeit, nach dem SOA-Prinzip fertiggestellte Dienste plattformunabhängig in einem Netzwerk über das HTTP-Protkoll bereitzustellen.

Möchte ein Anbieter einen Dienst als Webservice zur Verfügung stellen, so meldet er seinen Dienst und dessen Schnittstellen, beschrieben in der XML-basierten *Web Services Description Language* (WSDL), bei einem Verzeichnisserver, der *UDDI Registry* (*Universal Description, Discovery, and Integration*), an. Ein interessierter Konsument kann diesen Webservice daraufhin über die UDDI Registry finden (*discover*), und ihn anschließend aufrufen (*invoke*). Der Nachrichtenaustausch zwischen Anbieter und Konsument erfolgt mittels ebenfalls XML-basierter *SOAP*-Nachrichten (*Simple Object Access Protocol* [**?**]). Die Rollen des Anbieters und des Konsumenten übernehmen dabei die Prozesse, die den Dienst bereitstellen bzw. benötigen. Da die aufrufenden Prozesse auch selbst Webservices sein können, wird mit Hilfe dieses Konzeptes eine hohe Kombinationsmöglichkeit und Wiederverwendung der Webservices ermöglicht.

2.3 Workflows und Workflowmanagementsysteme

Ein *Workflow* (Arbeitsablauf) ist die Verknüpfung verschiedener Arbeitsschritte zu einem Gesamtablauf, basierend auf kausalen Abhängigkeiten oder Datenabhängigkeiten zwischen den Teilschritten. Die Repräsentation von Workflows erfolgt entweder mit Hilfe von Diagrammen oder durch eine *Workflowsprache*, die einen Prozess in maschinenlesbarer Form beschreibt. Beispiele für eine solche Sprache sind die *Business Process Execution Language* (BPEL) (Abschnitt 2.4) oder die *XML Process Definition Language* (XPDL) [xpd]. Wir betrachten Workflows, die vollständig durch ein *Workflowmanagementsystem* (WfMS) auf einem Computer ausgeführt werden.

In der Vergangenheit wurden Workflows hauptsächlich zur EDV-gestützten Beschreibung und Ausführung wirtschaftlicher Geschäftsprozesse verwendet (*business workflows*). Seit kurzem finden Workflows auch im wissenschaftlichen Bereich Anwendung (*scientific workflows [TDG07]*), insbesondere auch im Gebiet der Simulationen [G+11]. Die dabei eingesetzten Workflowsysteme werden *scientific workflow management systems*, kurz *sWfMS*, genannt. Die folgenden Teilabschnitte befassen sich mit den Eigenschaften unterschiedlicher Workflows und der Architektur von WfMs.

[1] http://www.w3.org/
[2] https://www.oasis-open.org/

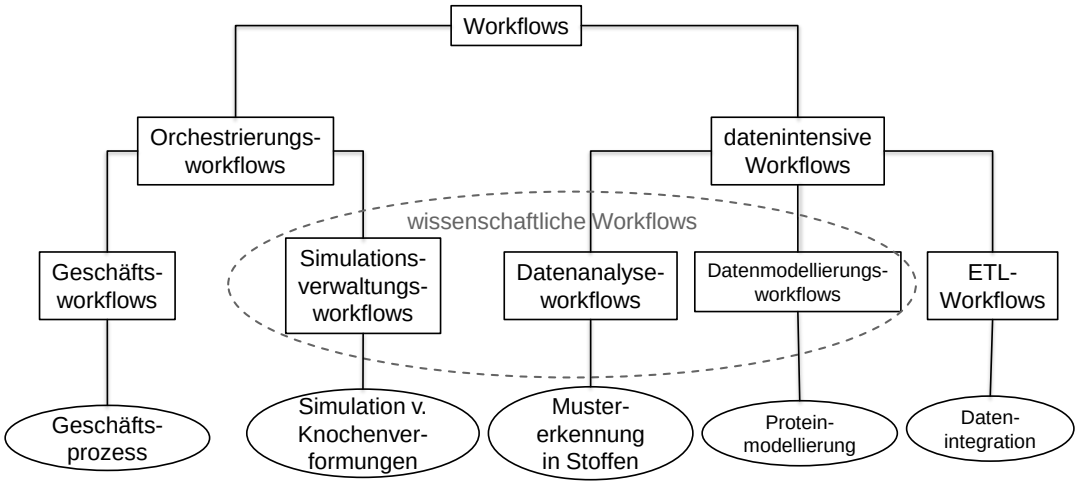

Abbildung 2.1: Einteilung von Workflows nach ihrer Datenorientierung. Unterschiedliche Workflowklassen sind rechteckig dargestellt, deren Anwendungsszenarien sind oval gekennzeichnet.

2.3.1 Workflow-Klassen

Bei wissenschaftlichen Workflows steht oftmals die Verarbeitung größerer Datenmengen im Vordergrund. Im Gegensatz dazu werden *Geschäftsworkflows* (business workflows) hauptsächlich zur Steuerung des Kontrollflusses eines Prozesses eingesetzt. Diese beiden Schwerpunkte erlauben eine Klassifizierung von Workflows in datenfluss- bzw. kontrollflussorientierte Workflows. Abb. 2.1 zeigt eine Einteilung von Workflows anhand der Ausprägung ihrer Datenorientierung gemäß [RSM11].

Orchestrierungsworkflows beinhalten traditionell *Geschäftsworkflows*, bei denen die Koordination (*Orchestrierung*) verschiedener Geschäftsprozesse zu einem Gesamtprozess im Vordergrund steht. Zu diesem Zweck werden dabei hauptsächlich die Konzepte der SOA, insbesondere das der Webservices eingesetzt. Die neuere Klasse der *Simulationsverwaltungsworkflows* koordiniert auf ähnliche Weise einzelne Simulationsprogramme wissenschaftlicher Anwendungunen. Dabei rückt allerdings die Datenverabeitung der einzelnen Programme

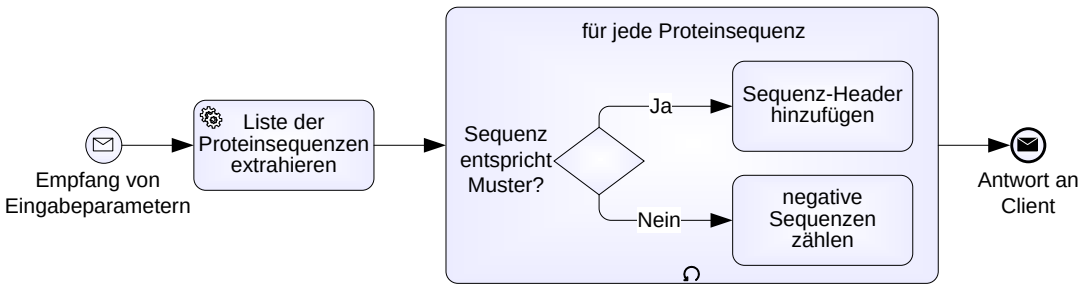

Abbildung 2.2: Workflow zur Proteinmodellierung nach [RSM11] in Business Process Modeling Notation (BPMN).

(Teilprozesse) mehr in den Vordergrund, was auch Auswirkungen auf die Datenverarbeitung innerhalb des Workflows mit sich bringt. Beispielsweise werden bei der Simulation von Knochenverformungen in einem Teilprozess die Daten für das Modell eines Knochens erzeugt, die dann dem eigentlichen Simulationsprozess zur Verfügung gestellt werden müssen.

Datenintensive Workflows stellen im Gegensatz zu den Orchestrierungsworkflows die Datenverarbeitung in den Vordergrund. Dabei werden große Datenmengen von unterschiedlichen Quellen durch den Workflow selbst verarbeitet, anstatt dies durch externe Programme zu veranlassen. *Datenanalyseworkflows* haben das Ziel, zuvor gesammelte Daten zu untersuchen und in einer bestimmten Form darzustellen, um so neue Erkenntnisse zu gewinnen. Hierbei spielt Mustererkennung eine wesentliche Rolle, z.B. bei der Untersuchung von Eigenschaften chemischer Verbindungen.

Die Klasse der *Datenmodellierungsworkflows* dient unter anderem der Extraktion von Modellen aus gegeben Problemstellungen, die anschließend für Berechnungen herangezogen werden können. Sie werden beispielsweise auch bei der Proteinmodellierung eingesetzt, wobei unterschiedliche Modelle erzeugt und auf ihre Eigenschaften hin untersucht werden. Der in Abb. 2.2 dargestellte Workflow zeigt das Beispiel eines solchen Proteinmodellierungsworkflows. Dabei wird zunächst eine Liste mit Proteinsequenzen geladen. Einzelne Proteinsequenzen werden anschließend auf Ähnlichkeit mit einem Muster hin überprüft. Bei positivem Ergebnis wird der Header der Proteinsequenz zu den bisher gefundenen hinzugefügt, bei negativem Ergebnis wird nur ein entsprechender Zähler erhöht.

ETL-Workflows (*extract, transform, load*) werden eingesetzt, um die Integration großer Datenmengen zu realisieren, wobei diese in einer bestimmten Form zusammengestellt werden. Der Workflow selbst ist in diesem Fall lediglich ein Werkzeug für die Datentransformation, das von übergeordneten Anwendungen dazu aufgerufen wird.

Der Begriff der *„wissenschaftliche Workflows“* (scientific workflows) umfasst eine Teilmenge dieser Workflow-Klassen, nämlich die *Simulationsverwaltungsworkflows*, die *Datenanalyseworkflows* und die *Datenmodellierungsworkflows*. Diese Arbeit befasst sich verstärkt mit eben diesen wissenschaftlichen Workflows, wobei der Fokus auf den dantenintensiven Workflows liegt.

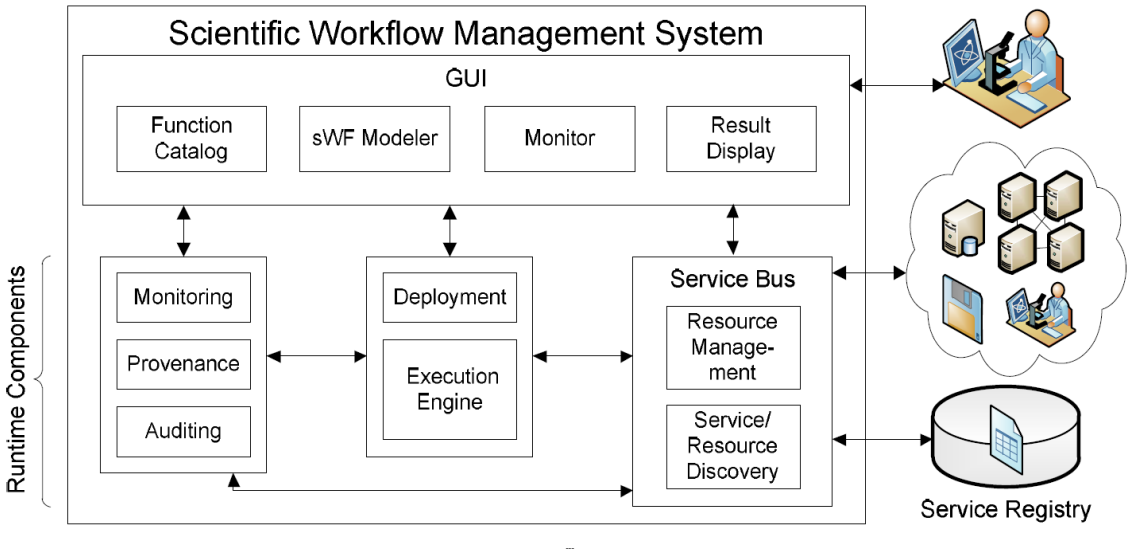

Abbildung 2.3: Architektur eines sWfMS nach [G+11]

Im nächsten Abschnitt wird das Architekturmodell eines Managementsystems für solche wissenschaftliche Workflows vorgestellt.

2.3.2 Architektur eines sWfMS

Um Workflows zu modellieren und auszuführen, wurden Architekturmodelle entwickelt, dass den Anprüchen unterschiedlicher Workflowarten entspricht. Abbildung 2.3 veranschaulicht die Architektur eines sWfMS basierend auf der in [LR00] definierten Technologie für Geschäfts- und Produktionsworkflows. Es folgt eine Beschreibung der einzelnen Komponenten gemäß [RRS+11] und [G+11].

Zuerst werden die Komponenten der **GUI** betrachtet:

- Der **sWf Modeler** unterstützt den Entwickler beim Modellieren der Spezifikationen und Deploymentinformationen eines Workflows.
- Der **Function Catalog** stellt eine Liste von im Workflow verfügbaren Diensten bereit, die bei der Modellierung eingesetzt werden können.
- Die **Monitor**-Komponente bietet eine Benutzerschnittstelle, die zur Überwachung der Ausführung von Workflow-Instanzen und damit zum Erkennen von unerwarteten Ereignissen bzw. Fehlern während der Ausführung dient.
- Das **Result-Display** liefert die Zwischen- und Endergebnisse des ausgeführten Workflows (z.B. Simulationsergebnisse) in einem für den Benutzer bedarfsgerechten Format.

Als nächstes werden die Laufzeitkomponenten (Runtime Components) erläutert:

- Die **Deployment**-Komponente überführt das Modell eines Workflows in ein Objekt und installiert dieses in einer **Ausführungsengine** (z.B Apache ODE[3]), die später Instanzen davon ausführt.
- Die **Auditing**-Komponente speichert workflowbezogene Ereignisse und Aktivitäten, die zur Laufzeit auftreten, z.B. den Anfangszeitpunkt eines Workflowaufrufs.
- Die **Monitoring**-Komponente überwacht die Zustände von Workflowausführungen mit Hilfe der Daten aus der Auditing-Komponente.
- Die **Provenance**-Komponente erfasst weitere, detaillierte Daten einer Ausführung als die Auditing-Komponente. Mit Hilfe dieser Informationen lassen sich Ausführungen exakt nachvollziehen.
- Der **Service Bus** ist für das Finden und Auswählen von Diensten zur Implementierung des Workflows zuständig. Desweiteren dient er der Zustellung von Nachrichten sowie der Durchführung von Datentransformationen innerhalb des sWfMs und kann externe Ressourcen (z.B. Datenquellen) an den Workflow anbinden.
- Die **Resource Management**-Komponente speichert Meta-Informationen über die externen Ressourcen und Dienste.
- Die **Service/Resource Discovery**-Komponente erstellt anhand dieser Meta-Informationen oder mit Hilfe externer Verzeichnisse eine Liste der in Frage kommenden Dienste und Ressourcen, die vorher beispielsweise durch *semantische Annotation* beschrieben wurden (vgl. lose Kopplung, Abschnitt 2.2.1). Mit Hilfe dieser Liste ist während der Ausführung beipielsweise im Fehlerfall die Anbindung eines alternativen Dienstes oder einer alternativen Ressource möglich.

2.4 Web Services Business Process Execution Language

Die **W**eb **S**ervices **B**usiness **P**rocess **E**xecution **L**anguage [bpe], kurz: WS-BPEL, ist eine XML-basierte Workflowsprache, die Vorgänge innerhalb von Geschäftsprozessen mit Hilfe von Webservices beschreibt. Sie gilt als de-facto Standard zur Implementierung von Geschäftsprozessen und wird in dieser Arbeit verwendet. Mit ihrer Hilfe können Workflows beschrieben, bearbeitet und ausgetauscht werden. Sie ermöglicht eine integrierte und selbstständige Ausführung des Workflows durch ein WfMS, sofern alle Teilprozesse und die Bereitstellung der dazu benötigten Daten ebenfalls automatisch durchgeführt werden können.

Bei der Beschreibung von Geschäftsprozessen mit Hilfe von BPEL werden einzelen Teilschritte im Allgemeinen durch Webservices implementiert. Ausnahmen hiervon bilden Erweiterungen wie z.B. *BPEL4People* [KKL+05], bei der menschliche Interaktion in einen Workflow eingebettet werden kann. Die wesentlichen Vorteile von BPEL bei der Beschreibung von wissenschaftlichen Workflows sind der modulare Aufbau, die Flexibilität im Umgang

[3] http://ode.apache.org/

mit genereischen XML-Datentypen und der späten Anbindung von Diensten an den Workflow, sowie umfangreiche Möglichkeiten zur Erweiterung sowie Fehlerbehandlung und -kompensation [AMAo6].

Grundlage der Ausführung von Workflows mit BPEL sind *Aktivitäten*. Eine Aktivität repräsentiert einen Ausführungsschritt innerhalb eines Workflows. Es gibt drei grundlegende Arten von BPEL-Aktivitäten:

Aktionen

Aktionen führen einen bestimmten Arbeitsschritt innerhlab eines Workflows aus. Dazu gehören unter anderem die folgenden Aktivitäten:

- **RECEIVE** wartet auf den Empfang einer bestimmten SOAP-Nachricht, die von einem aufrufenden Prozess als Anfrage versendet wird, und setzt daraufhin die Workflowausführung an der modellierten Stelle fort. Ein BPEL-Workflow beginnt stets mit einer *RECEIVE*-Aktivität, die eine Instanz des Workflows erzeugt (Attribut *createInstance=yes*).
- **REPLY** erzeugt und versendet eine Ausgangsnachricht als Rückantwort an den aufrufenden Prozess. Eine *REPLY*-Aktivität lässt sich damit immer einer empfangenden (*RECEIVE*)-Aktiviät zuordnen.
- **PICK** wartet, ähnlich wie Receive, auf den Eingang einer Nachricht (*onMessage*) oder auf das Eintreten einer zeitlichen Bedingung (*onAlarm*). Hierbei können nun mehrere Nachrichten angegeben werden, die jeweils eine unterschiedliche Folgeaktivität auslösen können. Es wird immer diejenige Folgeaktivität ausgelöst, deren Ereignis als erstes eingetreten ist.
- **INVOKE** ruft eine bestimtme Operation eines anderen Webservice auf. Der Workflow, der die *INVOKE*-Aktivität ausführt, übernimmt dabei die Rolle des aufrufenden Prozesses. Die Aus- bzw. Eingangsnachrichten werden dabei aus bzw. in Workflowvariablen zugewiesen.
- **ASSIGN**: Durch *ASSIGN*-Aktivitäten können workflowinterne Variablen manipuliert werden. Innerhalb einer *ASSIGN*-Aktivität können Zuweisungen durch mehrere *COPY-Blöcke* stattfinden, wobei einer Variablen unter anderem eine andere Variable, ein Literalwert oder ein Ausdruck (z.B. XPath oder XQuery) zugewiesen werden kann. Die unterschiedlichen Zuweisungsarten von *ASSIGN*-Aktivitäten werden im weiteren Verlauf dieser Arbeit näher untersucht.

Kontrollstrukturen

Kontrollstrukturen dienen der bedingten oder wiederholten Ausführung bestimmter Teile eines Workflows sowie der Organisation von Aktivitäten.

- **IF**: Überprüft eine odere mehrere Bedingungen (*„expression evaluation“*) und führt abhängig vom Ergebnis unterschiedliche Folgeaktivitäten aus.
- **WHILE**: Innerhalb einer *WHILE*-Aktivität geschachtelte Aktivitäten werden wiederholt ausgeführt, so lange die damit verknüpften Bedingungen zutreffen.
- **SCOPE**: Durch eine *SCOPE*-Aktivität können mehrere Aktivitäten zu einem semantischen Verbund zusammengefasst werden. Ein Scope stellt einen Gultigkeits- und Sichtbarkeitsbereich für Variablen dar und erlaubt die Definition von Fehlerbehandlung und -kompensation für die enthaltenen Aktivitäten. Im Fehlerfall können alle Aktivitäten innerhalb eines Scope wieder rückgängig gemacht werden, ohne dass Aktivitäten außerhalb des Scope davon betroffen sind.

Fehlerbehandlung

Fehlerbehandlung kann in WS-BPEL durch das Werfen von Ausnahmen (***THROW***-Aktivität) kontrolliert werden. Daraufhin kann etwa die Ausführung eines Prozesses abgebrochen und eine Fehlermeldung ausgegeben werden (***EXIT***-Aktivität). Fehlerbehandlung wird im Kontekt dieser Arbeit nicht näher verwendet und daher an dieser Stelle nicht weiter nehandelt.

Detailiertere Informationen zu diesem oder anderen BPEL-Themen sind im OASIS-Standard [bpe] nachzulesen.

2.5 Datenbanksysteme

Für den Zugriff auf Daten und deren Speicherung wurden im Verlauf der letzten Jahrzehnte viele Technologien und Produkte entwickelt, die diese Aufgaben mit besonders hoher Effizienz erfüllen. Datenbanksysteme (DBS) sind heutzutage ein wichtiges Werkzeug zur Datenverarbeitung und werden von den meisten Unternehmen eingesetzt. Auch im Internet spielen Datenbanken eine große Rolle. So kommt heutzutage kaum eine Webseite ohne eine Datenbank aus, das im Hintergrund den auf den Seiten dargestellten Inhalt verwaltet.

Ein Datenbanksystem setzt sich aus der zugrundeliegenden Datenbank sowie einem Datenbankmanagementsystem (DBMS) zusammen. Ein DBMS ermöglicht es dem Anwender seine Daten sicher und komfortabel zu verwalten. So können beispielsweise unterschiedlichen Benutzern verschiedene Zugriffs- und Verwaltungsrechte auf bestimmte Daten gewährt oder verweigert werden. Durch die Festlegung von Integritätsbedingungen kann gewährleistet werden, dass keine fehlerhaften Eingaben gemacht werden können und sich die Datenbank somit zu jedem Zeitpunkt in einem konsistenten Zustand befindet. Für weitere Vorteile und Einsatzmöglichkeiten von Datenbanksystemen sei auf die entsprechende Literatur zum Thema Datenbanken verwiesen [KE06].

Am weitesten verbreitet sind sogenannte *relationale Datenbanksysteme*, bei denen Daten als Zeilen einer Tabelle aufgefasst werden. Die auf diese Weise gespeicherten Daten werden durch entsprechende Operatoren *mengenorientiert* verknüpft und verarbeitet, d.h. Resultate werden letztlich durch Selektion, Vereinigung oder dem Schnitt verschiedener Datensätze erzeugt. Die entsprechenden Operationen werden mit Hilfe der Abfragesprache *SQL* (*Structured Query Language*) [sql] formuliert und mit Hilfe eines DBMS in der Datenbank ausgeführt.

Durch die anhaltende Verbreitung von XML enstand die Anforderung an DBMS, dieses Datenformat innerhalb von Datenbanken zu unterstützen. Dabei enstanden unter anderem sogenannte *native* XML-Datenbanken, die ausschließlich XML-Dokumente mit Hilfe der dafür entwickelten Abfragesprachen speichern und verarbeiten. Etablierte relationale Datenbanken führten jedoch einen neuen XML-Spaltentyp ein und werden als *XML-Enabled* bezeichnet. Dabei werden alle Eigenschaften einer SQL-basierten relationalen Datenbank beibehalten, während die XML-Spalten zusätzlich durch XML-Abfragesprachen verarbeitet werden können.

Bei dem in dieser Arbeit verwendeten Datenbanksystem handelt es sich um das Produkt DB2 der Firma IBM (*International Business Machines*), das in den 1980er Jahren eingeführt und seitdem stetig weiterentwickelt wurde. DB2 ist eine relationale Datenbank und besitzt seit Version 9 einen XML-Spaltentyp. Es gehört damit zu den Vertretern der *XML-Enabled* Datenbanken. Die in DB2 umgesetzten XML-Technologien und Abfragesprachen werden unter dem Begriff *pureXML* zusammengefasst. PureXML unterstützt die XQuery- und somit auch die XPath-Spezifikation. Es ist möglich XQuery-Anfragen in SQL-Ausdrücke einzubetten und relationale Daten in XQuery-Anfragen einzubinden. Zur Einbettung von XQuery in SQL wird die SQL-Funktion *XMLQUERY* verwendet. Für die Einbettung von SQL oder den Zugriff auf ein relationales XML-Feld innerhalb eines XQuery-Ausdrucks kommen die XQuery-Erweiterungsfuntionen *db2-fn:sqlquery* bzw. *db2-fn:xmlcolumn* zum Einsatz. Darüber hinaus besitzt pureXML eine eigene, in XQuery eingebettete Syntax zur Manipulation von XML-Daten.

Es fanden bereits durch vorangegangenen Arbeiten Evaluationen mit Hilfe von DB2 statt. In dieser Arbeit wird ebenfalls DB2, in der Version 10.1, zur Evaluation eingesetzt.

3 Nutzung von Datenbanken in WfMS

In diesem Kapitel soll das Zusammenspiel von Datenbanken und Workflowmanagementsysteme untersucht werden. Zunächst wird erläutert wie verschiedene, konventionelle WfMS Datenbanksysteme bisher einsetzen. Danach werden Konzepte, die eine stärkere Integration von DBS bei der Workflowausführung thematisieren, sowie deren momentaner Entwicklungsstand vorgestellt. Schließlich werden die bereits entwickelten Konzepte weiter untersucht, um fehlende bzw. unausgereifte Teilaspekte zu beleuchten und mögliche Lösungs- und Verbesserungswege aufzuzeigen.

Wie von vielen anderen Softwarearchitekturen werden Datenbanken auch von WfMS eingesetzt. Sie werden verwendet, um Daten über bekannte Prozesse und laufende Workflowinstanzen (z.B. Variableninhalte) zu speichern und persistent zu halten. Dadurch können laufende Instanzen nach einem Systemausfall wiederhergestellt oder im Fehlerfall über Monitoring-Tools analysiert werden. Die meisten WfMS benutzen dazu eine integrierte Datenbank. Die Nutzung dieser Datenbank für Aufgaben während der Workflowausführung wird von konventionellen Systemen bisher nicht verfolgt.

3.1 Ansätze zur Datenverarbeitung

Es sind bereits in der Vergangenheit Ansätze entwickelt worden, um die Leistungsfähigkeit von Datenbanksystemen bei der Ausführung von Workflows ausnutzen zu können. Dazu ist konzeptionell zunächst zu unterscheiden, ob eine anfallende Datenoperation innerhalb oder außerhalb des Workflows definiert ist und ob sie innerhalb oder außerhalb des Workflows ausgeführt wird (Vgl. Tabelle 3.1).

Im Fall der Ausführung außerhalb des Workflows werden Datenoperationen durch das Aufrufen eines externen Prozesses realisiert, der dann die entsprechende Datenoperation ausführt. Übernimmt der externe Prozess dabei auch die Definition der Datenoperation selbst, d.h. also sowohl Definition als auch Ausführung der Datenoperation finden vollständig außerhalb des Workflows statt, so nennt man diese externen Prozesse *Data Management-* bzw. *DM-Dienste* (engl.: *data services*). Der Zugriff auf die Datenquelle wird damit durch den externen Prozess vollständig vom Workflow gekapselt und damit getrennt. Die meisten aktuellen WfMS bieten Möglichkeiten für die Einbindung solcher externen Prozesse innerhalb eines Workflows an, z.B. in Form von Webservice-Aufrufen.

Finden sowohl Definition als auch Ausführung von Datenoperationen innerhalb des Workflows statt, so spricht man von *lokaler Datenverarbeitung* (engl.: *local data processing*). Dabei werden alle verwendeten Daten innerhalb des Workflows erfasst und verarbeitet. Diese

Ausführung \ Definition	innerhalb des Workflows	außerhalb des Workflows
innerhalb des Workflows	lokale Datenverarbeitung	-
außerhalb des Workflows	DM-Aktivitäten	DM-Dienste

Tabelle 3.1: Konzeptionelle Unterscheidung von Datenoperationen in Workflows. (DIe Abkürzung *DM* steht hierbei für *data management*) (Vgl. [RSM])

Daten müssen einer Workflowinstanz zunächst bereitgestellt werden, dies geschieht z.B. mit Hilfe von Eingangsnachrichten. Dieses Konzept bildet den Schwerpunkt dieser Arbeit, wobei die Datenverarbeitung innerhalb des Workflows zusätzlich mit Hilfe einer integrierten Datenbank erfolgt.

Ein dritter Fall liegt dann vor, wenn Datenoperationen von externen Prozessen ausgeführt, jedoch innerhalb des Workflows definiert werden. Man spricht dabei von *Data Management-* bzw. *DM-Aktivitäten* (engl.: *data management activities*). Hierbei wird nicht die gesamte Datenoperation durch den externen Prozess gekapselt, stattdessen wird der Zugriff auf eine externe Datenquelle durch besondere Aktivitäten modelliert. Ein Beispiel für den Einsatz dieses Konzepts ist eine Erweiterung von BPEL-Prozesen um die Möglichkeit SQL-Befehle an ein gekapseltes DBS abzusetzen [VSRM08]. Eine Abstraktion dieser Funktionalität bietet das *SIMPL*-Rahmenwerk, das als reine BPEL-Erweiterung unabhängig von WfMS entwickelt wird und Zugriff auf bestimmte, unterschiedliche Datenquellen erlauben soll [RRS+11]. Abbildung 3.1 veranschaulicht die drei beschriebenen, unterschiedlichen Vorgänge der Ausprägungen von Datenoperationen.

3.2 Ansätze zur Verbesserung der Integration von Datenbanken

Um die Möglichkeiten einer engeren Integration zwischen Datenbank und WfMS untersuchen zu können, wird zunächst das bestehende Funktionsprinzip bei der Datenverarbeitung innerhalb eines Workflows beleuchtet. Anschließend wird ein Konzept vorgestellt, das auf dieser Grundlage die integrierte Datenbank und deren Funktionen während der Workflowausführung stärker ausnutzt.

3.2.1 Konventionelle Funktionsweise von WfMS

Abb. 3.2(a) zeigt die für die interne Datenverarbeitung wesentlichen Teile des Architekturmodells, das der Ausführung kontrollflussorientierter Workflows zugrundeliegt. Bei der abgebildeten Datenbank handelt es sich wohlgemerkt nicht um eine extern angebundene Datenbank, wie sie von *DM-Aktivitäten* oder *DM-Diensten* (vgl. Abschnitt 3.1) verwendet wird. Es handelt sich um die interne Datenbank des WfMS, die in erster Linie für die prozessbegleitenden Daten sowie für die Persistenz der Workflowdaten verwendet wird. Diese persistenten Daten werden vom WfMS jedoch während einer Workflowausführung

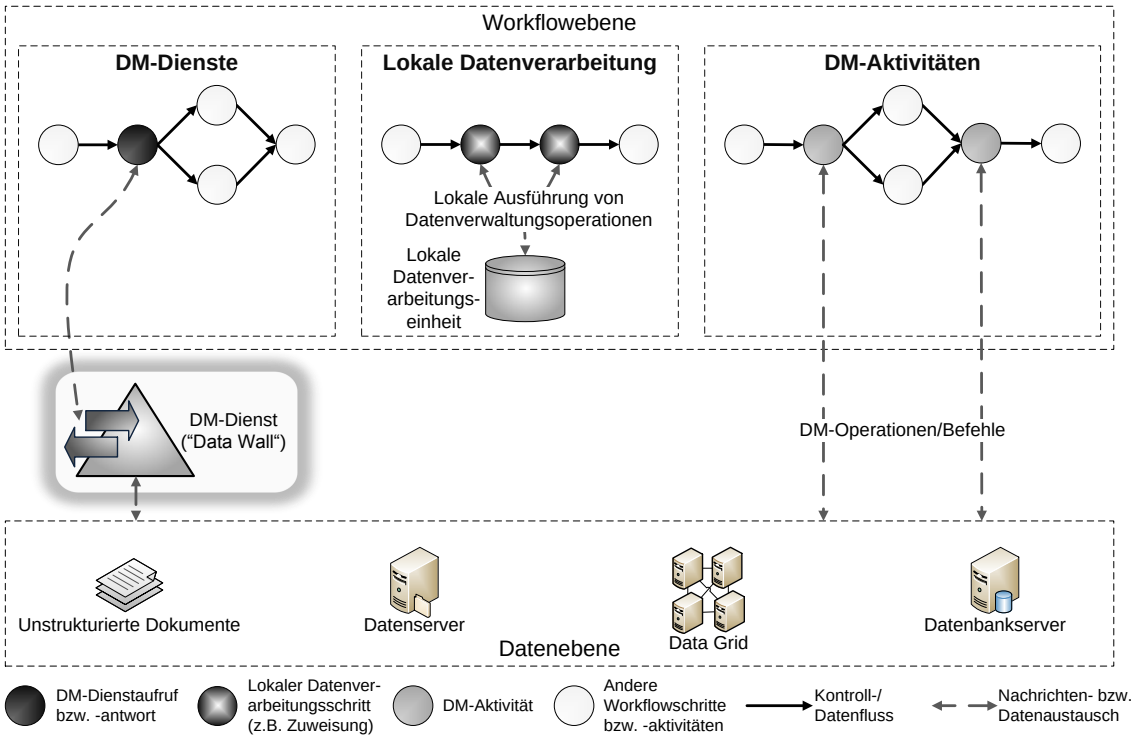

Abbildung 3.1: Darstellung der Vorgänge bei den verschiedenen Ausprägungen von Datenoperationen in Workflows (Vgl. [RSM])

nicht verwendet (z.B. für Variablenzuweisungen), sodern nur in Ausnahmefällen zur Korrektur oder Diagnose herangezogen. Stattdessen werden alle Datenoperationen von der Workflowengine auf einem ihr zugrundeliegenden, internen Datenmodell durchgeführt. Bei diesem Datenmodell können die Variablen beispielsweise durch Java-Objekte auf der Halde (*heap*) des Arbeitsspeichers der Workflowengine realisiert sein, die von dort z.B. durch die *Ausdruckauswertungsengine* (*expression evaluation engine*) ausgewertet werden. Die Komponente *Datenverarbeitungslogik* (*data processing logic*) bestimmt dabei, wie die Bestandteile des WfMS (z.B. einzelne BPEL-Aktivitäten) auf diese Variablen zugreifen und diese verarbeiten können.

Ein *Persistenzmanager* (*persistence manager*) speichert Inhalte aus dem Datenmodell des Workflows in das eingebettete Datenbanksystem und sorgt so für deren Dauerhaftigkeit. Welches Datenbanksystem dabei verwendet wird spielt dabei grundsätzlich keine Rolle. Um ein Abbild von Inhalten eines Workflows zu gewährleisten, muss die Datenbank, ebenso wie die *Laufzeitkomponente* (*runtime*) des WfMS, einen eigenen *Variablenvorrat* (*variable pool*) verwalten. Es ist Aufgabe des Persistenzmanagers, den Variablenvorrat in der Datenbank mit dem aktuellen Inhalt der Variablen in der Laufzeitkomponente zu synchronisieren. Dies geschieht entweder selbsttätig oder auf Anfrage durch die Laufzeitkomponente. Der Persistenzmanager verwendet dabei die *Anfrage-/Ausdruckausführungsengine* (*query/expression execution engine*) des verwendeten DBMS, um die entsprechenden Zuweisungen der Inhalte durchzuführen.

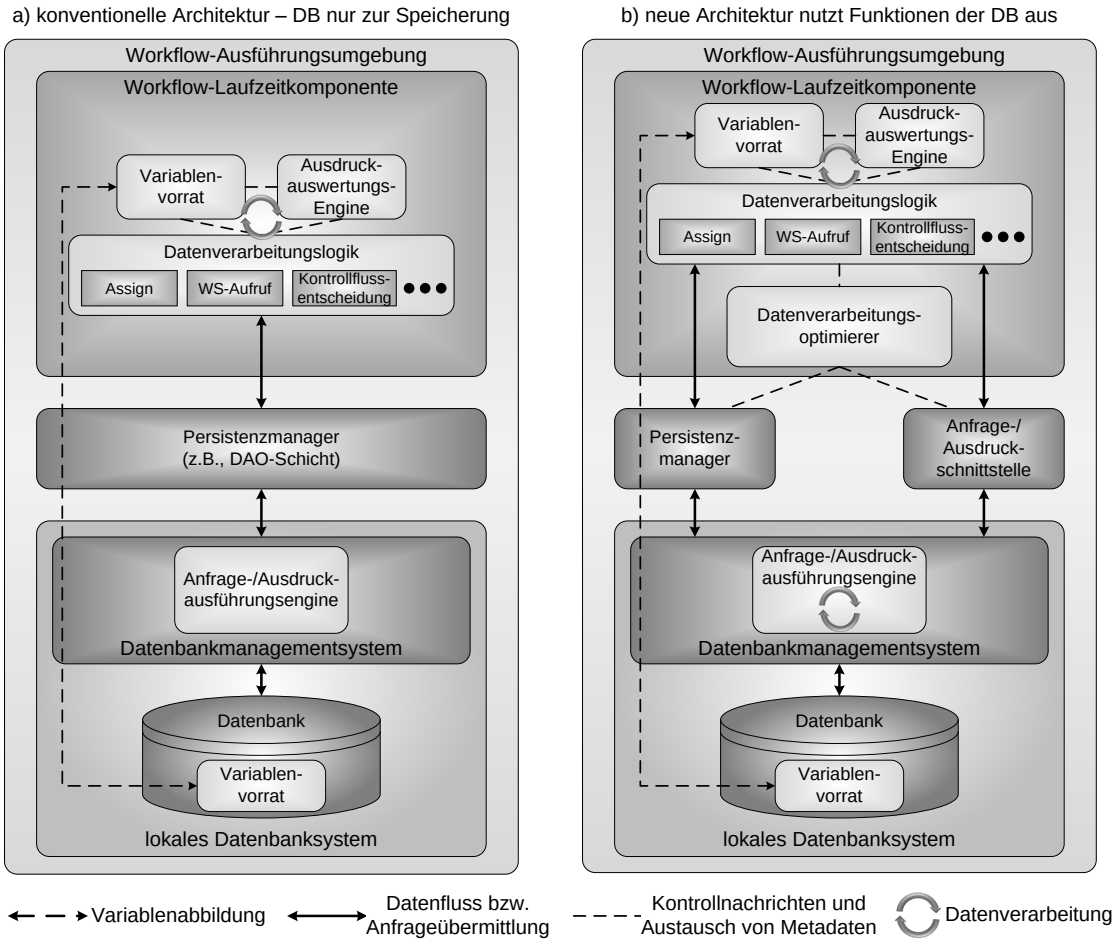

Abbildung 3.2: Architektur kontrollflussorientierter Workflowausführung (a) sowie vorgeschlagene Änderungen daran (b) (vgl. [RSM11]).

3.2.2 Konzept für eine stärkere Integration von DBS

Um Workflowdaten innerhalb datenintensiver Workflows effizienter und zuverlässiger verwalten zu können, wurde das Architekturmodell aus Abb. 3.2(b) entwickelt, bei dem einige Teile der Architektur verändert und einige hinzugefügt wurden. Das Ziel war dabei eine Aufteilung der Datenverwaltungsaufgaben zwischen der Laufzeitkomponente und dem Datenbanksystem, sodass die Stärken des DBS während der Workflowausführung bestmöglich ausgenutzt werden. Dabei sollten keinerlei Auswirkungen auf die Modellierung der Workflows entstehen. Das bedeutet, dass alle Änderungen für den Anwender transparent sein müssen, damit die Bedienung durch die Integration nicht beeinflusst wird.

Neben dem Persistenzmanager wurde eine *Anfrage-/Ausdruckschnittstelle* (*query/expression interface*) eingeführt, mit deren Hilfe sich zusätzliche Operationen auf dem DBS ausführen lassen. Hiermit sollen keine Datenmengen zwischen der Laufzeitkomponente und dem DBS

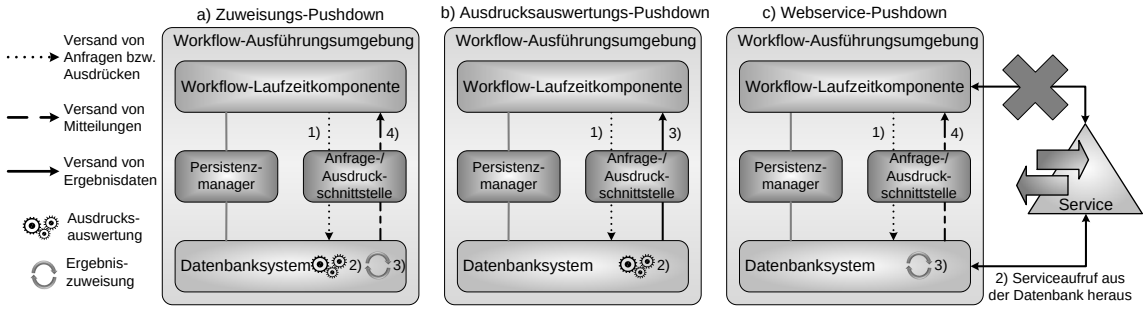

Abbildung 3.3: Pushdown-Arten nach [RSM11]. Die Zahlen deuten die Reihenfolge an, in der einzelne Teilschritte ausgeführt werden.

ausgetauscht werden, es soll lediglich eine Schnittstelle sein, um Ausdrücke bzw. Anfragen an die Datenbank zu senden, die höchstens durch kurze Statusnachrichten quittiert werden. Damit soll beispielsweise der Kontrollfluss innerhalb eines Workflows abhängig von Datenbankinhalten gesteuert werden können. Da die im Rahmen der Integration von WfMS und DBS entstehenden Datenbankanfragen nicht immer so trivial sind, wie es etwa bei der Abbildung von Variablen durch den Persistenzmanager der Fall ist, müssen bei dieser Schnittstelle für unterschiedliche DBS gegebenenfalls verschiedene Implementierungen bestimmter Anfragen erstellt werden. Die Laufzeitkomponente bleibt damit weiterhin unabhängig vom eingesetzten DBS.

Der ebenfalls neu eingeführte *Datenverarbeitungsoptimierer* (*data processing optimizer*) entscheidet, ob eine Datenoperation in der Laufzeitkomponente ausgeführt oder an das DBS delegiert werden soll und wo das Ergebnis der Operation gespeichert werden soll. Das Ziel dieser Entscheidungen ist es wiederum, die Stärken des DBS bei der Datenverarbeitung und -abfrage bestmöglich auszunutzen und dabei gleichzeitig den Datenverkehr zwischen dem DBS und der Laufzeitkomponente so gering wie möglich zu halten. Welche Operationen an das DBS delegiert und welche intern verarbeitet werden, hängt unter anderem von der Datenmenge und der Komplexität der Operation ab.

3.2.3 Techniken zur Verbesserung der Datenverarbeitung

Der im vorigen Abschnitt beschriebene Datenverarbeitungsoptimierer kann verschiedene Änderungen bei der Ausführung workflowinterner Datenoperationen vornehmen. Die Vorgänge, bei denen bestimmte Operationen auf das DBS „heruntergedrückt“ werden, werden als *Pushdowns* bezeichnet. Einige Pushdown-Konzepte sind in Abb. 3.3 dargestellt.

Beim **Zuweisungs-Pushdown** (*assignment pushdown*) werden Variablenzuweisungen aus der Laufzeitkomponente in die Datenbank ausgelagert. Die Datenbank empfängt dabei zunächst einen Zuweisungsausdruck (1), den sie auswertet (2). Danach wird die entsprechende Zuweisung innerhalb des Variablenvorrats der Datenbank durchgeführt (3) und anschließend wird der Laufzeitkomponente eine Statusnachricht über den Erfolg bzw. Fehlerfall des Ausdrucks

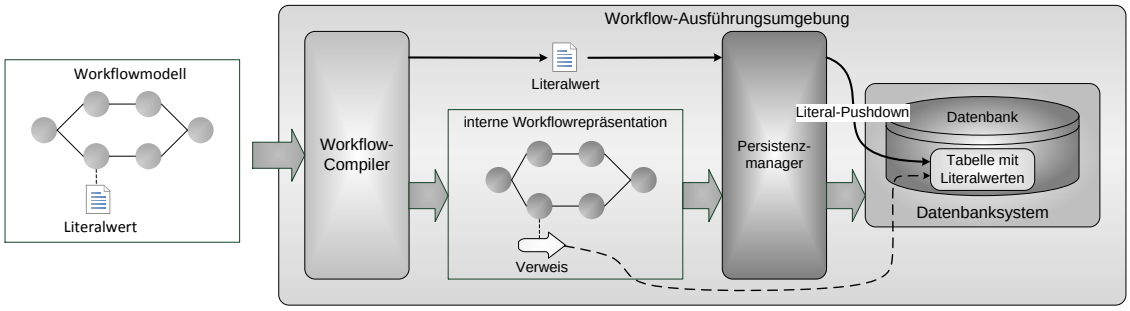

Abbildung 3.4: Literal-Pushdown nach [RSM11]

zurückgegeben (4). Die entsprechende Datenoperation kann vom WfMS asynchron ausgelöst werden, d.h. während das DBS die Operation durchführt, kann die Workflowausführung fortgesetzt werden.

Der **Ausdruckauswertungs-Pushdown** (expression evaluation pushdown) dient zur Abfrage von Variableninhalten für Entscheidungen über den Kontrollfluss innerhalb der Workflowausführung. Hierbei werden auch Ausdrücke an das DBS gesendet und dort ausgewertet. Die Antwortnachricht des DBMS wird allerdings diesmal von der Laufzeitkomponente des WfMS benötigt, wodurch die Workflowausführung bis zu ihrem Eintreffen unterbrochen werden muss (synchrone Ausführung).

Mit Hilfe dieser beiden Pushdown-Konzepte lässt sich prinzipiell schon eine Geschwindigkeitssteigerung erreichen, da das DBS DM-Operationen effizient und teilweise asynchron übernehmen kann, ohne dass viel Datenverkehr zwischen der Laufzeitkomponente des WfMS und dem DBS entsteht. Allerdings setzen diese beiden Konzepte voraus, dass alle für die Auswertung eines Ausdrucks bzw. für eine Zuweisung benötigten Daten bereits in der Datenbank vorliegen. Ist dies nicht der Fall, so müssen diese Daten zunächst, etwa mit Hilfe des Persistenzmanagers, an die Datenbank übermittelt werden. Dies wirkt sich wiederum negativ auf die Ausführungsgeschwindigkeit aus. Um diese negativen Auswirkungen zu vermeiden wurden zwei weitere Pushdown-Arten entwickelt, die sicherstellen sollen, dass die benötigten Daten zum richtigen Zeitpunkt in der Datenbank vorliegen.

Es gibt in BPEL zunächst nur zwei Arten, wie ein Workflow Daten von außerhalb des WfMS erhalten kann: Durch SOAP-Nachrichten aus der Kommunikationsinfrastruktur als Ergebnis eines Webservice-Aufrufs und durch Literalwerte, die innerhalb des Workflowmodells definiert und anschließend einer Variablen zugewiesen wurden. Um das Problem der Antwortnachrichten bei einem Webservice-Aufruf durch den Workflow zu lösen, wurde der *Webservice-Pushdown* konzipiert. Hierbei wird ein vom Workflow aufzurufender Webservice direkt durch das DBS aufgerufen und das Ergebnis des Aufrufs in der Datenbank abgelegt. Dazu muss das eingesetzte DBS allerdings über einen entsprechenden Funktionsumfang zum selbständigen Aufruf von Webservices verfügen.

Literale sind Teile des Workflowmodells und werden als solche üblicherweise nicht durch den Persistenzmanager auf die Datenbank abgebildet. Beim Konzept des Literal-Pushdown wird die Tatsache ausgenutzt, dass Literalwerte bereits zur Modellierungszeit definiert werden. Sie können also bereits vor ihrer Verwendung während der Ausführung (z.B. beim Deployment innerhalb des Workflow-Compilers) in der Datenbank abgespeichert werden. Dort gespeicherte Literalwerte werden zur Laufzeit durch einen im Modell hinterlassenen *Verweis* (Referenz) auf den Datenbankeintrag adressiert (Vgl. Abb. 3.4).

Durch diese zwei Konzepte soll sichergestellt werden, dass beim Auftreten einer mit Pushdown umgesetzten Datenoperation alle für sie benötigten Daten in der Datebank vorliegen. Der Geschwindigkeitsverlust durch das Abspeichern der Literalwerte in der Datenbank kann auf weniger kritische Phasen eines Workflows verschoben werden, während Webservice-Aufrufe eigenständig vom DBS durchgeführt werden. Letzteres stellt einen Schritt in Richtung eines DBS als WfMS erster Klasse dar, was ein weiteres Forschungsthema ist [AIL98] [AIL98], auf das in dieser Arbeit jedoch nicht weiter eingegangen wird.

3.3 Prototypische Erweiterungen eines WfMS

Einige Teile der in den vorigen Abschnitten vorgestellten Techniken wurden bereits prototypisch umgesetzt [Wag11]. Das dabei verwendete, java-basierte WfMS *Apache ODE*, sowie wichtige Aspekte seiner Architektur werden in Kapitel 5 detailiert vorgestellt. Der entstandene Prototyp mit den umgesetzten Pushdown-Arten wurde *ODE-TI* (**T**ight **I**ntegration) genannt und in [RSM11] genauer evaluiert. Die im Folgenden beschriebenen Konzepte, Implementierungen und Auswertungen basieren auf den genannten Quellen.

Beim ODE-TI Prototyp wurden der *Zuweisungs-Pushdown*, der *Ausdruckauswertungs-Pushdown* sowie der *Webservice-Pushdown* implementiert und evaluiert. Dabei wurden anhand von Instanzlaufzeiten und Ausgaben in Logdateien die Laufzeiten verschiedener Workflowausführungen berechnet. Dadurch konnten bestimmte Aussagen über eine Leistungsveränderung durch die Modifikationen des Prototyps bei datenintensiven Workflows getroffen werden.

3.3.1 Prototypen und Aufbau der Zeitmessungen

Für den Vergleich der Pushdown-Konzepte mit den konventionellen Workflowausführungen wurde Apache ODE V1.3.4 in zwei unterschiedlichen Implementierungen verwendet (Abb. 3.5). Dabei wurden jeweils Hibernate[1] V3.2.5 als Persistenzmanager und IBM DB2[2] V9.7 als DBS gewählt. Innerhalb von ODE werden Variablen als XML-Dokumente gespeichert, die während der Ausführung als Java-Ojbekte vom Typ W3CNode[3] im Hauptspeicher liegen. Diese strukturierten Objekte werden vor dem Festschreiben in das DBS durch Hibernate in

[1]Hibernate: http://docs.jboss.org/hibernate/core/3.5/reference/en/html/
[2]IBM DB2: http://www-01.ibm.com/software/data/db2/
[3]W3C Node: http://download.oracle.com/javase/1.4.2/docs/api/org/w3c/dom/Node.html

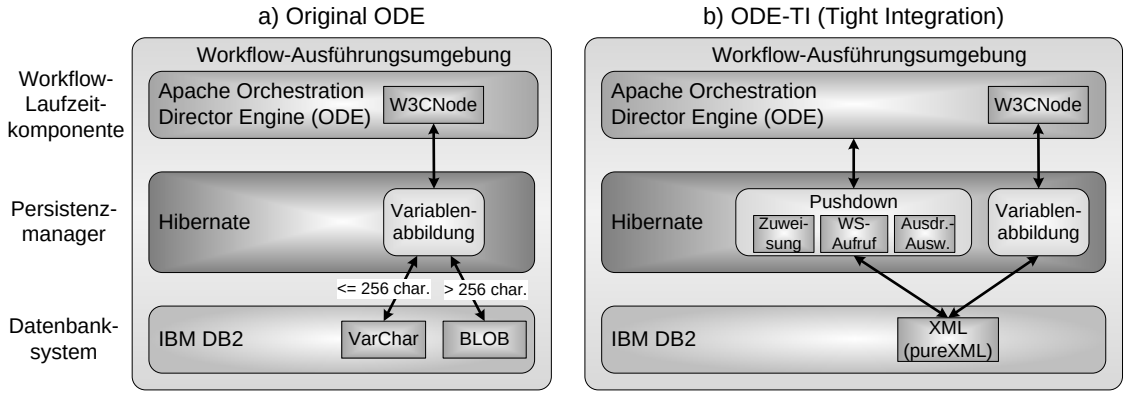

Abbildung 3.5: Architektur von Original ODE (a) und des ODE-TI Prototyps (b) (Vgl. [RSM11])

Text- bzw. Bytecoderepräsentationen umgewandelt. Dabei werden kleinere XML-Dokumente (<=256 Zeichen) in Textform gespeichert, während größere als Binärobjekte (BLOB) komprimiert in der Datenbank abgelegt werden (Abb. 3.5(a)). Diese Architektur ist stellvertretend für die konventionelle Ausführung von Workflows und wird als *Original ODE* bezeichnet. Um die pureXML-Abfragefunktionen von DB2 bei der Workflowausführung für die Pushdown-Konzepte nutzen zu können wurde bei der Architektur in Abbildung 3.5(b) die Datenbankstruktur und die damit verbundene Abbildung durch Hibernate dahingehend geändert, dass XML-Inhalte von Variablen nun ausschließlich in dem datenbanknativen XML-Spaltentyp gespeichert werden.

Es wurden verschiedene Testworkflows erstellt, die unterschiedliche Aspekte einer Workflowausführung betonen. Veränderliche Kriterien sind dabei die Größe der verarbeiteten Daten, die Komplexität der verwendeten Ausdrücke, sowie die Komplexität des gesamten Workflows. Durch Variation dieser Größen wurde versucht Rentabilitätsschwellen zu finden, ab denen die Workflowausführung auf dem ODE-TI-Prototyp gegenüber Original ODE effizienter oder zuverlässiger wird. Diese Renatbilitätsschwellen variieren je nach verwendetem System und Anwendungsszenario. Die Evaluation der Pushdown-Techniken wurde zunächst isoliert anhand eigens dafür erstellter Workflows durchgeführt, welche nur Aktivitäten jeweils einer zu messenden Pushdown-Technik modellieren. Dabei wurden beispielsweise der Zuweisungs-Pushdown mittels einer BPEL Assign-Aktivität, der Ausdruckauswertungs-Pushdown mittels einer If-Aktivität und der Webservice-Pushdown mittels einer Invoke-Aktivität modelliert und ausgewertet. Zusätzlich wurden Messungen unter Verwendung eines Beispielszenarios aus der Proteinmodellierung (vgl. Abb. 2.2) durchgeführt. Dabei wird eine Liste mit Proteinsequenzen mittels eines WS-Aufrufs (invoke) geladen, einzelne Proteine mittels einer If-Aktivität auf Ähnlichkeit mit einem Muster hin überprüft und anschließend mittels Assign-Aktivitäten zu den entsprechenden Ergebniswerten hinzugefügt. Dadurch werden die genannten Pushdown-Techniken nochmals im Zusammenspiel evaluiert.

Die Testdaten bei der Durchführung der isolierten Tests sind die selben Listen von Proteinsequenzen, wie sie vom späteren Beispielszenario verwendet werden. Dabei wird jede Proteinsequenz durch ein XML-Element innerhalb der Liste der Proteinsequenzen repräsentiert. Es wurden Messungen mit Listen der Größen 100 KB, 500 KB, 4 MB, 9 MB sowie 50 MB durchgeführt. Diese entsprechen jeweils 40, 199, 697, 1394 sowie 7695 Proteinsequenzen, was auch der Anzahl der Iterationen der Vergeichsschleife des Proteinmodellierungsworkflows entspricht. Größere XML-Dokumente konnten nicht getestet werden, da Apache ODE dabei entweder Speicherüberläufe erzeugt oder die Workflowausführung ohne Angabe von Fehlern oder Ergebnissen nach 2000 Sekunden abbricht.

Die hier vorgestellten Messungen wurden auf einem Testsystem unter Windows Server 2003 (32-bit) Enterprise Edition SP2 mit einem Intel Xeon PowerEdge 2850 3,2 GHz-Prozessor und 8 GB Hauptspeicher durchgeführt [RSM11].

3.3.2 Testergebnisse

Bei Datengrößen bis zu 9 MB wurde jeder BPEL-Prozess für die isolierte Evaluation 100 Mal ausgeführt, die Prozesse mit 50 MB Datengröße wurden 50 Mal ausgeführt. Aus allen gemessenen Laufzeiten der jeweils verwendeten Aktivität wurden Mittelwerte für diese berechnet. Die Ergebniswerte der mittleren Laufzeiten von ODE-TI sind in Abbildung 3.6 relativ zu den Laufzeiten von Original ODE dargestellt.

Der Zuweisungs-Pushdown wurde unter Verwendung unterschiedlich komplexer XPath-Ausdrücke evaluiert. Der dabei verwendete Ausdruck muss zum Auslesen des Variableninhalts ausgewertet werden, damit anschließend das Ergebnis des Ausdrucks abgespeichert werden kann. Der erste Testfall beinhaltet *keinen Ausdruck*, hier wird einfach der gesamte Variableninhalt kopiert. Der zweite Fall (*einfacher Ausdruck*) beinhaltet einen XPath-Ausdruck, der genau eine Proteinsequenz aus der Liste selektiert. Der dritte Fall (*komplexer Ausdruck*) selektiert zwei Proteinsequenzen aus der Liste und fügt diese zusammen. In den letzten beiden Fällen wird für alle Listengrößen die gleiche Datenmenge in die Zielvariable gespeichert, da immer nur ein bzw. zwei Elemente aus dem Ausdruck hervorgehen. Bei den Zuweisungen ohne Ausdruck sowie bei denen mit einfachem Ausdruck wurden bei allen Testgrößen Leistungseinbußen zwischen 358% und 14% beobachtet. Bei Zuweisungen mit komplexem Ausdruck beträgt der Leistungsverlust bei Testgrößen von 100 KB und 500 KB zwischen 131% und 38%. Erhöht man jedoch die Größe weiterhin, erreicht man beim verwendeten System ab etwa 4 MB eine Rentabilitätsschwelle, ab der die Ausführung durch ODE-TI um bis zu 18% im Vergleich zu Original ODE beschleunigt wird.

Der Ausdruckauswertungs-Pushdown wurde auf ähnliche Weise mit verschieden komplexen Ausdrücken evaluiert. Auch hier wird beim *einfachen Ausdruck* eine einzelne Proteinsequenz selektiert, während beim *komplexen Ausdruck* zwei Sequenzen konkateniert werden. Hier wird im Gegesatz zum Zuweisungs-Pushdown keine Schreiboperation durchgeführt, es wird lediglich der Ausdruck anhand der Liste der Proteinsequenzen ausgewertet. Beim Messen der Operationen mit einfachem Ausdruck stellten sich bis 9 MB Größe Leistungsverluste gegenüber Original ODE im Bereich von 127% bis 17% ein. Die Rentabilitätsschwelle wird

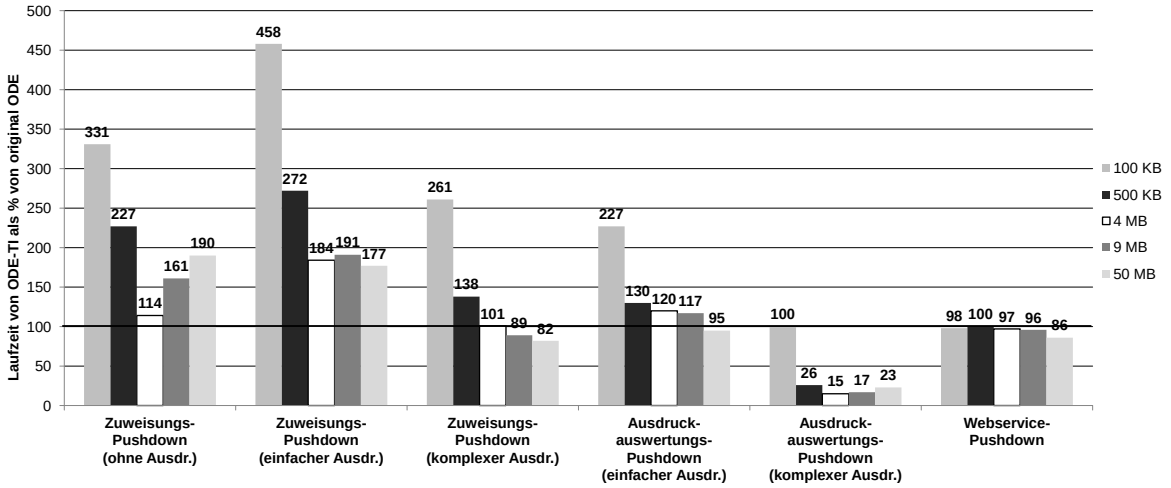

Abbildung 3.6: Ergebnisse der Messungen von ODE-TI relativ zur Laufzeit von Original ODE in Prozent.

hier bei der 50 MB-Messung erreicht, wo eine Leistungssteigerung von 5% erkennbar ist. Beim komplexen Ausdruck wurde durch den Einsatz von ODE-TI bei Datengrößen ab 500 KB Leistungssteigerungen zwischen 74% und 85% gemessen. Der Webservice-Pushdown erzielt ähnliche Ergebnisse wie Original ODE, mit einer Leistungssteigerung bis zu 14% für steigende Datengrößen.

Diese Messungen haben gezeigt, dass die Pushdown-Techniken für sich allein betrachtet nur dann eine Leistungssteigerung bewirken, wenn komplexe Ausdrücke auf einer großen Datenmenge ausgewertet werden sollen. Vergleicht man die Messergebnisse des Zuweisungs-Pushdowns mit denen des Ausdruckauswertungs-Pushdowns, stellt man fest, dass die zusätzliche Schreiboperation bei der Zuweisung von Original ODE wesentlich effizienter durchgeführt wird als von ODE-TI. Dies lässt sich dadruch erklären, dass das DBMS zusätzliche Log-Informationen zu jeder Anfrage auf die Festplatte speichert. Außerdem müssen beim Einfügen oder Ändern von Daten auch betroffene Indizes, die Leseoperationen auf der Datenbank beschleunigen sollen, angepasst werden. Bei den Messungen der Zuweisungen ohne Ausdruck nimmt die Leistung von ODE-TI bei den Größen bis 4 MB schrittweise zu, verschlechtert sich jedoch ab 9 MB wieder deutlich. Dies lässt sich dadurch erklären, dass die durch ihre Begleitoperationen ineffiziente Schreiboperation einen immer größeren Einfluss auf die Gesamtlaufzeit nimmt. Bei allen anderen Messungen wird die Schreiboperation durch die vorherige Selektion von einem bzw. zwei Elementen konstant gehalten. Dadurch hat die Schreiboperation bei wachsender Datengröße einen immer geringeren Einfluss auf die Gesamtlaufzeit. Der Webservice-Pushdown erzielt in ODE-TI und Original ODE ähnliche Ergebnisse. Dies rührt daher, dass das Ergebnis eines WS-Aufrufs durch Invoke bei Original ODE grundsätzlich durch den Persistenzmanager im DBS gespeichert wird, damit es für eine etwaige, spätere Wiederherstellung persistent ist. Dies wiegt die Schreiboperation von ODE-TI nach Erhalt der Antwort in vergleichbarem Maße auf, für größere Daten ist jedoch

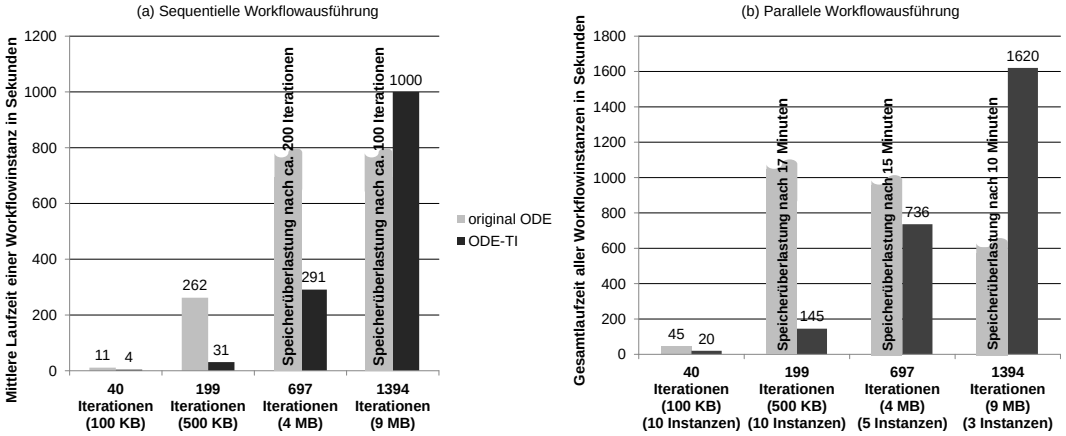

Abbildung 3.7: Messergebnisse der sequenziellen (a) und parallelen (b) Ausführung des Proteinmodellierungsworkflows(vgl. [RSM11])

auch hier eine Tendenz zu Gunsten von ODE-TI erkennbar, nicht zuletzt weil dort der Datenverkehr zwischen WfMS und DBS größtenteils entfällt.

Am Beispielszenario des Proteinmodellierungsworkflows wurden jeweils Messungen bei sequenzieller und paralleler Ausführung durchgeführt. Bei der sequenziellen Ausführung wurden jeweils 100 Instanzen mit den Datengrößen 100 KB und 500 KB sowie 50 Instanzen mit den Größen 4 MB und 9 MB erzeugt. Messungen mit Datengrößen von 50 MB und größer konnten wegen der bereits angesprochenen Limitierungen von ODE (Speicher, Laufzeiten) nicht durchgeführt werden. Bei der Listengröße von 100 KB wurde eine Leistungssteigerung um Faktor 3 gemessen, eine Verbesserung um Faktor 8 zeigte sich bei 500 Kb Größe. Original ODE stürzte bei 4 MB und 9 MB nach ca. 200 bzw. 100 Iterationen wegen einer Überladung des Hauptspeichers ab und konnte daher kein vergleichbares Ergebnis liefern, ODE-TI konnte jedoch alle 50 Instanzen der beiden Workflows ohne Probleme bis zu deren Beendigung ausführen.

Bei den parallelen Messungen wurden alle Workflowinstanzen gleichzeitig gestartet. Es wurden jeweils 10 Instanzen der Größen 100 KB und 500 KB, 5 Instanzen mit 4 MB, sowie 3 Instanzen mit 9 MB Listengröße ausgeführt. Auch hier zeigte ODE-TI eine Leistungssteigerung um Faktor 2 für 100 KB. Bei der parallelen Ausführung mehrerer Instanzen scheitert Original ODE bereits bei den Workflows mit 500 KB Listengröße an Hauptspeicherüberladungen. Dies lässt sich durch den addierten Speicherbedarf der zeitgleich ausgeführten Workflowinstanzen erklären.

Aus diesen Messergebnissen lässt sich schließen, dass der Einsatz der Pushdown-Techniken sowohl die Effizienz als auch die Zuverlässigkeit von Workflowausführungen steigern kann. Dies gilt sowohl bei sequenzieller als auch paralleler Workflowausführung bei großen Datenmengen und komplexen Ausdrücken.

3.4 Erweiterung von ODE-TI

Eine genauere Betrachtung des ODE-TI Prototyps zeigt einige Problemstellungen auf. Beispielsweise wurde hier der Literal-Pushdown, der eine wichtige Rolle beim Zuweisungs-Pushdown spielt, nicht implementiert. Zuweisungen, die Literalwerte verwenden, werden weiterhin auf konventionelle Weise innerhalb der ODE-Laufzeitkomponente durchgeführt und sind zunächst im DBS nicht sichtbar. Ebenso wurden etliche Aktivitäten, die, ähnlich wie ein WS-Aufruf, Daten in Form von Nachrichten an den Workflow heranführen bzw. aus dem Workflow versenden (z.B. Receive), nicht als Pushdownvarianten implementiert.

Diese Umstände haben zur Folge, dass vor dem Aufruf implementierter Pushdown-Aktivitäten oft ein zusätzlicher Schritt notwendig ist, um sicherzustellen, dass alle für die Aktivität erforderlichen Inhalte in der Datenbank vorhanden sind. Dieser zusätzliche Schritt wird in der momentanen Implementierung von ODE-TI vom Persistenzmanager durch einen vollständigen *Flush* realisiert. Dabei werden die workflowinternen Daten an das DBS übermittelt. Neben den eigentlich benötigten Variableninhalten, werden dabei zusätzlich auch Prozess- und Auditingdaten des WfMS unselektiv mit der Datenbank synchronisiert [Wag11]. Der dabei entstehende Aufwand wirkt sich negativ auf die Laufzeit der Workflowinstanzen aus.

Ziel dieser Arbeit ist es unter anderem die Probleme durch fehlende Daten in der Datenbank bzw. Leistungseinbußen durch die unselektive Synchronisation zu beheben. Dazu sollen im weiteren Verlauf Pushdown-Implementierungen für die restlichen datenrelevanten Aktivitäten erarbeitet werden. Die im Rahmen dieser Arbeit entstandenen Lösungensansätze werden im nächsten Kapitel vorgestellt.

4 Konzeptionelle Erweiterungen

Die im vergangenen Kapitel vorgestellte Umsetzung der Integration von WfMS und DBS enthält einige Lücken und Nebeneffekte. In diesem Kapitel werden nun weitere Konzepte zur Integration eines DBS vorgestellt, die im Rahmen dieser Arbeit entwickelt und untersucht wurden. Durch Umsetzung der fehlenden Konzepte aus Kapitel 3, sowie durch das Einführen neuer Pushdown-Konzepte, sollen zusätzliche Verbesserungen der Effizienz und Zuverlässigkeit bei der Workflowsausführung erreicht werden. Diese werden in den folgenden Unterabschnitten vorgestellt.

4.1 Literal-Pushdown

Der Literal-Pushdown wurde bereits in Abschnitt 3.2.3 und Abbildung 3.4 konzeptionell vorgestellt. Daten, die in Form von Literalen über das Prozessmodell in den Workflow eingebracht werden, sind zunächst nur in der Laufzeitkomponente des WfMS sichtbar und können vom DBS daher nicht verwendet werden. Wird innerhalb einer Zuweisung ein Literal verwendet, so kann diese Zuweisung bisher nur auf konventionelle Weise auf dem Variablenvorrat der Laufzeitkomponente durchgeführt werden. Wird die mit dem Literalinhalt beschriebene Variable nun Quelle einer Zuweisung, die durch Pushdown-Techniken implementiert ist, so muss vorher ein zusätzlicher Schritt durchgeführt werden, der sie in der Datenbank abspeichert. Um diesen Zwischenschritt zu umgehen, wurde der Literal-Pushdown bereits konzeptionell entwickelt, jedoch noch nicht implementiert.

Der Literal-Pushdown kann in einen schreibenden und einen lesenden Teil unterteilt werden. Beim schreibenden Teil werden Literalwerte aus dem Workflowmodell extrahiert und in der Datenbank gespeichert, während der lesende Teil die in der Datenbank gespeicherten Literalwerte für Zuweisungsoperationen verwendet. Im Folgenden werden unterschiedliche Varianten zur Umsetzung des Literal-Pushdowns vorgestellt und diskutiert. Sie kann grundsätzlich auf drei verschiedene Arten erfolgen, die sich hauptsächlich in Ort und Zeitpunkt des schreibenden Teils unterscheiden. Die verschiedenen Varianten des schreibenden Teils sind in Abbildung 4.1 dargestellt.

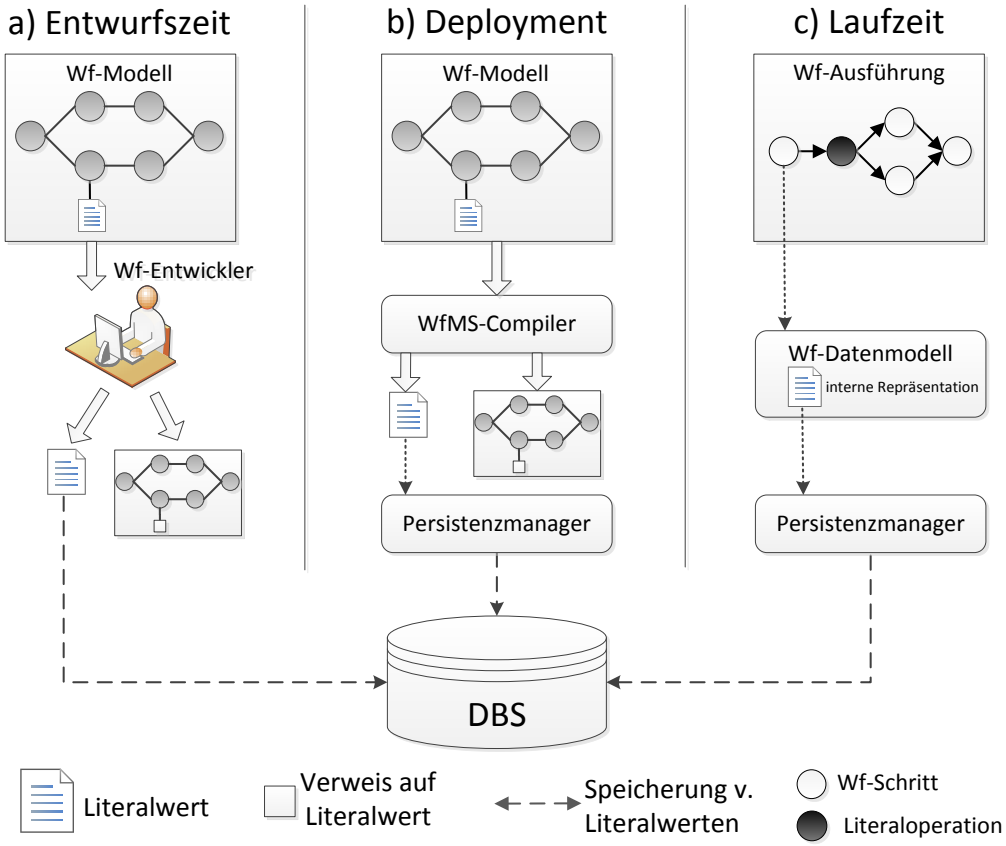

Abbildung 4.1: Verschiedene Konzepte zur Umsetzung des schreibenden Teils eines Literal-Pushdown zur Entwurfszeit, während des Deployments oder zur Laufzeit.

a) **Entwurfszeit**
Da Literalwerte bereits bei der Modellierung eines Workflows bekannt sind, können diese prinzipiell zur Entwurfzeit durch den Entwickler bzw. durch Entwicklungswerkzeuge extrahiert und in der Datenbank abgespeichert werden. Dies setzt jedoch voraus, dass die entsprechenden Datenbankbefehle dem Entwickler bekannt sind bzw. durch die verwendeten Entwicklungswerkzeuge implementiert werden. In der Laufzeitkomponente des ausführenden WfMS muss anschließend für Literalzuweisungen der lesende Teil des Literal-Pushdowns implementiert werden.

b) **Deployment**
Der schreibende Teil des Literal-Pushdowns kann, ähnlich wie beim Abspeichern der Literalwerte zur Entwurfszeit durch Entwicklungswerkeuge, auch beim Deployment eines Workflows durch die Deploymentkomponente des WfMS erfolgen. Zusätzlich

zum lesenden Teil muss, etwa im Wf-Compiler, noch eine Logik zum Festschreiben der Literalwerte implementiert werden. Die eigentlichen Datenbankbefehle zur Speicherung werden dabei üblicherweise vom Persistenzmanager (DAO-Schicht) implementiert.

c) **Laufzeit**
Literalwerte, die auf herkömmliche Weise in den Workflowkontext geladen wurden, sind während der Ausführung als Teil des im Hauptspeicher befindlichen Datenmodells der entsprechenden Ausführungsinstanz verfügbar. So können beispielsweise alle Literalwerte zu Beginn der Ausführung in der Datenbank gespeichert werden. Die Anforderungen an die Umsetzung dieser Variante des Literal-Pushdows sind dabei prinzipiell die selben wie in Variante (b), konzeptionell wird hierbei lediglich der schreibende Teil in die Laufzeitkompomente verlagert.

Bei allen drei Varianten muss der lesende Teil des Literal-Pushdowns innerhalb der Zuweisungslogik eines WfMS implementiert werden und mit dem schreibenden Teil abgestimmt sein. Der lesende Teil stellt damit konzeptionell eigentlich eine Erweiterung des Zuweisungs-Pushdowns dar. Bei allen vorgestellten Varianten muss nach der Extraktion eines Literals durch den schreibenden Teil ein Verweis hinterlassen werden, mit dessen Hilfe der Literalwert vom lesenden Teil zur Laufzeit in der Datenbank wiedergefunden werden kann. Ein individueller Workflow wird ein Mal modelliert. Das fertige Modell kann dann mehrmals (z.B. in verschiedenen WfMS) als Prozess deployt werden, welcher wiederum mehrmals instanziiert und ausgeführt werden kann. Daraus ergibt sich eine $1 : n : m$ -Beziehung zwischen Modellierung, Deployment und Ausführung eins Workflows, wobei $1 \leq n \leq m$. Aus dieser Tatsache ergibt sich zunächst die Überlegung, dass Variante (a) den geringsten Laufzeitaufwand darstellt und daher angestrebt werden sollte.

Bei Variante (a) wird der schreibende Teil in die Entwicklungswerkzeuge ausgelagert oder gar gänzlich dem Entwickler überlassen. Problematisch ist dabei allerdings, dass hier die Modellierungs- und Ausführungsphase des Workflowmanagements miteinander verwoben werden und dadurch nicht mehr unabhängig voneinander durchgeführt werden können. Weiterhin verwenden unterschiedliche Workflow-Ausführungsumgebungen auch unterschiedliche Datenbanksysteme. Deswegen müssten die Daten dennoch in mehrere Datenbanksysteme und damit auch mehrmals gespeichert werden, wobei hierfür sogar entsprechende Integrationsprobleme bzgl. heterogener Datenbanksysteme gelöst werden müssen. Damit ergeben sich die oben genannten Performance-Vorteile dieser Variante (a) nur in sehr seltenen Fällen. Außerdem soll der Entwickler auch nicht mit zusätzlichen Aufgaben belastet werden. Auf Grund dieser Überlegungen sehen wir in dieser Arbeit von einer Umsetzung dieser Variante (a) ab.

Die Varianten (b) und (c) lässen sich ohne Auswirkungen auf die Modellierung umsetzen. Die Entscheidung zwischen Variante (b) und (c) hängt davon ab, ob das Deployment oder die Ausführung die performanzkritische Phase des Lebenszyklus eines Workflows darstellt. Bei den in dieser Arbeit betrachteten Fällen sind die Auswirkungen auf die Deploymentphase vernachlässigbar, da Prozesse meist nur ein Mal in der selben Umgebung deployt werden und somit der Fall $n = 1$ eintritt, wodurch der Pushdown nach Variante (b) nur ein Mal ausgeführt werden muss. Nachdem ein Prozess nach Variante (b) deployt wurde, können

die gespeicherten Literalwerte von allen seinen Ausführungsinstanzen verwendet werden, ohne dass der schreibende Teil des Pushdowns wiederholt werden muss. Bei Variante (c) müssen Literale für jede Instanz eines Workflows neu geschrieben werden, daher ist Variante immer (b) effizienter als Variante (c), wenn ein deployter Workflow mehrmals ausgeführt werden muss. Außerdem besteht dabei zusätzlich die Möglichkeit, Literalwerte nach dem Deployment und vor der Ausführung eines Workflows auf externem Wege direkt in der Datenbank zu bearbeiten. Wenn also eine Flexibiliät zur Laufzeit gewünscht ist, können die Literalwerte auch bei Variante (b) noch nachträglich geändert werden. Details zu einer Implementierung dieser Variante in Apache ODE sowie eine Evaluation werden in den Kapiteln 6 bzw. 7 behandelt.

4.2 Nachrichten-Pushdown

Abgesehen von Variableninhalten und Literalwerten gibt es noch eine weitere Art von Daten, die während einer Workflowausführung verwaltet werden. Ein BPEL-Prozess kann SOAP-Nachrichten von anderen Prozessen empfangen und an andere Prozesse senden. Dies geschieht insbesondere jeweils am Anfang und am Ende eines Workflows, wenn die Ausführung durch eine Eingangsnachricht gestartet oder die Antwortnachricht nach der erfolgreichen Beendigung zurückgesendet wird. Die Eingangsnachricht enthält Daten, die im Verlauf der Workflowausführung meist in irgendeiner Form Gegenstand von Zuweisungen oder Ausdruckauswertungen sind. Der datenrelevante Teil dieser Nachricht wird unmittelbar nach deren Verarbeitung in einer *Eingangsvariable* abgelegt. Um auch diese Daten ohne aufwendigen Zusatzschritt innerhalb von Pushdown-Aktivitäten nutzen zu können, wird an dieser Stelle der *Nachrichten-Pushdown* vorgestellt.

Beim Nachrichten-Pushdown soll, ähnlich wie beim Zuweisungs-Pushdown, die Zuweisung des Inhalts einer Nachricht in die Eingangsvariable in der Datenbank unmittelbar nach dem Empfang der Nachricht erfolgen. Zwei der dabei in Frage kommenden Umsetzungen sind in Abbildung 4.2 dargestellt. Bei der Variante (a) wird der komplette Inhalt einer Nachricht nach ihrem Empfang bei der Workflowausführung zum Speichern über den Persistenzmanager an das DBS übermittelt. Anschließend kann der datenrelevante Teil der Nachricht innerhalb der Datenbank der Eingangsvariable zugewiesen werden. Variante (b) setzt zunächst voraus, dass die Nachricht bereits in irgend einer Form im DBS gespeichert ist. Indem die Nachricht innerhalb des DBS einer Variablen zugewiesen wird, wird der Kommunikationsaufwand zwischen Laufzeitkomponente und DBS verringert. Je nach Implementierung eines WfMS kann die Nachricht zum Zeitpunkt des Beginns der Workflowausführung bereits z.B. durch Komponenten des Nachrichtenaustausches im DBS persistent gemacht worden sein. Falls diese Voraussetzung erfüllt ist, ist Variante (b) für die Umsetzung eines Nachrichten-Pushdowns zu bevorzugen, da durch den verringerten Kommunikationsaufwand ein besseres Laufzeitverhalten zu erwarten ist. Variante (a) *muss* hingegen eingesetzt werden, wenn die Nachricht nicht zuvor im DBS gespeichert wurde.

Bei der Umsetzung nach der Variante (b) liegt der Unterschied zum Zuweisungs-Pushdown lediglich darin, dass die Nachrichten wahrscheinlich an einer anderen Stelle der Datenbank

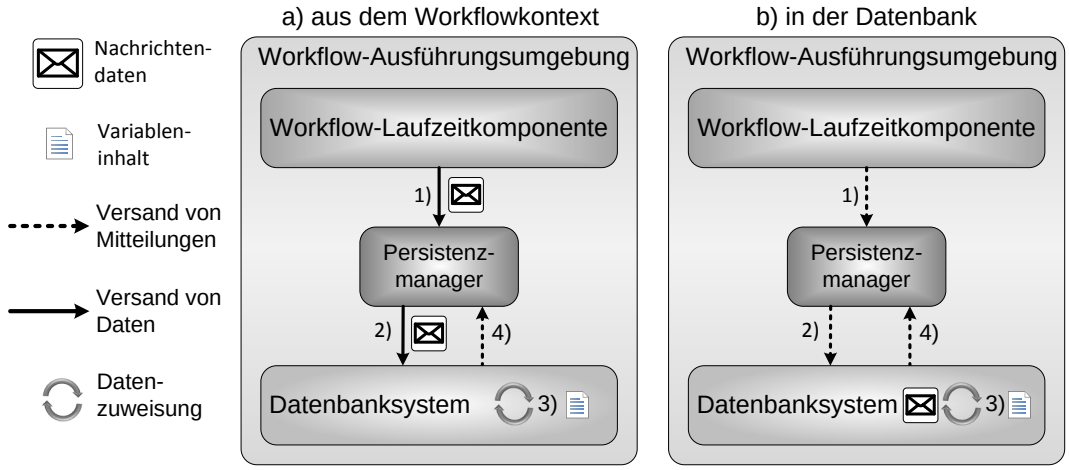

Abbildung 4.2: Unterschiedliche Umsetzungen des Nachrichten-Pushdowns aus dem Workflow heraus (a) bzw. in der Datenbank (b). Die Zahlen deuten die Reihenfolge an, in der einzelne Teilschritte ausgeführt werden.

und in einer anderen Form gespeichert werden als die Variablen. Der Ablauf ist dabei identisch mit dem des Zuweisungs-Pushdowns in Abbildung 3.3(a), jedoch wird beim Nachrichten-Pushdown kein Ausdruck im engeren Sinne evaluiert, da immer der selbe (komplette) Inhalt einer Nachricht kopiert werden muss. Statdessen kann eine Transformation zwischen den möglicherweise unterschiedlichen Formaten der Nachrichten und Variablen notwendig sein.

Der umgekehrte Weg, die Zuweisung einer Variablen in eine Nachricht, kann problematisch werden. Da das DBS konventionellerweise nicht bei der Workflowausführung eingesetzt wird, werden im DBS gespeicherte Nachrichten auch nicht als Quelle für den Nachrichtenversand verwendet. Es kann zwar eine Variable wieder in eine Nachricht kopiert werden, diese wird aber u.U. nicht vom WfMS ausgewertet. In diesem Fall bietet sich die Variante (a) des Nachrichten-Pushdowns in umgekehrter Form an: Der Inhalt einer Ausgangsvariable wird aus der Datenbank in den Workflowkontext geladen und dort zu einer Nachricht verarbeitet, die dann auf konventionelle Weise an einen Partnerprozess versendet wird. Es wäre denkbar weitere Komponenten des WfMS zu verändern, damit diese die Nachrichten aus der Datenbank nach Variante (b) verarbeiten. Dieser Ansatz kann jedoch im Verhältnis zum erwarteten Nutzen zu komplex ausfallen, da die Verarbeitung von Eingangs- und Ausgangsnachrichten erfahrungsgemäß keine signifikanten Auswirkungen auf die Gesamtlaufzeit einer Workflowinstanz hat. Typischerweise bestehen Eingangs- wie auch Ausgangsnachrichten einzelner Workflow-Instanzen nur aus kleineren Parameterwerten, die bei der Gesamtlaufzeit - insbesondere von datenintensiven Workflows - weniger ins Gewicht fallen. Der Nachrichten-Pushdown ist für eingehende Nachrichten dennoch wichtig, um etwaige Zwischenschritte für das Herunterschreiben von Daten vor anderen Pushdown-Aktivitäten zu vermeiden. Im Rahmen dieser Arbeit wurde auf eine auwendige Implementierung des Pushdowns für

ausgehende Nachrichten verzichtet und dafür stattdessen die Umgekehrte Form der variante (a) genutzt.

Eine weitere denkbare Variante für eine möglicherweise effizientere Umsetzung des Nachrichten-Pushdowns wäre die Verlagerung des Nachrichtenaustasches in das DBS analog zum Webservice-Pushdown. Dazu müssten Eingangsnachrichten in Form von Prozeduraufrufen an das DBS kommuniziert werden, welches dann die gesamte Logik zur Instanziierung von Workflows selbst implementieren muss und damit Aufgaben der Workflowverwaltung übernimmt. Diesem Implementierungsaufwand steht die Tatsache gegenüber, dass der erwartete Geschwindigkeitszuwachs beim Nachrichten-Pushdown im Verhältnis zur Laufzeit der gesamten Workflowausführung, wie bereits erwähnt, sehr gering ist. Aus diesem Grund wurde diese Variante im Rahmen dieser Arbeit nicht umgesetzt.

Details zu einer Implementierung des Nachrichten-Pushdowns in ODE-TI sowie dessen Evaluation wird in den Kapiteln 6 bzw. 7 behandelt.

4.3 XQuery-Pushdown

In den vergangenen Abschnitten wurden konzeptionelle Erweiterungen vorgestellt, die die Ausführung der bisher in ODE-TI umgesetzten Pushdowntechniken ermöglichen, unterstützen und verbessern sollen. An dieser Stelle wird nun mit dem *XQuery-Pushdown* eine Erweiterung vorgestellt, die einen Zusatz zur bisherigen Funktionalität darstellt.

Die bisherige Umsetzung von ODE-TI ist derzeit nur in der Lage, XPath-Ausdrücke innerhalb von Zuweisungen und Ausdruckauswertungen zu verarbeiten. Dies genügt zwar der Midestanforderung durch die Spezifikation von BPEL, Apache ODE bietet aber beispielsweise die Möglichkeit, XQuery-Ausdrucke innerhalb von datenverarbeitenden BPEL-Aktivitäten zu verwenden. Die Evaluation der bisherigen Pushdown-Konzepte (s. Abschnitt 3.3.2) hat gezeigt, dass gerade bei komplexen Ausdrücken die Ausführung innerhalb des DBS effizienter ist, als die Ausführung im WfMS. Da XQuery-Ausdrücke wesentlich komplexer als XPath-Ausdrücke werden können, ist also durch den XQuery-Pushdown eine Leistungssteigerung bei deren Auswertung zu erwarten. Die Erweiterung von ODE-TI um die Verarbeitung von XQuery-Ausdrücken stellt keine konzeptionelle Innovation dar. Eine Implementierung des XQuery-Pushdowns ist im Rahmen dieser Arbeit nicht durchgeführt worden.

5 Architektur von Apache ODE

Im vergangenen Kapitel wurden Konzepte und Techniken vorgestellt, die die Nutzung von Datenbanken innerhalb von Workflowmanagementsystemen behandeln. In diesem Kapitel wird *Apache ODE*[1] (**O**rchestration **D**irector **E**ngine) als konkretes WfMS mit den für diese Arbeit relevanten Teilen seiner Architektur detailiert vorgestellt. Die Wahl von Apache ODE als WfMS ist hier bedingt durch den Prototyp aus [Wag11], der die Konzepte aus Abschnitt 3.4 in Apache ODE implementiert und der in dieser Arbeit erweitert werden soll.

5.1 Gesamtarchitektur

Apache ODE ist eine Java-basierte Workflowengine zum Deployment und zur Ausführung von BPEL-Prozessen. Das Überwachen, Anhalten und Fortsetzen von Workflowinstanzen ist dabei eingeschränkt über ein Application Programming Interface (API) oder über eine Webseite möglich. BPEL-Prozesse, die in Apache ODE bekannt gemacht wurden, werden in Form von Webservice-Aufrufen instanziiert. Aus diesem Grund muss Apache ODE in eine Kommunikationsinfrastruktur für Webservices eingebettet werden. Üblicherweise wird dazu ein Apache Tomcat Server[2] mit Axis2 verwendet. Es besteht ebenfalls die Möglichkeit Apache ODE in den Apache ServiceMix[3] einzubetten, dieser wird im Rahmen dieser Arbeit jedoch nicht verwendet.

Die Gesamtarchitektur wird in Abb. 5.1 veranschaulicht. In WS-BPEL definierte Prozesse werden durch den *ODE BPEL Compiler* (oben) zuerst in ein Java Objektschema übersetzt und anschließend serialisiert als Datei abgespeichert. Für das Instanziieren des Prozesses und seiner WS-Aufrufe müssen die entsprechenden *WSDL Dateien* übergeben werden. Um XML-Daten (Literalwerte) innerhalb einer BPEL-Prozessdatei zu validieren, müssen die entsprechenden *XML-Schemadateien* beim kompilieren vorliegen. Wurde ein Prozess durch den BPEL-Compiler erfolgreich kompiliert und bekannt gemacht, kann dieser ab sofort instanziiert und aufgerufen werden.

Die *ODE BPEL Runtime* besteht aus vielen Komponenten, von denen in Abb 5.1 nur die für die Gesamtarchitektur wichtigsten dargestellt sind:

[1] http://ode.apache.org/
[2] http://tomcat.apache.org/
[3] http://servicemix.apache.org/

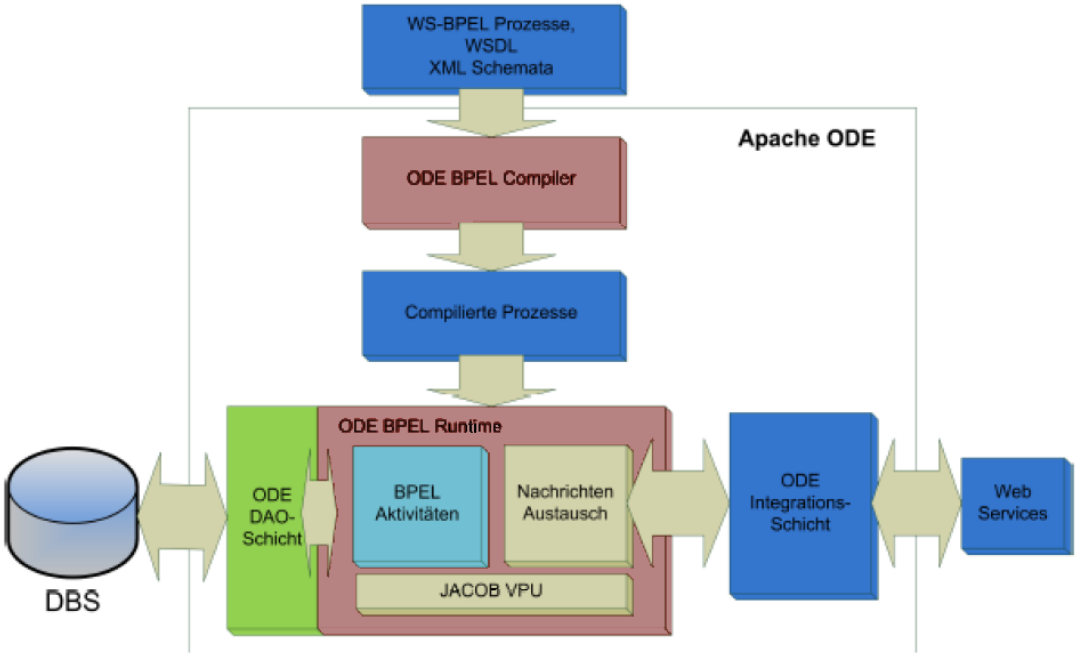

Abbildung 5.1: Gesamtarchitektur von Apache ODE. (Vgl. [ode])

- **JACOB VPU**[4]: Die *JACOB Virutal Processing Unit* ist ein für ODE entwickeltes Rahmenwerk, das die Grundlage der Architektur der BPEL Runtime bildet. Dieses Rahmenwerk koordiniert die gleichzeitige Ausführung von ODE-Prozessinstanzen und sorgt für die Dauerhaftigkeit deren Ausführungszustände.
- **BPEL Aktivitäten**: Die durch den OASIS-Standard (s. Abschnitt 2.4) definierten und von ODE umgesetzten BPEL-Aktivitäten sind in dieser Komponente der Runtime implementiert. Die verschiedenen Aktivitäten interagieren mit der benachbarten DAO-Schicht sowie mit der Komponente für den Nachrichtenaustausch, um Daten (z.B. Variableninhalte) einer Prozessinstanz persistent zu halten und mit anderen Webservices zu kommunizieren.
- **ODE DAO-Schicht**: Die DAO-Schicht (**D**ata **A**ccess **O**bject) ist für die Speicherung von Prozess- und Instanzdaten zuständig. Hier ist die Kommunikation mit dem eingebundenen DBS implementiert, die dadurch von den Aktivitäten der Runtime gekapselt wird (siehe auch Abschnitt 3.1).
- **Nachrichtenaustausch**: Diese Komponente verwaltet den Empfang und den Versand von WSDL-Nachrichten zwischen Prozessinstanzen und anderen Webservices. Sie interagiert über die *ODE Integrationsschicht* der verwendeten Infrastruktur (z.B. Axis2, ServiceMix) mit den beteiligten Webservices.

[4] http://ode.apache.org/jacob.html

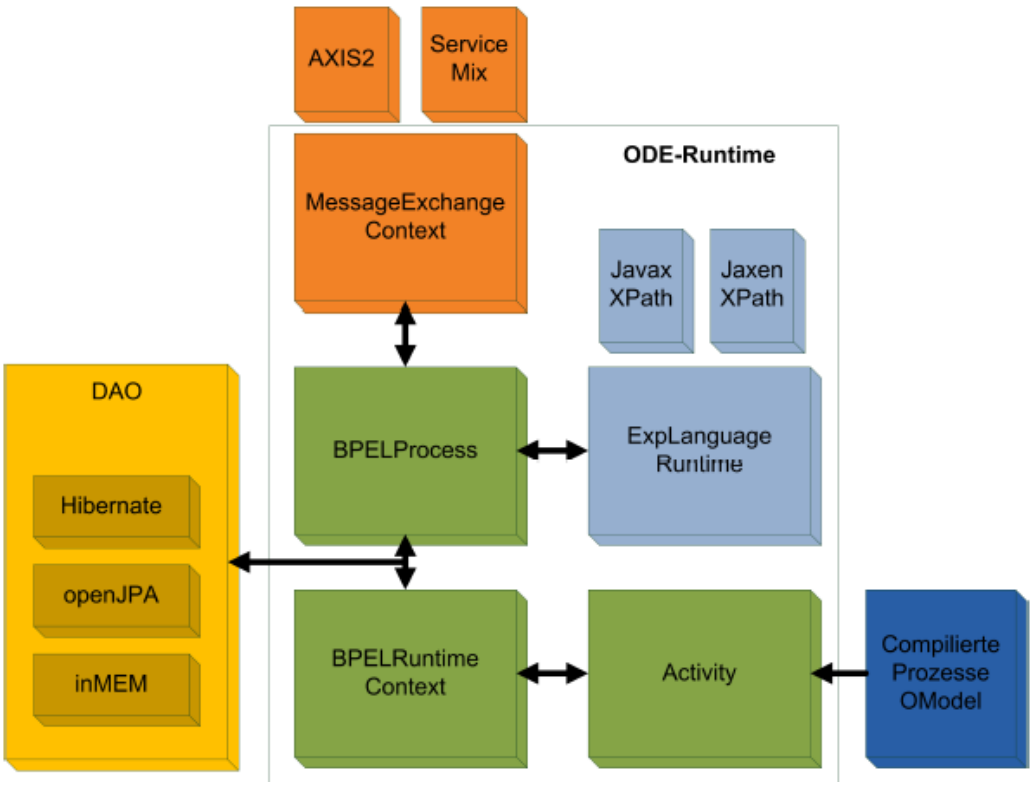

Abbildung 5.2: Detailiertere Ansicht der ODE Runtime (Vgl. [Wag11])

5.2 Runtime im Detail

Die im vorigen Abschnitt beschriebene ODE Runtime-Komponente, sowie mehrere Bestandteile davon, werden nun in Abb. 5.2 genauer betrachtet. Hierbei ist zu beachten, dass immer noch eine starke Abstraktion der tatsächlichen Implementierung vorliegt. Die einzelnen Komponenten und ihr Zusammenspiel werden im folgenden Vorgestellt.

- **DAO-Schicht**: Bei der Speicherung von Prozess- und Instanzdaten kann bei Apache ODE zwischen drei unterschiedlichen Implementierungen gewählt werden: *Hibernate*[5], *openJPA*[6] sowie in-memory Ausführung. Hibernate und openJPA sind Middleware-Systeme, die markierte Objekte einer Java-Anwendung in einem DBS persistent machen. Die beiden Systeme unterscheiden sich lediglich in den einbindbaren DBS und kleineren

[5] http://www.hibernate.org/
[6] http://openjpa.apache.org/

Implementierungsdetails [Wag11]. Die in-memory Ausführung von BPEL-Prozessen unterliegt einigen Einschränkungen und wird daher nur in Sonderfällen eingesetzt.

- **Ausführungslogik**: Die Komponenten *BPELProcess*, *BPELRuntimeContext* und *Activity* bilden in diesem Modell die Ausführungslogik. *Activity* implementiert die BPEL-Aktivitäten und deren Logik. Der *BPELRuntimeContext* implementiert die Zugriffsfunktionen auf die zu verwendende DAO-Schicht und enthält unter anderem die Laufzeitparameter von ODE (z.B. Datenbank-Einstellungen). Die Komponente *BPELProcess* verwaltet die Informationen zu einem kompilierten BPEL Prozess, wie aufzurufende Webservices und die im Prozess verwendeten Query-Sprachen (XPath, XQuery etc.). Die Auswertung von Ausdrücken sowie der Aufruf von Webservices innerhalb eines Prozesses erfolgt daher über diese Komponente.
- **Query-Auswertung**: Für die Evaluierung von XPath-Ausdrücken in BPEL-Aktivitäten werden das Jaxen und Javax Framework verwendet.
- **Kommunikationsinfrastruktur**: Als Kommunikationsinfrastruktur kann wie bereits erwähnt entweder Axis2 oder der ServiceMix verwendet werden.

In den folgenden Abschnitten stellen wir das Objektmodell, die Hibernate DAO-Schicht und die Runtime-Schicht vor.

5.2.1 OModel und BPEL-Typsystem

Durch den ODE BPEL Compiler erzeugte Prozesse werden intern durch das OModel repräsentiert. Jede Aktivität sowie die meisten Sprachkonstrukte (Literalwerte, Ausdrücke) innerhablb eines BPEL-Workflows werden durch ein entsprechendes Objekt aus diesem OModel gespeichert. Abbildung 5.3 stellt den für diese Arbeit wesentlichen Teil des OModel dar.

- **OBase** ist die Superklasse aller weitere OModel-Klassen. Die Methode dehydrate() wird von den jeweiligen Unterklassen erweitert und erlaubt so die Informationen, die in einem OModel-Objekt gespeichert sind, aus dem Hauptspeicher zu entfernen um so Systemressourcen frei zu geben. Dies kann z.B. bei lang laufenden Prozessen mit hohen Wartezeiten sinnvoll sein.
- **OScope** repräsentiert ein BPEL Scope, einen Sichtbarkeitsblock für Variablen mit statischer Namensbindung (vgl. Abschnitt 2.4). Dieser trägt die Informationen zu allen Variablen, die in diesem Block definiert wurden.
- **OScope.Variable** stellt eine BPEL-Prozessvariable dar. Dieses Objekt beinhaltet den Namen sowie den Typ der Variable sowie eine Referenz auf den Scope in dem sie deklariert ist.
- **OVarType** ist die Oberklasse der im OModel repräsentierten BPEL Datentypen, denen eine Variable angehören kann. Die BPEL-Variablentypen werden später erläutert.
- **OActivity** bildet die Oberklasse für alle Objekte, die BPEL-Aktivitäten repräsentieren.

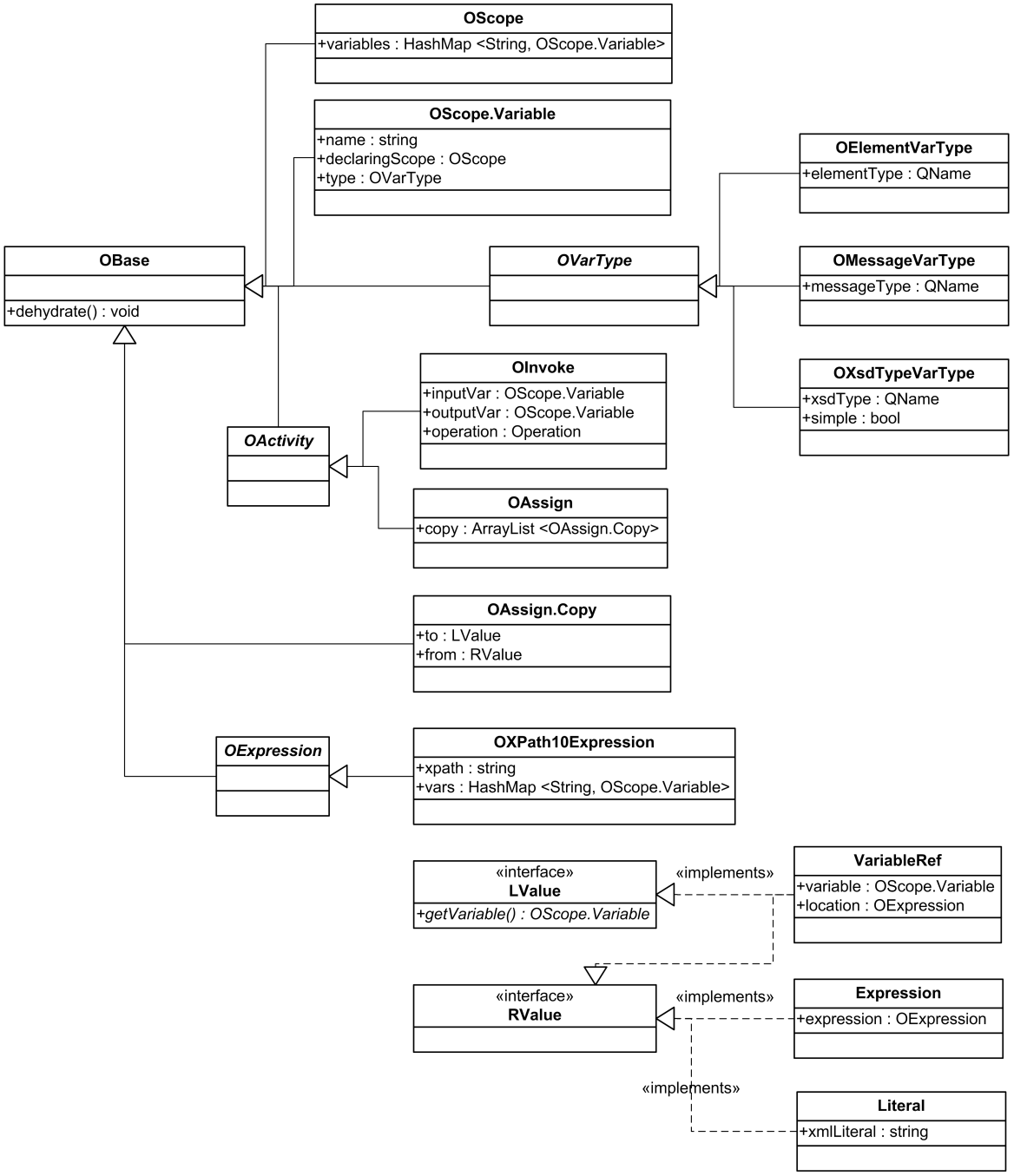

Abbildung 5.3: Ausschnitt des ODE OModel als UML-Diagramm (Vgl. [Wag11]).

- **OInvoke** repräsentiert eine Invoke-Aktivität. Gespeichert werden die Variable, die die Ausgangsnachricht hält (inputVar), die Variable, in welche die Eingangsnachricht gespeichert wird (outputVar), und die aufzurufende WSDL Operation (Operation). Der Webservice wird in seiner WSDL-Datei beschrieben und über BPEL partnerLinks (OPartnerLink) eingebunden.
- **OAssign** repräsentiert die Assign-Aktivität, mit der Zuwseiungen erfolgen. Diese kann mehrere Copy Blöcke beinhalten (OAssign.Copy).
- **OAssign.Copy** stellt einen Copy Block innerhalb einer Assign-Aktivität dar. Gespeichert werden die linke Seite (to) und die rechte Seite der Zuweisung (from). Die linke Seite muss auf eine Variable referenzieren, weshalb das entsprechende Interface *LValue* die getVariable() Methode implementieren muss. Die rechte Seite der Zuweisung (RValue) kann eine Variable (VariableRef), einen Ausdruck (Expression) oder ein Literal enthalten. Literale sind Start- bzw. Initialwerte für BPEL-Variablen und werden im Prozessmodell definiert.
- **OExpression** ist die Oberklasse für alle Query-Sprachen, die implementiert sind. Wir betrachten zunächst nur die Unterklasse OXPath10Expression, welche XPath1.0 Ausdrücke repräsentiert. Sie beinhaltet den XPath-Ausdruck sowie alle an dem Ausdruck beteiligten Variablen.

Die *OVarType*-Unterklassen *OMessageVarType*, *OXsdTypeVarType* und *OElementVarType* repräsentieren die im BPEL-Standard [bpe] spezifizierten Typen *WSDL message*, *XML Schema* und *XML Schema element*, die eine Variablendeklaration enthalten darf. Der jeweils zugehörige XML Schema Typ aus der Prozessdefinition wird als Qualified Name (QName) gespeichert. Zur Verwendung und Manipulation von XML Daten innerhalb von Apache ODE werden intern die Wrapper-Elemente <*message/*> , <*xsd-complex-type-wrapper/*> und <*temporary-simple-type-wrapper/*> verwendet, Variablen vom Typ *OMessageVarType*, *OXsdTypeVarType*(complex) bzw. *OXsdTypeVarType*(simple) zu speichern. Diese Wrapper-Elemente werden benötigt um z.B DOM-Operationen und XPath-Auswertungen korrekt zu verarbeiten zu können.

Im OModel werden keine Variableninhalte gespeichert. Lediglich im Prozessmodell definierte Literalwerte werden durch die *Literal*-Implementierung des *RValue*-Interfaces im OModel vorgehalten. Die Speicherung von Variableninhalten erfolgt in der DAO-Schicht, die im nächsten Abschnitt näher betrachtet wird.

5.2.2 ODE Hibernate DAO

In diesem Abschnitt wird nun vorgestellt, wie die Variablenspeicherung mittels der DAO-Schnittstellen bewerkstelligt wird. Es wird Hibernate DAO verwendet, da dies die vom zu erweiternden Prototyp verwendete Implementierung ist. Von besonderem Interesse sind dabei nur die DAO-Schnittstellen zu Scopes und Variableninhalten. Die entsprechenden Komponenten sind in Abbildung 5.4 dargestellt.

Über die *ProcessInstanceDAO*- und die *ScopeDAO*-Schnittstelle erhält man Zugriff auf die Informationen zu den *XmlDataDAO*-Schnittstellen, welche die Daten zu den Variablen

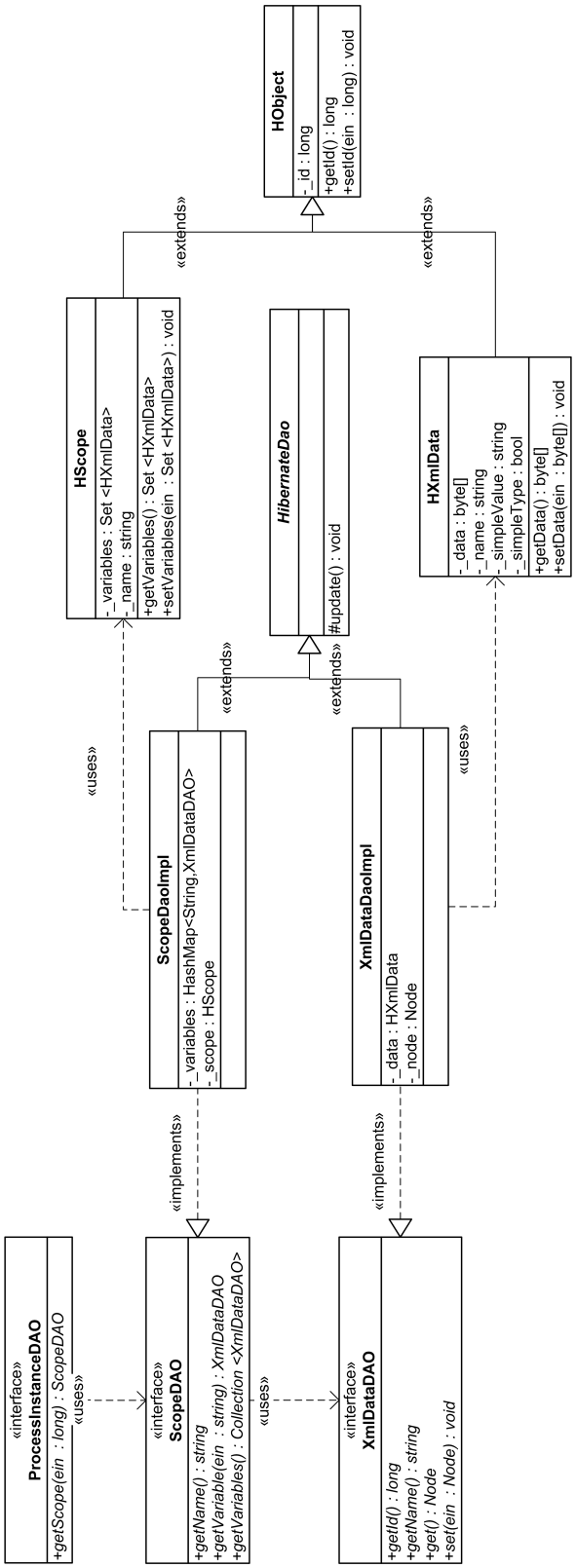

Abbildung 5.4: Ausschnitt der ODE DAO-Schicht als UML-Diagramm (Vgl. [Wag11]).

```
/**
 * @hibernate.class table="BPEL_XML_DATA"
 */
public class HXmlData extends HObject {

  private byte[] _data;
  ...

  /**
   * @hibernate.property type="byte[]"
   * @hibernate.column name="DATA" sql-type="BLOB"
   */
  public byte[] getData() {
    return _data;
  }

  public void setData(byte[] data) {
    _data = data;
  }
    ...
}
```

Listing 5.1: Beispiel für die Annotation einer Java Klasse, die von Hibernate synchronisiert werden soll.

beinhalten. Diese werden über Getter- und Settermethoden verfügbar gemacht. Die *ProcessInstanceDAO*-Schnittstelle bildet die Verbindung zum *BpelRuntimeContext*, die genaue Implementierung ist für uns dabei uninteressant.

In der Oberklasse *HibernateDao* werden unter anderem aktuelle DB-Sitzungen verwaltet und Hibernate-Methden (z.B. update()) zur Verfügung gestellt. Die Unterklassen *ScopeDaoImpl* und *XmlDataDaoImpl* sind Hibernate-Implementierungen der entsprechenden Interfaces.

XmlDataDaoImpl enthält ein Attribut vom Typ *W3C Node* (_node), sowie ein Attribut vom Typ *HXmlData* (_data). Ein XML-Dokument wird in _node gespeichert. Falls dieses größer als 256 Zeichen ist wird es in eine Byte-Repräsentation konvertiert und in *HXmlData*._data gespeichert. Andernfalls wird es als HXmlData._simpleValue gespeichert. Dies wird aus Performanzgründen durchgeführt, um String- anstatt BLOB-Felder für kleine Inhalte innerhalb der DB zu verwenden. Entsprechend referenziert *ScopeDaoImpl* auf ein Objekt vom Typ *HScope* in dem unter anderem der Name des Scopes gespeichert wird.

Objekte von *HScope* und *HXmlData* stellen durch die Hibernate Middleware direkt Zeilen entsprechender Datenbanktabellen dar. Hibernate verwaltet die Synchronisierung dieser Objekte über die Getter-/Settermethoden und durch Überwachung des Java Bytecodes selbstständig. Dazu müssen die Datenfelder solcher Objekte entsprechend annotiert werden (siehe Listing 5.1).

Aus diesen Annotationen ergeben sich im Fall von *HScope* und *HXmlDate* die Tabellenschemata für die Datenbank, die in Abb. 5.5 dargestellt sind.

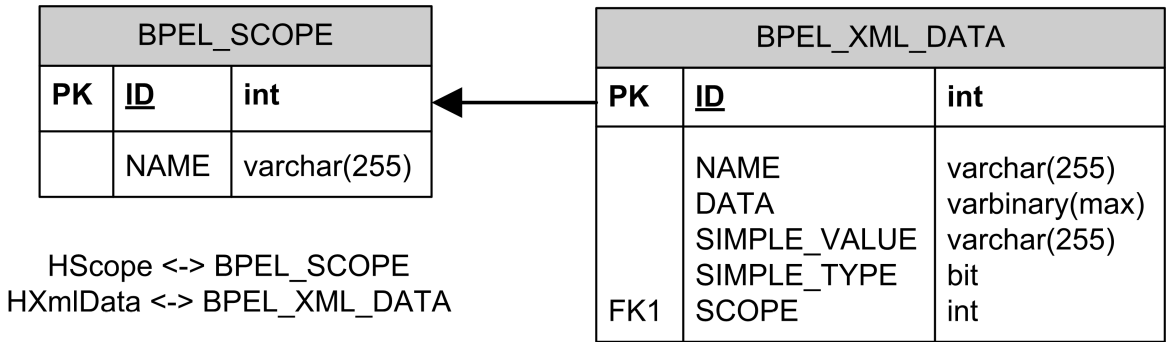

Abbildung 5.5: Tabellenschema, welches sich durch die Hibernate Middleware direkt aus den annotierten Klassen HScope und HXmlData aus Abb. 5.4 ergibt.(Vgl. [Wag11]).

5.2.3 BpelRuntimeContext und Aktivitäten

Abschließend zur detaillierten Betrachtung der Runtime-Komponenten wird nun die Funktionsweise der Aktivitäten (*ACTIVITY*), des Laufzeitkontext (*BpelRuntimeContext*) und deren Anbindung an die DAO-Schicht und an das *OModel* vorgestellt. Die Komponenten sind in Abbildung 5.6 dargestellt.

- **BpelRuntimeContext** und die Implementierung *BpelRuntimeContextImpl* stellen Methoden zur Verfügung, mit denen Variableninhalte gelesen (readVariable) und geschrieben (writeVariable) werden können. Diese greifen direkt auf die DAO-Schicht zu. Der *BpelRuntimeContext* ist somit das Bindeglied zwischen Runtime und DAO-Schicht. Außerdem werden WS-Aufrufe an die Kommunikationsinfrastruktur weitergeleitet und die Auswertungsmodule für Query-Sprachen (wie XPath, XQuery etc.) den Aktivitäten zur Verfügung gestellt.
- **ScopeFrame** implementiert die Funktionen der BPEL-Scopes, z.B das Auflösen einer Variable (resolve) entsprechend der Sichtbarkeit, die durch die im BPEL-Prozess definierten Scopes gegeben sind. Darüber hinaus stellt ScopeFrame Methoden für das Lesen (fetchVariableData) und Schreiben (writeVariable, commitChanges) von Variableninhalten bereit. ScopeFrame ist direkt mit seiner OModel-Repräsentation verbunden (oscope).
- **VariableInstance** ist eine Wrapperklasse für eine Variable aus dem OModel (OScope.Variable) und der ID des Scope, dem sie angehört.
- **ACTIVITY** ist die Oberklasse aller implementierten BPEL-Aktivitäten. Sie beinhaltet den ScopeFrame, in dem sie eingebettet ist, sowie die OModel-Repräsentation der Aktivität über ein Objekt der Klasse ActivityInfo. Sie stellt ebenfalls Methoden zum Lesen (fetchVariableData) und Schreiben (commitChanges) von Variableninhalten bereit. Desweiteren hat sie Zugriff auf das aktuelle *BpelRuntimeContext*-Objekt, welches für die laufende Instanz von Apache ODE gültig ist. Auf dieses kann über die Methode

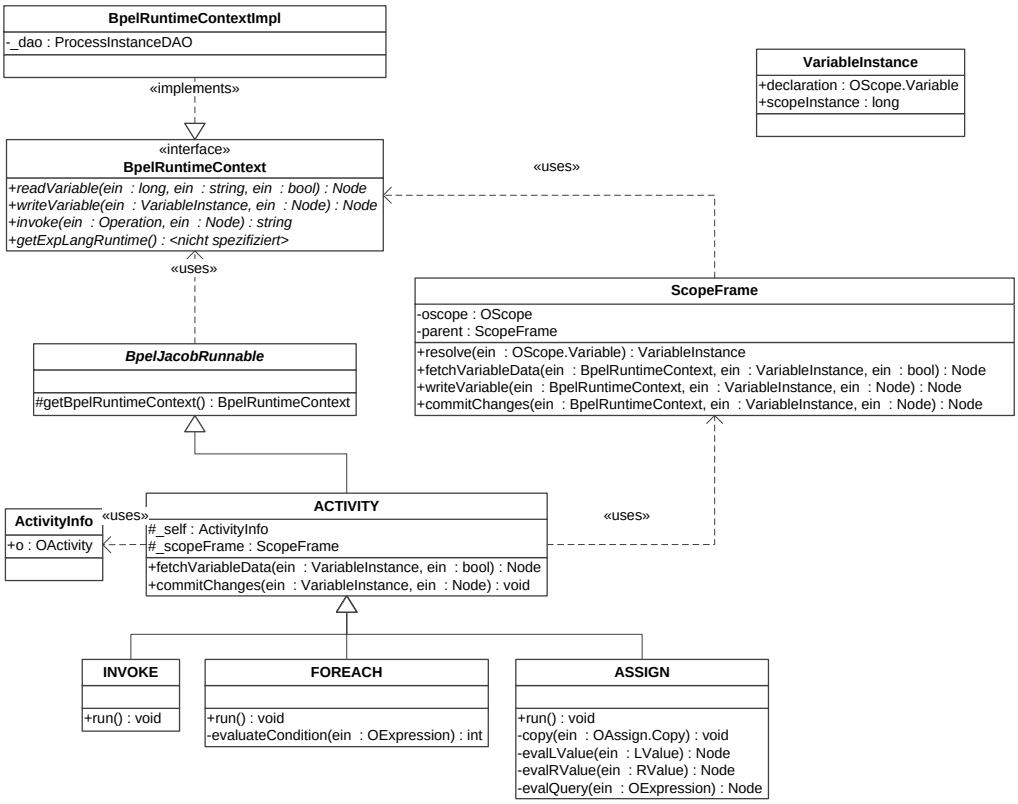

Abbildung 5.6: Ausschnitt der ODE-Laufzeitkomponenten als UML-Diagramm (Vgl. [Wag11]).

getBpelRuntimeContext(), welche von *BpelJacobRunnable* ererbt wurde, zugegriffen werden. Im Folgenden werden einige Aktivitäten vorgestellt, die im Zusammenhang dieser Arbeit relevant sind. Alle von *ACIVITY* abgeleiteten Aktivitäten müssen die Methode *run()* implementieren. Diese wird von der JacobVPU aufgerufen um die Aktivität zu starten.

- **INVOKE** realisiert die Logik eines WS-Aufrufs. Zuerst wird die Variable mit der Ausgangsnachricht gelesen, diese an die invoke-Methode des BpelRuntimeContext übergeben und anschließend die Antwortnachricht des WS in die dafür vorgesehene Variable geschrieben.
- **FOREACH** realisiert die Logik der BPEL-Foreach Schleife. Diese Schleife besitzt einen Start- und einen Endwert, über den ein Zähler läuft. Diese Werte werden über Query-Ausdrücke bestimmt (evaluateCondition).
- **ASSIGN** realisiert die BPEL-Assign Logik. Hierbei werden sequentiell alle Copy-Blöcke durchlaufen und jeweils die Variable der linken Seite aufgelöst (evalLValue) sowie

das Resultat des Ausdrucks oder der Inhalt der Variable der rechten Seite (evalRValue) und dieser Wert anschließend in die Variable der linken Seite gespeichert. Die Methode evalQuery wird verwendet um Query-Ausdrücke innerhalb von evalRValue auszuwerten.

- **PICK** (nicht abgebildet) vereint die BPEL-Aktivitäten Pick und Receive. Eine Receive-Aktivität wird dabei intern auf eine Pick-Aktivität mit einem einzelnen *onMessage*-Event abgebildet. Hierbei sind vor allem die instanzerzeugenden Aktivitäten (*createInstance=yes*) von Interesse, da jeder BPEL-Prozess davon mindestens eine enthält. Die Methode *initVariable()* extrahiert dabei aus der Eingangsnachricht den Nachrichteninhalt, der anschließend der Eingangsvariable zugewiesen wird.
- **REPLY** (nicht abgebildet) extrahiert den Variableninhalt der Ausgangsvariable, stellt eine Antwortnachricht zusammen und sendet diese an den aufrufenden Prozess.

Lesende und schreibende Zugriffe auf eine Variable innerhalb einer Aktivität finden grundsätzlich folgendermaßen statt:

1. Die Variable liegt als OScope.Variable vor und wird mit Hilfe von ScopeFrame.resolve aufgelöst und zusammen mit seiner Scope-ID in ein Objekt vom Typ VariableInstance umgewandelt.
2. Es wird auf die Schreib- und Lese-Methoden von ACTIVITY unter Verwendung von VariableInstance zugegriffen, diese geben den Aufruf an die Methoden von ScopeFrame weiter, die prinzipiell auch innerhalb der Aktivität direkt angesprochen werden können. Hierfür muss zusätzlich der BpelRuntimeContext übergeben werden.
3. ScopeFrame leitet die Anfrage an die Methoden zum Lesen und Schreiben von Variablen des BpelRuntimeContext weiter.
4. Der BpelRuntimeContext greift auf die konkreten Variableninhalte über die DAO-Schicht zu, überschreibt diese mit neuen Werten oder liefert den aktuellen Inhalt zurück.

Die Aktivitäten sind indirekt mit dem OModel über *ActivityInfo* und über den *ScopeFrame* verknüpft (vgl. Abschnitt 5.2.1). Die Anbindung an die DAO-Schicht erfolgt innerhalb der BpelRuntimeContextImpl über die *ProcessIntanceDAO* (vgl. Abschnitt 5.2.2).

5.3 BPEL-Compiler

An dieser Stelle wird der Aufbau und die Funktionsweise der Compiler-Komponente von Apache ODE ausschnittsweise vorgestellt. Sie ist später für die Umsetzung des Literal-Pushdowns von Bedeutung. Die dazu relevanten Teile des Compilers sind in Abbildung 5.7 als UML-Diagramm dargestellt und werden nun im einzelnen erläutert.

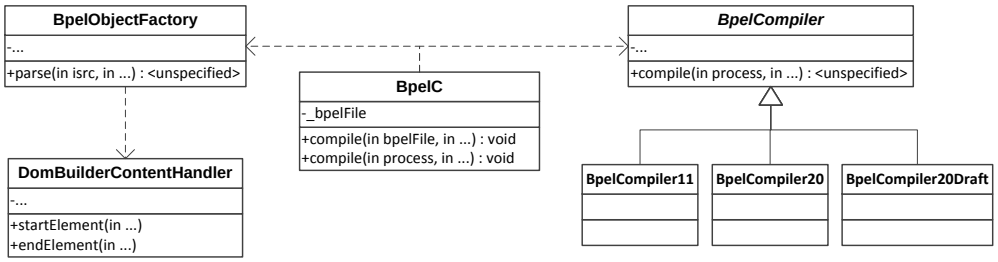

Abbildung 5.7: Ausschnitt relevanter Komponenten des ODE-Compilers als UML-Diagramm.

- **BpelC** ist die Klasse, in welche der für uns relevante Teil des Kompiliervorgangs stattfindet. Sie besitzt ein Attribut *_bpelFile* vom Typ *File*, das auf die Quelldatei des zu kompilierenden BPEL-Processes (.bpel) verweist. Diese Datei wird in einem ersten Kompilierschritt (*compile(bpelFile,...)*) zunächst in ein internes DOM-Objekt (*Process*) umgewandelt. In einem weiteren Schritt, dem eigentlichen Kompiliervorgang (*compile(process,...)*), wird dieses Prozessobjekt schließlich in das OModel für Prozesse (*OProcess*) überführt und das Resultat als .cbp-Datei (compiled bpel process) im Deploymentverzeichnis zu weiteren Verwendung durch ODE abgespeichert.
- **BpelObjectFactory** bietet Funktionen zur Umwandlung von XML-Daten in die von ODE benötigte, internen Objektrepräsentationen. Insbesondere beinhaltet sie die Methode *parse(irsc,...)* die eine Datei, die durch eine *InputSource*-Referenz spezifiziert wird, mit Hilfe des SAX-Parsers *XMLReader*[7] in ein ODE-Prozessobjekt umwandelt. Die Methode *parse(isrc,...)* wird von *BpelC* im ersten Kompilierschritt verwendet.
- **DomBuilderContentHandler** ist eine Erweiterung der Klasse DOMBuilder[8] und wird vom SAX-Parser XMLReader verwendet, um auf bestimmte Ereignisse während des Parsens zu reagieren. Beispielsweise können hier über die Methoden *startElement(...)* und *endElement(...)* zusätzliche Aktionen ausgeführt werden, sobald beim Parsen der Anfang bzw. das Ende eines Elementknotens erreicht wird.
- **BpelCompiler** ist eine abstrakte Klasse, die den Kompiliervorgang eines ODE-Prozessobjekts im Detail implementiert. Dabei wird aus einem DOM-Objekt (*Process*) mit Hilfe zahlreicher Kompiliermethoden das fertige OModel-Objekt (OProcess) für die weitere Verwendung in ODE erzeugt.
- **BpelCompiler11, 20 und 20Draft** sind Erweiterungen der abstrakten Klasse *BpelCompiler*, mit deren Hilfe der Kompiliervorgang für die unterschiedlichen BPEL-Spezifikationen realisiert wird.

[7] org.xml.sax.XMLReader
[8] org.apache.xml.utils.DOMBuilder

5.4 Änderungen durch den ODE-TI Prototyp

Zur Evaluierung der Konzepte aus ODE-TI wurde in einer vorangegangener Arbeit ein Prototyp mit Umsetzungen der Pushdown-Techniken auf Basis von Apache ODE in der Version 1.3.4 implementiert. In diesem Abschnitt wird der bei Beginn dieser Arbeit bestehende Prototyp kurz und auszugsweise vorgestellt, bevor im nächsten Kapitel auf die darauf aufbauende Implementierung der Konzepte aus dieser Arbeit detailliert eingegangen wird. Eine detailliertere und umfassendere Beschreibung der zuvor bestehenden Implementierung ist der Vorgängerarbeit [Wag11] zu entnehmen.

5.4.1 Datenbankschema

Um die Implementierungen der Pushdown-Techniken vorzubereiten, musste das Datenmodell bzw. das Tabellenschema aus Abbildung 5.5 geändert werden. Das BLOB-Feld *DATA* der Tabelle *BPEL_XML_DATA* wurde in ein Feld vom XML-Spaltentyp geändert, um die XML-Verarbeitung innerhalb des DBS zu ermöglichen. Dazu wurde die Hibernate-Annotierung für das Attribut *_data* der Klasse *HXmlData* aus Abb. 5.4 von BLOB auf XML geändert.

In der Originalversion von Apache ODE wird zur Speicherung von XML-Daten ein benutzerdefinierter Typ verwendet, der XML-Daten aus *HXmlData* in komprimierter Form in die Datenbank ablegt. Um eine direkte Verarbeitung der XML-Inhalte durch das DBS zu ermöglichen, wurde diese Komprimierung aufgehoben.

Um eine einheitliche Verarbeitung und Struktur der SQL- bzw. XPath-Ausdrücke zu erhalten, die im Prototyp generiert werden müssen, wurde die Unterscheidung zwischen einfachen Werten (SIMPLE_VALUE) und großen Werten (DATA) innerhalb von ODE aufgehoben. Alle XML-Daten werden fortan im Feld DATA gespeichert. Da XML Felder in einem DBS nur wohlgeformte XML Dokumente enthalten dürfen, wurde für XSD-einfache Typen ein Wrapperelement (<*temporary-simple-type-wrapper*/>) verwendet, um den Wert im XML-Feld *DATA* der Datenbank ablegen zu können. Das veränderte Tabellenschema ist in Abb. 5.8 dargestellt.

5.4.2 DAO-Schicht

Die DAO-Schicht verwaltet die Zugriffe auf die Datenbank. Dies geschieht im Allgemeinen über das DB-Middlewaresystem Hibernate, welches SQL Anfragen an das DBS kapselt und somit die Runtime prinzipiell unabhängig vom konkreten DBS und dem Datenschema macht. Die DAO-Schicht ist deshalb der Ort, an dem die SQL-Anfragen der Pushdown-Techniken generiert und das DBS geschickt werden müssen. Abb. 5.9 zeigt das modifizierte UML-Diagramm der DAO-Schicht. Unveränderte Komponenten sind grau eingefärbt, neue Klassen und Typen sind eingerahmt im rechten unteren Teil dargestellt. Alle anderen Komponenten (weiß) wurden in irgendeiner Weise modifiziert, um Pushdown-Techniken zu realisieren. Im Folgenden werden die einzelnen Änderungen kurz vorgestellt.

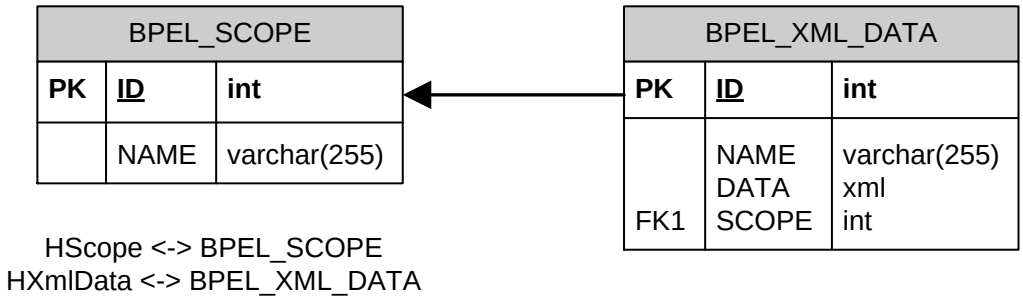

Abbildung 5.8: Durch den ODE-TI-Prototyp verändertes Tabellenschema (Vgl. [Wag11]).

- **HibernateDao** Die HibernateDao wird um die Methode *hibernateFlush()* erweitert. Diese wird in der Methode *XmlDataDaoImpl.set()* verwendet, um das Festschreiben eines Variablenwertes und somit dessen Persistenz zu erzwingen, was für die Realisierung der Pushdown-Techniken essentiell ist.
- **ScopeDAO und ScopeDaoImpl** In der *ScopeDAO* werden die Hauptmethoden der Pushdown-Techniken implementiert. Mit *dataAssignByContext* wird der Zuweisungs-Pushdown, mit Hilfe von *inDatabaseXPath* und insbesondere *inDatabaseExpression* der Ausdruckauswertungs-Pushdown realisiert. Die Methode *invokeWS* wird für die Realisierung des Webservice-Pushdown benötigt.
- **XmlDataDAO und XmlDataDaoImpl** Die XmlDataDAO wurde verändert, um einen veralteten Variableninhalt in *HXmlData* zu kennzeichnen (*setDetached*). Wird eine Variable durch einen Pushdown in der Datenbank verändert, muss bei einer späteren Verwendung dieser Variable innerhalb des Workflowkontext ihr veränderter Wert in den Hauptspeicher geladen werden.
- **HXmlData** Die Hibernate-Annotierung für *HXmlData* wurde von der komprimierten Bytedarstellung auf XML geändert. Das zusätzliche Attribut *_detached* ermöglicht die Kennzeichnung von veraltetem Inhalt. Dieses ist nur für die Workflowausführung interessant und wird daher nicht auf das Datenbankschema übertragen.
- **VariableContext** Um z.B. die Logik für den Zuweisungs-Pushdown auf die DAO-Schicht übertragen zu können, wird diese Wrapper-Klasse eingeführt, die alle dafür benötigten Informationen einer Variable zusammenfasst. Ausdrücke werden dabei als Pseudovariablen ohne *ID* übergegeben. Insbesondere wird dabei im Attribut *ID* der Primärschlüssel aus der Tabelle *BPEL_XML_DATA* der jeweiligen Variable sowie im Fall von Ausdrücken in *exprContext* Referenzen auf enthaltene Variablen gespeichert. In *varType* wird eine durch den Prototyp eingeführte Typunterscheidung von Variablen festgehalten, die zur Umsetzung verschiedener Pushdown-Techniken verwendet wird.
- **VarType** *VarType* ist eine Aufzählung möglicher Typen, die bei einer Zuweisung auftreten können und wird in *VariableContext* verwendet. Die Typen entsprechen dabei

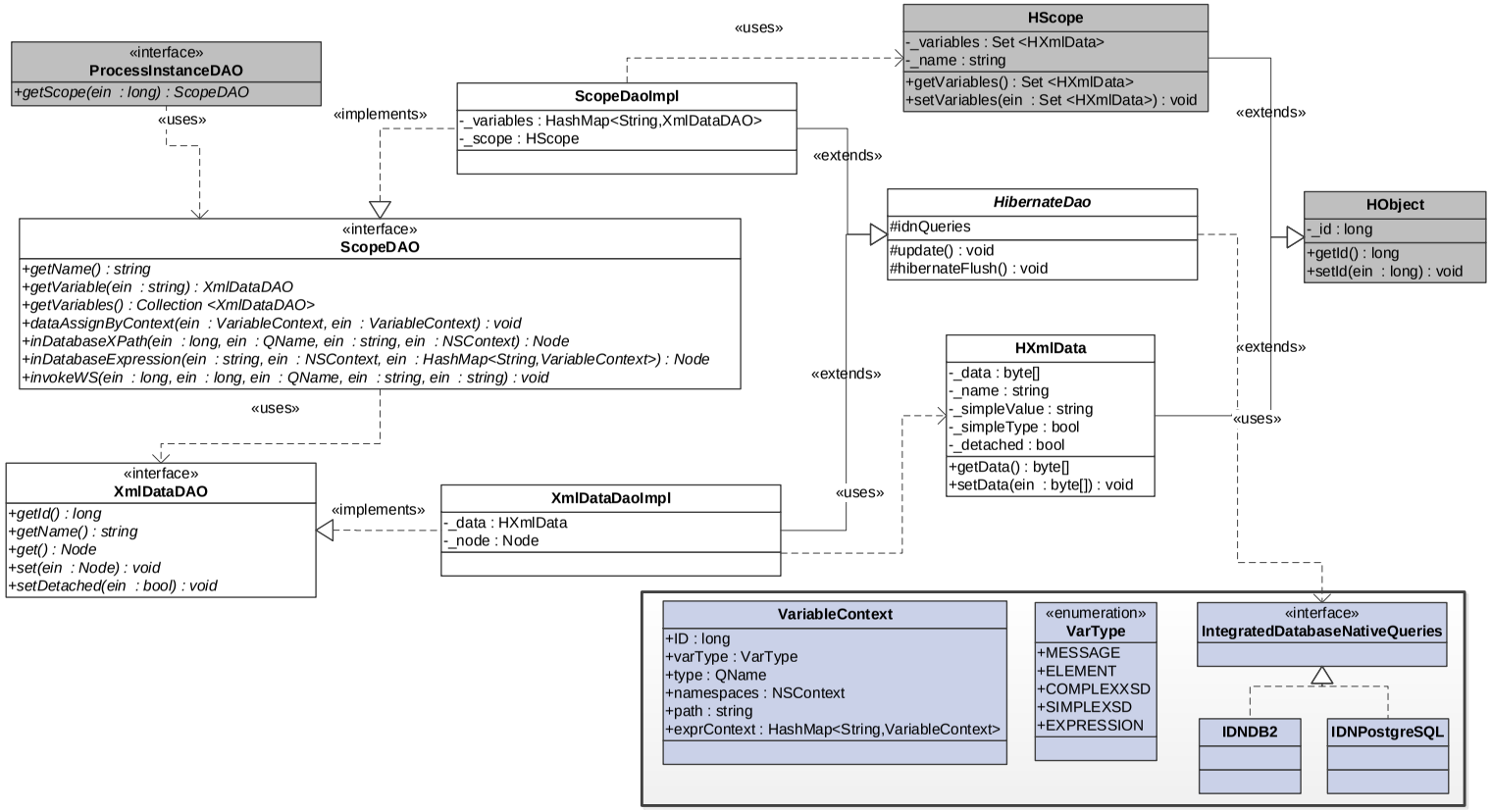

Abbildung 5.9: UML-Diagramm der modifizierten DAO-Schicht. Die eingerahmten Komponenten wurden hinzugefügt, die weißen wurden im Vergleich zu Abb. 5.4 verändert. Grau eingefärbte Komponenten wurden nicht modifiziert (Vgl. [Wag11]).

den Typen aus dem OModel von ODE (*OMessageVarType*, *OElementVarType*,...). Die Typen COMPLEXXSD und SIMPLEXSD entsprechen dem Typ *OXsdTypeVarType* mit den Eigenschaften simple=false bzw. simple=true.

- **IntegratedDatabaseNativeQueries** Diese Schnittstelle kapselt alle SQL-Anfragen und Fragmente, die für das Zusammenstellen der Anfragen an das DBS benötigt werden. Diese Schnittstelle wurde erstellt, um verschiedene Datenbanksysteme anbinden zu können, da die Middleware Hibernate keine eigenen Möglichkeiten zur internen XML-Verarbeitung bietet. Diese Schnittstelle wird innerhalb von *ScopeDaoImpl* verwendet, um die konkreten SQL-Anfragen an das DBS zu generieren. Beim ODE-TI Prototyp wurden einige Techniken neben DB2 auch für das DBS PostgreSQL[9] implementiert. Für diese Arbeit wird jedoch nur die DB2-Implementierung betrachtet, da sich PostgreSQL nicht als geeignet erwiesen hat, um alle Pushdown-Konzepte zu realisieren [Wag11].

Hauptmethoden von ScopeDAO

Hier weden kurz die Methoden vorgestellt, die die Anweisungen der umgesetzten Pushdown-Konzepte an das DBS weitergeben.

- **invokwWS** ruft eine benutzerdefinierte Funktion (*UDF*) des DBS auf, die einen Webservice-Aufruf erzeugt. Sie besitzt die zwei Parameter *inputVar* und *outputVar*, die für die Primärschlüssel der Variablen stehen, die als Eingangs- bzw. Ergebnisvariable des Webservice-Aufrufs agieren. Die übrigen Parameter der Funktion dienen zur Identifikation der Webservice-Operation.
- *inDatabaseXpath* stellt SQL-Anfragen für synchronen Pushdown innerhalb von Zuweisungen zusammen und liefert das Ergebnis als (XML-) Node. Im Gegensatz zu *inDatabaseExpression* kann nur auf eine Variable referenziert werden, die Methode stellt damit einen Sonderfall des Ausdruckauswertungs-Pushdowns dar.
- **inDatabaseExpression** stellt die SQL-Abfrage für den synchronen Pushdown eines allgemeinen Ausdrucks zusammen und liefert das Ergebnis als (XML-) Node. In einem Parameter *xPath* wird der auszuwertende Ausdruck gespeichert, während ein Parameter *exprContext* alle im Ausdruck vorkommenden Variablen als Hashtabelle mehrerer *VariableContext*-Objekte speichert.
- **dataAssignByContext** ist für den (asynchronen) Zuweisungs-Pushdown verantwortlich. Diese Methode implementiert die veränderte Zuweisungslogik der ASSIGN-Aktivität innerhalb der DAO-Schicht. Als Parameter übernimmt sie den *VariableContext* der linken sowie die rechten Seite einer Zuwesiung (*lContext*, *rContext*). Sie sorgt dafür, dass innerhalb des DBS die Zuweisung der rechten Seite an die linke Seite erfolgt. In *rContext* kann dabei entsprechend der Definition von *VariableContext* entweder eine Variable oder ein Ausdruck enthalten sein. Die endgültige Durchführung eines Zuweisungs-Pushdowns stellt sich unter Zuhilfenahme von zahlreichen weiteren

[9]PostgreSQL - http://www.postgresql.org/

Methoden zusammen. Dabei werden z.B. Ausdruckauswertungen nach dem selben Prinzip wie bei *inDatabaseExpression* durchgeführt, bevor das Ergebnis der linken Seite Zugewiesen wird.

5.4.3 Runtime-Schicht

Die in der DAO-Schicht vorgestellten Veränderungen reichen nicht aus, um die Pushdown-Techniken bei der Workflowausführung zu verwenden. Um dies zu bewerkstelligen müssen noch für alle BPEL-Aktivitäten, die nach einem Pushdown-Konzept ausgeführt werden sollen, die Ausführungslogik so verändert werden, dass die neu implementierten Methoden der DAO-Schicht aufgerufen werden. Die Hauptarbeit in der Runtime-Schicht besteht nun darin, Informationen zu Variablen und Ausdrücken aus dem OModel zu extrahieren und in geeigneter Form (direkt oder durch die Wrapperklasse *VariableContext*) an die DAO-Schicht zu übergeben, die dann die entsprechenden Operationen im DBS durchführt. Abb. 5.10 zeigt das modifizierte UML-Diagramm der Runtime-Schicht. Unveränderte Komponenten sind grau eingefärbt, weiße Komponenten wurden in irgendeiner Weise modifiziert, um Pushdown-Techniken zu realisieren. Im Folgenden werden die einzelnen Änderungen kurz vorgestellt.

- **BpelRuntimeContext und BpelRuntimeContextImpl** stellen vier Pushdown-Methoden (*inDatabaseAssign*, *inDatabaseExpressionAssign*, *inDatabaseXPath*, *inDatabaseXPath-Expression*) bereit, die ausschließlich von *ScopeFrame* aus aufgerufen werden. Hier werden aufgelöste Variablen und Ausdrücke übernommen und die dazugehörigen für die DAO-Schicht benötigten Informationen extrahiert. Diese Informationen werden ggf. in Objekte der Wrapper-Klasse *VariableContext* überführt. Die Methode *inDatabaseInvoke* wird direkt von der *INVOKE*-Aktivität aufgerufen.
- **ScopeFrame** stellt drei Methoden (*inDatabase**) bereit, die von den einzelnen BPEL-Aktivitäten verwendet werden können, um einen Pushdown einzuleiten. Der Aufruf von *inDatabaseAssign* führt zu einem asynchronen Zuweisungs-Pushdown. Der Aufruf von *inDatabase-XPathExpression* bewirkt einen allgemeinen synchronen Ausdruckauswertungs-Pushdown, der auch für die Evaluierung von Bedingungen in Kontrollstrukturen und Schleifen genutzt werden kann.
- **ACTIVITY und Unterklassen** leiten eine Pushdown-Operation ein, indem sie die neuen Methoden von *ScopeFrame* aufrufen. Eine Ausnahme bildet die Aktivität INVOKE, die direkt auf Methoden von *BpelRuntimeContext* zugreift.

Durch diese Aufteilung der Methoden zur Realisierung von Pushdowns auf die verschiedenen Runtime-Komponenten können die implementierten Methoden von weiteren Aktivitäten und eventuell auch für die Umsetzung anderer Konzepte wiederverwendet werden.

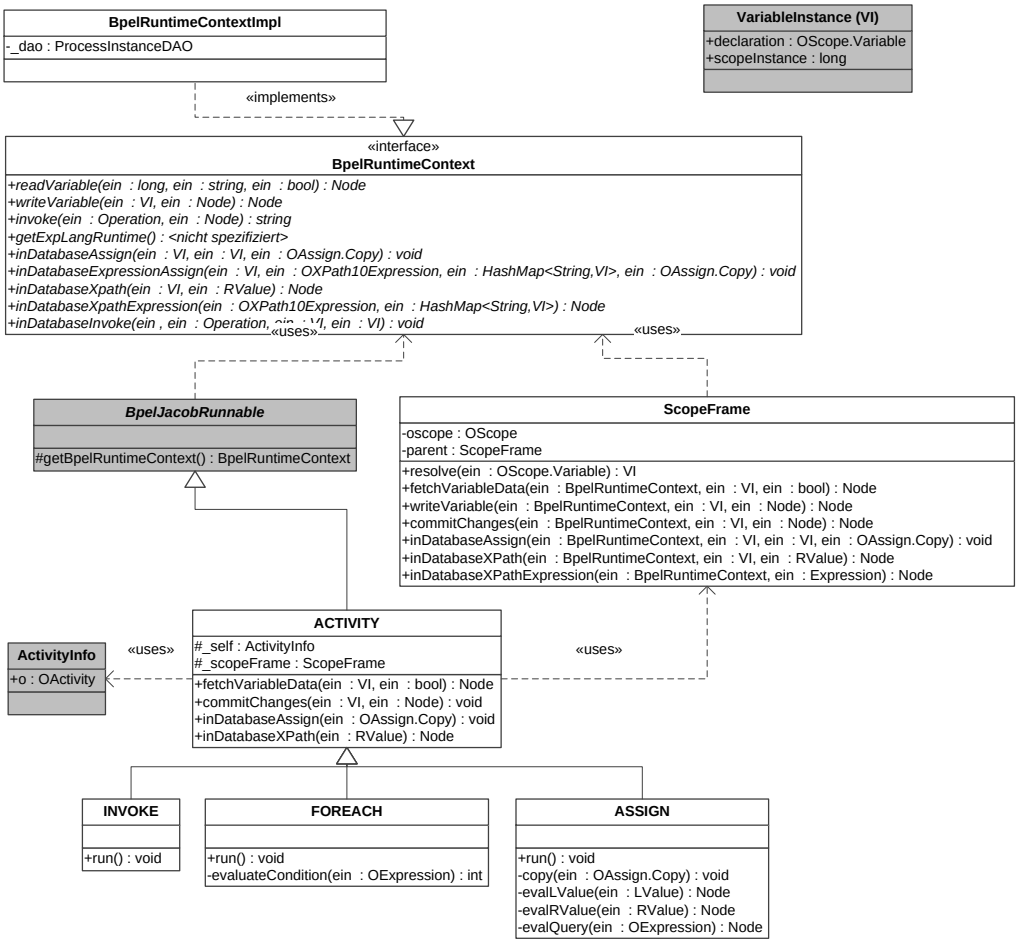

Abbildung 5.10: UML-Diagramm der modifizierten Runtime-Schicht. Weiße Komponenten wurden im Vergleich zu Abb. 5.6 verändert, grau eingefärbte Komponenten wurden nicht modifiziert (Vgl. [Wag11]).

5.4.4 Funktionalität des Prototyps

Das Konzepte des Zuweisungs-Pushdowns sowie des Ausdruckauswertungs-Pushdowns wurden für XPath-Ausdrücke in IBM DB2 vollständig implementiert. Der Webservice-Pushdown wurde rudimentär implementiert, hier fehlt noch eine Logik zur Fehlerbehandlung und Fehlerweitergabe an Apache ODE aus dem DBS heraus. Für die Evaluation der Laufzeit des Webservice-Pushdowns ist die bestehende Implementierung jedoch zureichend. Generell wird im Falle eines Fehlers, z.B. bei Auftreten einer Ausnahme in einer der neuen Methoden, die ursprüngliche Logik der entsprechenden Aktivität ausgeführt.

Im einzelnen wurden generelle Pushdown-Funktionen für die Aktivitäten *ASSIGN*, *INVOKE*, *FOREACH*, *WHILE*, *REPEATUNTIL*, *IF*, *ONALARM*, *WAIT* und für *TransitionConditions* umgesetzt. Gegenstand dieser Arbeit sind die nicht aufgeführten Aktivitäten *RECEIVE*, *PICK*, *REPLY* und *ON_EVENT*, sowie die Erweiterung von *ASSIGN* um den *Literal-Pushdown*. Außerdem sollen nach Möglichkeit alle Pushdown-Konzepte neben *XPath* auch *XQuery* bzw. *pureXML* als Ausdrucksprache im Zusammenspiel mit DB2 unterstützen.

6 Implementierung der konzeptionellen Erweiterungen

In diesem Kapitel werden Implementierungen der in Kapitel 4 eingeführten, neuen Pushdown-Konzepte auf der Grundlage des ODE-TI-Prototyps vorgestellt. Damit wird die in Abschnitt 5.4 vorgestellte Implementierung erweitert. In den folgenden Unterkapiteln werden die durch diese Arbeit entwickelten Änderungen an betroffenen Komponenten von ODE-TI im Detail beschrieben.

Bevor die bestehende Implementierung verändert wurde, wurden alle durch den ODE-TI-Prototyp vorgenommenen Änderungen an Apache ODE auf die zum Zeitpunkt dieser Arbeit aktuellste ODE Version 1.3.5 portiert, um bei der Weiterentwicklung der Pushdown-Implementierungen von den Bugfixes und Verbesserungen[1] der neueren Version zu profitieren.

Nach jedem erfolgreich ausgeführten Pushdown wird die interne Workflowvariable, die ohne den Pushdown verändert worden wäre, mittels der Metode *setDetached* ihres DAO-Objekts als veraltet gekennzeichnet. Vor jeder Verwendung von internen Workflowvariablen wird diese Eigenschaft durch die Implementierungen des ODE-TI-Prototyps abgefragt und bei Bedarf die Variable mit ihrem Wert aus der Datenbank synchronisiert. Dieser Fall sollte nach Fertigstellung aller Pushdown-Implementierungen idealerweise nicht mehr auftreten, da dann alle Variablenoperationen ausnahmlos im DBS stattfinden sollten. Die hier vorgestellten Erweiterungen tragen zum Teil dazu bei.

6.1 Literal-Pushdown

Zur Implementierung des Literal-Pushdowns müssen Veränderungen zur Realisierung des schreibenden sowie des lesenden Teils (vgl. Abschnitt 4.1) vorgenommen werden. Der lesende Teil, bei dem die Literale zur Laufzeit abgerufen werden, muss innerhalb der Zuweisungslogik in der Runtime-Schicht von ODE implementiert werden. Der schreibende Teil des Pushdowns wird nach Variante (b) von Abbildung 4.1 in den BPEL-Compiler von ODE implementiert. Außerdem muss das Datenbankschema verändert werden, um die zusätzlichen Literalwerte aufzunehmen. Die genauen Änderungen an den Komponenten werden nun vorgestellt.

[1] https://issues.apache.org/jira/secure/ReleaseNote.jspa?projectId=12310270&version=12314243

BPEL_LITERALS		
PK	**ID**	**bigint**
	DATA	xml

Abbildung 6.1: Tabellenschema für die Speicherung von Lieralwerten.

DOMBuilderContentHandler
-literalcount : int -currentID : long -con : Connection
+startElement(in localName : String, in atts : Attributes, in ...) +endElement(in localName : String, in ...)

Abbildung 6.2: Veränderte Version des *DOMBuilderContentHandler* in UML-Notation. (vgl. Abb. 5.7)

6.1.1 Datenbankschema

Um Literalwerte in der Datenbank speichern zu können, muss dort zunächst ein geeigneter Ort gefunden werden. Dazu wird eine neue Tabelle mit dem Namen *BPEL_LITERALS* erzeugt. Das Schema der neuen Tabelle ist in Abbildung 6.1 dargestellt.

Ein Literalwert wird in das XML-Feld *DATA* einer Zeile der Tabelle *BPEL_LITERALS* abgelegt. Die Spalte *ID* ist Primärschlüssel der Tabelle und dient der eindeutigen Identifikation eines gespeicherten Lietrals.

6.1.2 BPEL-Compiler

Die Architektur des BPEL-Compilers wurde in Abschnitt 5.3 vorgestellt. Seine Komponenten sollen nun so geändert werden, dass Lieralwerte während des Kompiliervorgangs eines BPEL-Prozesses aus der Quelldatei in das DBS geschrieben werden. Um dies zu realisiren, wurde lediglich die Klasse *DOMBuilderContentHandler* modifiziert. Die wichtigsten Bestandteile der daraus resultierenden Klasse sind in Abbildung 6.2 in UML-Notation in Anlehnung an Abbildung 5.7 dargestellt.

Die Methode *startElement* wird immer dann aufgerufen, wenn beim zeilenweisen Parsen der Quelldatei der Beginn eines Elementknotens festgestellt wird. Dabei wird der Name

des beginnenden Elements im Parameter *localName* übergeben. Über den Parameter *atts* werden zudem die Attribute des beginnenden Elements als *Attributes*[2]-Objekt übergeben. Die Attribute des Elementknotens stehen also innerhalb der Methode *startElement* zur Verfügung und können hier während des Parsevorgangs manipuliert werden.

In der Variablen *literalcount* wird die Anzahl gefundener Literale beim Parsen der Quelldatei festgehalten. Diese wird zu Beginn des Parsevorgangs mit dem Wert 0 initialisiert und in der Methode *startElement* immer dann um den Wert 1 erhöht, wenn im Parameter *localName* der Wert „literal“ übergeben wird. Innerhalb von BPEL-Zuweisungen ist das Umschließen von Literalwerten mit dem Wurzelelement <*literal* ...>...</*literal*> durch die BPEL-Spezifikation vorgeschrieben. Während der weiteren Ausführung der Methode *startElement* wird die Variable *literalcount* als Identifikationsnummer des aktuellen Literalwerts verwendet. Unter Zuhilfenahme der Systemzeit wird daraus zu jedem Literalwert ein eindeutiger *ID*-Wert, der später als Primärschlüssel in der Datenbanktabelle *BPEL_LITERALS* dient, erzeugt und in der Variable *currentID* festgehalten.

Die Methode *endElement* wird immer dann aufgerufen, wenn beim Parsen das Ende eines Elementknotens festgestellt wird. Zu diesem Zeitpunkt ist der endende Elementknoten vollständig geparst worden und es kann auf seinen kompletten Inhalt zugegriffen werden. Jedem Aufruf der Methode *endElement* geht immer ein Aufruf die Methode *startElement* voraus. Handelt es sich bei dem Element um ein Literal, wird immer der gesamte Inhalt als Blattknoten aufgefasst, sodass zwischen dem Aufruf von *startElement* und *endElement* eines Literalknotens kein weiterer Aufruf der beiden Methoden stattfindet. Insbesondere beinhaltet dann die Variable *currentID* bei jedem Aufruf von *endElement* noch den Wert, der durch *startElement* des selben Elements gesetzt wurde. Mit Hilfe der *currentID* und des nun vollständig bekannten Inhalts des Literalknotens wird mittels der Datenbankverbindung *con* sowie entsprechender Datenbankbefehle der Literalwert in die Tabelle *BPEL_LITERALS* hinzugefügt und steht damit im DBS zur Verfügung.

Um einen in der Datenbank gespeicherten Literalwert zur Laufzeit wiederzufinden muss im BPEL-Prozessmodell noch ein Verweis darauf hinterlassen werden. Zu diesem Zweck wurde in der Methode *startElement* eine eindeutige ID für den Literalwert erzeugt. Dieser Wert wird noch im selben Aufruf der Methode mit Hilfe des Attributobjekts *atts* des Literalknotens als ein zusätzliches Attribut gespeichert. Damit kann jedem Literal über das neue Attribut *db2literal* im Wurzelelement sein Inhalt in der Datenbank zugeordnet werden. Um Fehler beim Literal-Pushdown zur Laufzeit leichter kompensieren zu können, wird durch den schreibenden Teil die konventionelle Ausführungslogik und Datenstruktur, bis auf das hinzufügen des ID-Attributs, nicht beeinflusst. Die Datenstruktur kann jedoch angepasst werden, sodass nur noch das ID-Attribut eines Literals vorgehalten wird, was zur Entlastung des Hauptspeichers während der Ausführung beitragen kann.

[2] org.xml.sax.Attributes

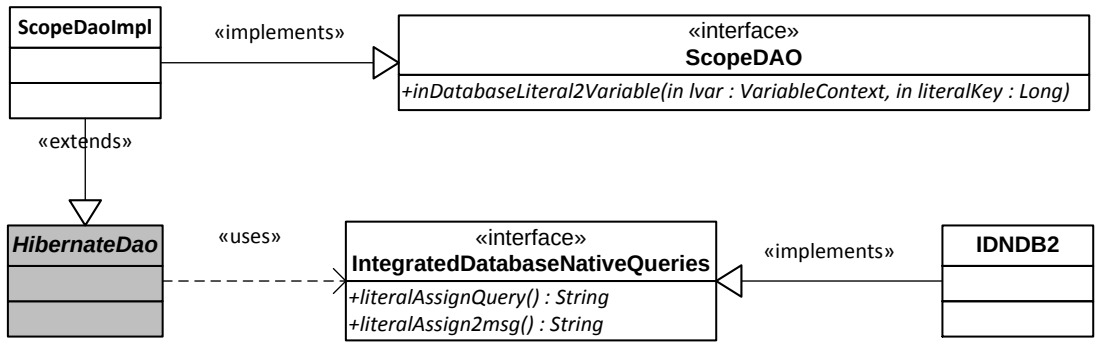

Abbildung 6.3: Am Literal-Pushdown beteiligte Komponenten der DAO-Schicht in UML-Notation. Unveränderte Klassen sind grau eingefärbt (vgl. Abb. 5.9).

```
1 UPDATE BPEL_XML_DATA
2 SET DATA = XMLDOCUMENT(
3             XMLQUERY('$literaldata/*[local-name()="literal"]/*'
4               PASSING (SELECT DATA
5                        FROM BPEL_LITERALS
6                        WHERE ID = :literalid)
7               AS "literaldata"))
8 WHERE ID = :varid
```

Listing 6.1: SQL-Befehl für die allgemeine Zuweisung eines Literalwerts in eine Variable

6.1.3 DAO-Schicht

Um den lesenden Teil des Literal-Pushdowns innerhalb des Zuweisungs-Pushdowns zu ermöglichen, müssen entsprechende Funktionen und Datenbankbefehle in der DAO-Schicht hinzugefügt werden (vgl. Abschnitt 5.4.2, Abb. 5.9). Die daran beteiligten Komponenten sind in Abbildung 6.3 dargestellt.

Die Hauptmethode des lesenden Literal-Pushdown in der DAO-Schicht ist *inDatabaseLiteral2Variable*. Sie wird in der Schnittstelle *ScopeDAO* definiert und in *ScopeDaoImpl* implementiert. Hier wird anhand der ID eines Literalwertes seine Zuweisung an die Variable, die durch *lvar* spezifiziert wird, durchgeführt. Der SQL-Befehl für die allgemeine Zuweisung von Literalwerten in Variablen ist Listing 6.1 zu entnehmen.

Der SQL-Befehl wird durch die Methode *literalAssignQuery* aus der *IDNDB2*-Implementierung der Schnittstelle *IntegratedDatabaseNativeQueries* bereitgestellt. Durch den XPath-Ausdruck *$literaldata/*[local-name()="literal"]/** wird der Literalwert ohne das Wurzelelement <*literal*> selektiert. Die geschachtelte *SELECT*-Anweisung liefert den Wert des Literals aus der Literaltabelle, dieser wird in die Variable *literaldata* des XPath-Ausdrucks kopiert. Die Platzhalter *:literalid* und *:varid* werden in der Methode *inDatabaseLiteral2Variable* durch die entsprechenden ID-Werte des Literals bzw. der Variable ersetzt, bevor der Befehl

```
UPDATE BPEL_XML_DATA
SET DATA = XMLQUERY('COPY $new := $DATA
                    MODIFY DO REPLACE $new:lValuePath
                    WITH $literaldata/*[local-name()="literal"]/*
                    RETURN $new'
                    PASSING (SELECT DATA
                             FROM BPEL_LITERALS
                             WHERE ID = :literalid)
                      AS "literaldata")
WHERE ID = :varid
```

Listing 6.2: SQL-Befehl für die Zuweisung eines Literalwerts in eine Variable vom Typ Nachricht mit spezifiziertem <*part*>-Teil.

an das DBS gesendet wird. Der SQL-Befehl hinter der Methode *literalAssign2msg* realisiert die Zuweisung für Variablen vom Typ *Nachricht*. Eine solche Variable kann aus mehreren Teilen (*parts*) bestehen, die nach dem BPEL-Standard individuell Ziel einer Zuweisung sein können, sodass nicht der gesamte Variableninhalt ersetzt wird. Der entsprechende SQL-Befehl ist ist ebenfalls in der Klasse *IDNDB2* implementiert und Listing 6.2 zu entnehmen.

Bei den Befehlen innerhalb der *XMLQUERY*-Funktion in Listing 6.2 handelt es sich um die XQuery-Erweiterung durch pureXML zur Manipulation von XML-Dokumenten (vgl. Abschnitt 2.5). Die Anweisung *MODIFY DO REPLACE* bewirkt, dass der Inhalt des XML-Dokuments an der durch einen XPath-Ausdruck spezifizierten Stelle mit dem Wert der XPath-Auswertung hinter der *WITH*-Anweisung ersetzt wird. Dies wird benötigt, da hier die Tabelle *BPEL_XML_DATA* innerhalb eines XML-Datenfelds verändert wird, und nicht mehr auf Zeilenebene. Der Platzhalter *:lValuePath* wird vor der Ausführung durch einen XPath-Ausdruck ersetzt, der dem in der Zweisung spezifizierten Nachrichtenteil (*part*) entspricht. Die Platzhalter *:literalid* und *:varkey* werden, wie bei allgemeinen Literalzuweisungen, durch die Primärschlüssel des verwendeten Literals bzw. der zu verändernden Variable ersetzt.

6.1.4 Runtime-Schicht

Wie bereits bei den Pushdown-Techniken des ursprünglichen ODE-TI-Prototyp muss auch für den Literal-Pushdown die Zuweisungslogik in der Runtime-Schicht so geändert werden, dass die neu eingeführten Methoden der DAO-Schicht aufgerufen werden, um so die Literalzuweisung innerhalb der Datenbank zu realisieren. Die daran beteiligten Komponenten sind in Abbildung 6.4 dargestellt.

Eine Zuweisung wird innerhalb einer *ASSIGN*-Aktivität mit der Methode *copy* eingeleitet. Ist die rechte Seite der Zuweisung ein Literalwert, so wird im bisherigen Prototyp der Zuweisungs-Pushdown abgebrochen und es wird mit einer konventionellen (original ODE) Literalzuweisung innerhalb des Workflows fortgefahren. Mit dem Literal-Pushdown wird nun der Zuweisungs-Pushdown auch beim Auftreten von Literalwerten weitergeführt. Zunächst wird das interne Objekt *ocopy*, das alle modellierten Informationen eines Zuweisungsblocks enthält, an die Methode *inDatabaseAssign* seiner Superklasse *ACTIVITY* weitergeleitet. Hier wird unter anderem die Referenz der Zielvariable aufgelöst und als

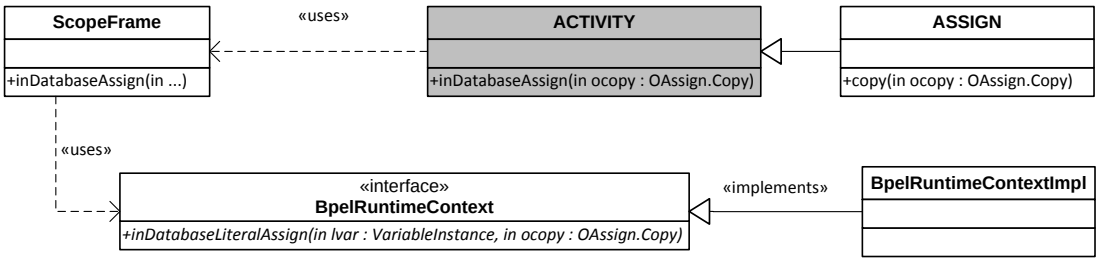

Abbildung 6.4: Am Literal-Pushdown beteiligte Komponenten der Runtime-Schicht in UML-Notation. Unveränderte Klassen sind grau eingefärbt (vgl. Abb. 5.10).

VariableInstance-Objekt an den *Scopeframe* weitergegeben. In der dortigen Methode *inDatabaseAssign* wird die rechte Seite des Zuweisungsblocks genauer analysiert und je nach Art ihrer Ausprägung unterschiedliche Zuweisungsmethoden aufgerufen. An dieser Stelle setzt der lesende Teil des Literal-Pushdown ein und ruft beim Auftreten eines Literalwerts die neue Methode *inDatabaseLiteralAssign* der DAO-Schicht auf, die die Literalzuweisung innerhalb des DBS durchführt (vgl. Abschnitt 6.1.3). Am Ende wird die interne Workflowvariable, die aufgrund der Pushdownlogik nicht verändert wurde, als veraltet gekennzeichnet (*setDetached*).

6.2 Nachrichten-Pushdown

Der Nachrichten-Pushdown wird für die Zuweisung von Nachrichten in Variablen nach der in Abschnitt 4.2 vorgestellten Variante (b), also ohne Datenaustausch zwischen Workflowinstanz und DBS, implementiert. Die umgekehrte Zuweisung von Variablen in Nachrichten wird hier nicht im DBS umgesetzt, stattdessen wird dazu die verwendete Workflowvariable vorher mit der Datenbank synchronisiert und auf konventionellem Wege verarbeitet. Die Gründe hierfür wurden bereits in Abschnitt 4.2 erörtert.

6.2.1 Datenbankschema

Zur Realisierung von Zuweisungen zwischen Nachrichten- und Variableninhalten muss zunächst deren Verarbeitung durch das DBS ermöglicht werden. Die Tabelle *BPEL_MESSAGE*, in der Nachrichten durch die DAO-Schicht gespeichert werden, hält den Dateninhalt einer Nachricht als komprimiertes BLOB-Objekt vor. Dieses lässt sich nicht ohne weiteres einem XML-Feld aus *BPEL_XML_DATA* zuweisen. Daher wurde analog zu den Änderungen an *HXMLData* nun auch die Hibernate-Annotierung der Nachrichtenklasse *HMessage* dahingehend geändert, dass die Daten als unkomprimierter XML-Wert gespeichert werden (Listing 6.3). Das daraus resultierende (Teil-)Tabellenschema ist in Abbildung 6.5 abgebildet.

BPEL_MESSAGE		
PK	**ID**	**bigint**
	MESSAGE_DATA	xml

Abbildung 6.5: Geändertes Hibernate-(Teil-)Tabellenschema für die Persistenz von Nachrichten.

```
/**
 * @hibernate.class table="BPEL_MESSAGE"
 */
public class HMessage extends HObject {

  private byte[] _data;
  ...

  /**
     * @hibernate.property type="byte[]"
     * @hibernate.column name="MESSAGE_DATA" sql-type="XML"
     */
    public byte[] getMessageData() {
        return _data;
    }
  ...
}
```

Listing 6.3: Hibernate-Annotation der Klasse *HMessage* zur Speicherung des Nachrichteninhalts als XML-Wert (vgl. Listing 5.1).

Durch das veränderte Tabellenschema wird eine direkte Manipulation zwischen den Spalten *BPEL_MESSAGE.MESSAGE_DATA* und BPEL_XML_DATA.DATA innerhalb des DBS unter Verwendung von SQL bzw. pureXML ermöglicht.

6.2.2 DAO-Schicht

In der DAO-Schicht müssen zur Realisierung des Nachrichten-Pushdowns ebenfalls entsprechende Funktionen und Datenbankbefehle hinzugefügt werden. Die daran beteiligten Komponenten sind in Abbildung 6.6 dargestellt.

Die Hauptmethode des Nachrichten-Pushdowns in der DAO-Schicht ist *inDatabaseMessage2Variable*. Sie wird in der Schnittstelle *MessageDAO* definiert und in *MessageDaoImpl* implementiert. Hier erfolgt die Zuweisung einer Nachricht, deren Primärschlüssel aus dem der Klasse zugehörigen *HMessage*-Objekt ausgelesen wird, in die Variable, deren Primärschlüssel durch den Parameter *varID* übergeben wird. Der SQL-Befehl für die Zuweisung einer Nachricht in eine Variable ist in der Klasse *IDNDB2* implementiert und aus Listing 6.4

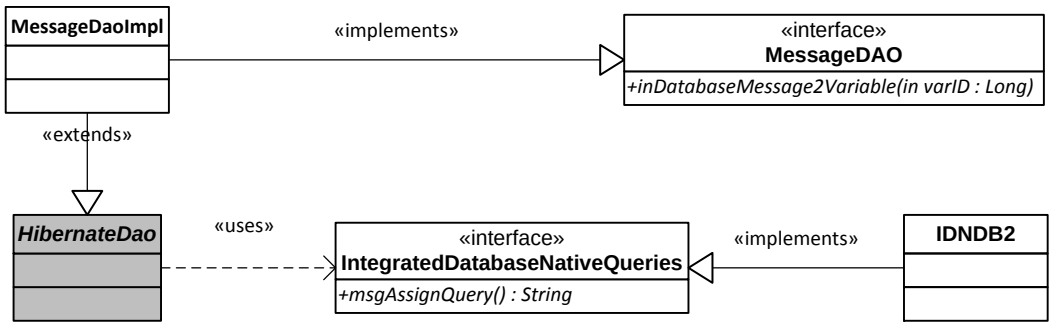

Abbildung 6.6: Am Nachrichten-Pushdown beteiligte Komponenten der DAO-Schicht in UML-Notation. Unveränderte Klassen sind grau eingefärbt (vgl. Abb. 5.10).

```
1 UPDATE BPEL_XML_DATA
2 SET DATA = (SELECT MESSAGE_DATA
3             FROM BPEL_MESSAGE
4             WHERE ID = :mexID)
5 WHERE ID = :varID
```

Listing 6.4: SQL-Befehl für die Zuweisung des Inhalts einer Nachricht in eine Variable.

zu entnehmen. Die Platzhalter *:mexID* und *:varID* werden vor der Ausführung durch den Primärschlüssel der verwendeten Nachricht bzw der zu verändernden Variable ersetzt.

6.2.3 Runtime-Schicht

Auch beim Nachrichten-Pushdown müssen die neuen Funktionen der DAO-Schicht durch Komponenten der Runtime-Schicht aufgerufen werden. Die daran beteiligten Komponenten sind in Abbildung 6.7 dargestellt.

Der Nachrichten-Pushdown wird innerhalb der Klasse *PICK*- bzw. *EH_EVENT* eingeleitet. Die Klasse *PICK* implementiert die BPEL-Aktivitäten *RECEIVE* und *PICK*, während die Klasse *EH_EVENT* einen *OnEvent-Handler* realisiert. Beide Aktivitäten reagieren auf einen Nachrichteneingang in ODE (vgl. Abschnitt 5.2.3). Die Methode *onRequestRcvd* der Klasse *PICK* wird aufgerufen, sobald eine Eingangsnachricht empfangen wurde. Darin wird die Hilfsfunktion *initVariable* mit den Parametern *mexID*, der die Eingangsnachricht identifiziert, und *onMessage*, indem unter anderem die Zielvariable spezifiziert ist, aufgerufen. Hier erfolgt die Initialisierung der Zielvariablen mit dem Nachrichteninhalt. An dieser Stelle ersetzt der Nachrichten-Pushdown die konventionelle Ausführungslogik, indem die neue Methode *inDatabaseReceiveMessage2Variable* des *BpelRuntimeContext* aufgerufen wird. Diese Methode übernimmt als Parameter wiederum die *mexID* der Nachricht sowie die aus dem *onMessage-Objekt* aufgelöste Variable als *VariableInstance* (Parameter *var*). Hier wird das zur

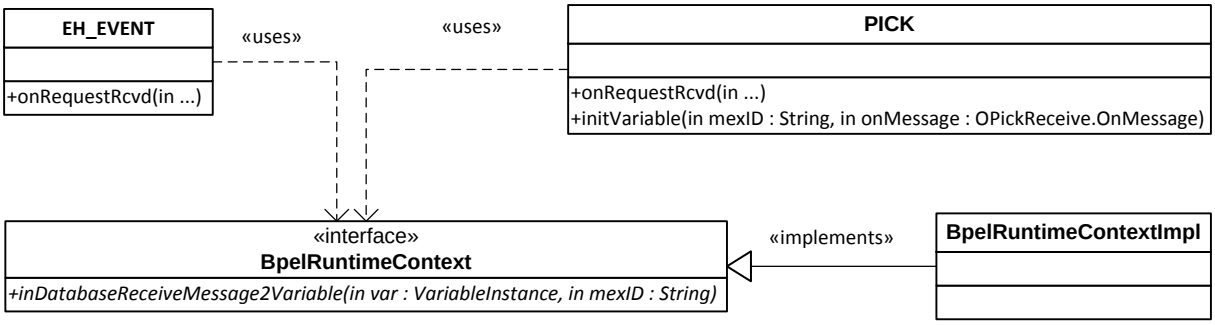

Abbildung 6.7: Am Nachrichten-Pushdown beteiligte Komponenten der Runtime-Schicht in UML-Notation. (vgl. Abb. 5.10).

Nachricht gehörende *MessageDAO*-Objekt ermittelt und anschließend dessen Pushdown-Methode *inDatabaseMessage2Variable* (vgl. Abschnitt 6.2.2) unter Angabe des Primärschlüssels der aufgelösten Zielvariable aufgerufen. Schließlich wird die interne Workflowvariable, die aufgrund der Pushdownlogik nicht verändert wurde, als veraltet gekennzeichnet (*setDetached*). Der gleiche Vorgang wird in der Klasse *EH_EVENT*für OnEvent-Handler ausgelöst. Hier wird die Methode *inDatabaseReceiveMessage2Variable* des *BpelRuntimeContext* direkt in der Methode *onRequestRcvd* aufgerufen, das Vorgehen ist ansonsten das selbe wie in der Klasse *PICK*.

6.3 XQuery-Pushdown

Um die Auswertung von XQuery-Ausdrücken innerhalb von BPEL-Aktivitäten unter Verwendung von Pushdowns innerhalb der Datenbank zu ermöglichen, müssen alle an einer Ausdruckauswertung beteiligten Komponenten erweitert werden:

- In der **DAO-Schicht** werden datenbankspezifische SQL- bzw. pureXML-Befehle in implementiert, mit deren Hilfe die XQuery-Audrücke an das DBS übermittelt werden. Dies geschieht, ebenso wie bei den anderen Pushdownarten, in der Klasse *IDNDB2*.
- In der *Runtime-Schicht* muss innerhalb der Pushdown-Logik jeder Aktivität, die XQuery unterstützen soll, über eine Fallunterscheidung die neuen Methoden der DAO-Schicht aufgerufen werden. Diese Fallunterscheidung bewirkt bisher beim Auftreten eines XQuery-Ausdrucks, dass die Pushdown-Logik abgebrochen wird und die Aktivität auf konventionelle Art durchgeführt wird, wodurch wiederum Daten entstehen, die zunächst nur im Workflowkontext verfügbar sind.

6.4 Einschränkungen der Funktionalität des erweiterten Prototyps

Die Implementierung des erweiterten Prototyps ließ sich aufgrund technischer und implementierungslogischer Besonderheiten nicht reibungslos in den vorhandenen ODE-TI-Prototyp integrieren.

Um beim Literal-Pushdown auch größere Literalwerte (>32 KB) in der DB2 speichern zu können, wird im BPEL-Compiler ein SQLXML-Objekt als Hostvariable für den Datenbankbefehl verwendet. Um diese Variable korrekt an das DBS zu übermitteln, wird ein aktualsierter JDBC-Treiber (db2jcc4.jar) für DB2 benötigt, der separat von IBM bezogen werden muss.

Die Verwendung der Wrapperelemente <*temporary-simple-type-wrapper/*> und <*xsd-complex-type-wrapper/*> führt bei Literalwerten im Zusammenhang mit den bestehenden Prototyp-Implementierungen zu Komplikationen bei der Auswertung von XPath-Ausdrücken in der Datenbank. Die Funktionalität des Literal-Pushdowns ist evtl. nicht mit allen möglichen Folgeoperationen kompatibel.

7 Evaluation des erweiterten Prototyps

In diesem Kapitel wird die in Kapitel 6 vorgestellte Implementierung des erweiterten ODE-TI-Prototypen anhand von Laufzeitmessungen evaluiert. Aufgrund technischer Komplikationen bei der endgültigen Implementierung konnte im Rahmen dieser Arbeit nur eine rudimentäre Evaluation der neuen Pushdown-Konzepte vorgenommen werden.

7.1 Testfälle

Um das Laufzeitverhalten des erweiterten Prototyps zu evaluieren, wurden Testfälle verwendet, die bereits beim bestehenden Prototypen zum Einsatz kamen.

Zur Evaluation des Literal-Pushdowns steht die Laufzeit von *ASSIGN*-Aktivitäten *ohne Ausdruck* im Vordergrund, da jeder Literalwert vor seiner weiteren Verwendung immer zuerst einer Variablen zugewiesen wird. Die darauffolgenden Aktivitäten bleiben vom Literal-Pushdown unbeeinflusst. Für die Messungen werden die selben BPEL-Prozesse verwendet, die schon zur Evaluation des Zuweisungs-Pushdowns (vgl. Abschnitt 3.3) eingesetzt wurden. Gemessen wurden Prozesse mit Literalwerten von 100 KB, 500 KB, 4 MB, 9MB sowie 50 MB Größe. Zur Auswertung der Messergebnisse wird in erster Linie die durchschnittliche Laufzeit aller Ausführungsinstanzen verwendet (im Normalfall 100 Instanzen). Innerhalb des Testprozesses wird ein Literalwert zunächst einer Variable zugewiesen, bevor diese Variable anschließend mit Hilfe des Zuweisungs-Pushdowns einer anderen Variable zugewiesen wird.

Zur Evaluation des Nachrichten-Pushdowns wurde ein einzelner, allgemeiner Testfall erzeugt, der mit verschieden Eingangsnachrichten der Größen 100 KB, 500 KB, 4 MB und 9 MB aufgerufen wird. Testläufe mit 50 MB Nachrichtengröße waren nicht möglich, da dabei die Kommunikationsinfrastruktur stets mit einer Hauptspeicherausnahme abbrach. Der zugrundeliegende Prozess betseht aus einer Receive-Aktivität, einer Assign-Aktivität, die den kompletten Inhalt der Eingangsnachricht in die Ausgangsnachricht kopiert, sowie einer Reply-Aktiviät, die diese zurückliefert.

7.2 Testumgebung

Alle Messungen wurden auf einem eigens für die Evaluation zur Verfügung gestellten Testsystem durchgeführt. Es handelt sich dabei um einen Desktop-PC mit einer Quad-Core

CPU (Intel Core2 Quad Q9300 @2.50 GHz), 4 GB RAM und installiertem Windows 7 Professional 64-Bit Betriebssystem. Als Datenbanksystem wurde IBM DB2 Advanced Enterprise Server Edition v10.1.0.872 verwendet. Alle Messungen wurden auf der Workflowenginge Apache ODE 1.3.5 unter Apache Tomcat 6.0 durchgeführt. Dabei wurden folgende Varianten unterschieden:

- **Original ODE** Die Originalversion von Apache ODE 1.3.5 unter Verwendung von DB2 als internes DBS mit unverändertem (Hibernate-)Tabellenschema.
- **Prototyp mit Literal-Pushdown** Der auf ODE 1.3.5 portierte und um die neuen Pushdown-Konzepte erweiterte Prototyp aus [Wag11] mit verändertem Datenbankschema (s. Kap. 5, Kap. 6) und ausgeschaltetem Nachrichten-Pushdown. Er dient zur Messung der Instanzlaufzeiten bei eingeschaltetem Literal-Pushdown.
- **Prototyp mit Nachrichten-Pushdown** Der selbe Protoyp mit ausgeschaltetem Literal-Pushdown und eingeschaltetem Nachrichten-Pushdown. Mit ihm sollen die Auswirkungen des Nachrichten-Pushdowns untersucht werden.
- **Prototyp ohne Erweiterungen** Der selbe Protoyp mit ausgeschaltetem Nachrichten- und Literal-Pushdown. Er repräsentiert die Implementierung des alten ODE-TI Prototyps mit den bis dahin umgesetzten Pushdown-Konzepten. Er dient hauptsächlich als Referenz für die Bewertung der neuen Pushdown-Konzepte.

Als Grundlage für Vergleiche wurden zunächst alle Messungen mit *Original ODE* durchgeführt. Danach wurden die selben Testdurchläufe mit dem zu Untersuchenden *Prototyp mit Literal-Pushdown* bzw. *Prototyp mit Nachrichten-Pushdown* und dann noch ein weiteres Mal mit dem *Prototyp ohne Erweiterungen* durchgeführt. Dadurch erhält man zum einen einen Überblick über die Leistungen der Pushdown-Implementierungen im Vergleich zu Original ODE, zum anderen eine Gegenüberstellung der Pushdown-Implementierungen mit und ohne der entsprechenden Erweiterung.

Die Gesamtlaufzeit des jeweiligen Testdurchlaufs wurde von der Startzeit der ersten Instanz bis zum Zeitpunkt der letzten Tätigkeit der letzten Instanz mit Hilfe einer eine SQL-Anfrage bestimmt. Apache ODE speichert die Informationen zu Ausführungsinstanzen in der Tabelle *BPEL_INSTANCE*. Die Testdurchläufe wurden mit Hilfe eines Perl-Skripts, das auch in [Wag11] verwendet wurde, automatisiert mehrmals direkt nacheinander ausgeführt. Der Ablauf einer Messung erfolgte nach folgenden Schritten:

1. Löschen aller Tabelleninhalte der verwendeten Datenbank, Tomcat-Logdateien sowie der in ODE deployten Prozesse (*processes*-Verzeichnis).
2. Starten des Tomcat-Servers mit der gemessenen ODE-Variante.
3. Kopieren des Testprozesses in das *processes*-Verzeichnis und warten, bis er vollständig deployt wurde. Bei den Prototyp-Varianten kann dies aufgrund des schreibenden Literal-Pushdowns etwas länger dauern (vor allem bei sehr Großen Literalwerten).
4. Erster, vollständiger Durchlauf der automatischen Testfall-Ausführung (Perl-Skript).

5. Löschen aller Instanzdaten (*BPEL_INSTANCE*) in der Datenbank sowie des Inhalts der Tomcat-Logdatei (Löschen der Datei ist zu diesem Zeitpunkt nicht möglich, da Tomcat noch läuft).
6. Zweiter, vollständiger Durchlauf der automatischen Testfall-Ausführung.
7. Stoppen des laufenden Tomcat-Servers.
8. Speichern der Tomcat-Logdatei sowie vermerken der Gesamtlaufzeit und Anzahl der gemessenen Instanzen.

Der zusätzliche, erste Durchlauf der Testfall-Ausführung wird durchgeführt, damit sich das DBS zu Beginn eines neuen Testfalls auf die neuen Daten einstellen kann. Speziell die ersten DB-Operationen eines neuen Testprozeses benötigen deutlich mehr Zeit als darauf folgende. Im nächsten Abschnitt werden die Ergebnisse der Messungen vorgestellt und anschließend diskutiert.

7.3 Messergebnisse

Um die Leistung der Pushdown-Konzepte zu untersuhen, werden alle Messergebnisse relativ zur Laufzeit von Original ODE angegeben. Neben der Evaluation der neuen Pushdown-Konzepte wird durch die Messung aller Varianten auch eine erneute Evaluation des ursprünglichen Prototyps in der portierten Version mit der Originalversion von ODE 1.3.5 ermöglicht.

Bereits bei der Implementierung des urpsrünglichen Prototyps wurde festgestellt, dass die verwendete Hibernate-DAO der Original Apache ODE Version die Daten erst nach Ende der Instanz oder an bestimmten Stellen im Workflow (z.B. während eines INVOKE) in die Datenbank überträgt und festschreibt [Wag11]. Da der Prototyp dieses Festschreiben während der jeweiligen Aktivität durchführt, sind die isolierten Messwerte der einzelnen Pushdown-Aktivitäten nicht direkt mit den Werten der Original ODE Variante vergleichbar.

7.3.1 Literal-Pushdown

Die Messergebisse sind in Abbildung 7.1 relativ zur Laufzeit von Original ODE in Prozent dargestellt. Für die Literalgrößen 100 KB, 500 KB, 4 MB und 9 MB wurden jeweils 100 Testdurchläufe durchgeführt, für 50 MB nur 10 Durchläufe.

Vergleicht man die Zuweisungen bei untersdchiedlichen Datengrößen mit den Messergebnissen aus Abschnitt 3.3.2 (Abb. 3.6), so fällt zunächst auf, dass die relativen Laufzeiten wesentlich konstanter verlaufen, als bei den dortigen Messungen, die mit ODE 1.3.4 und DB2 V9.7 durchgeführt wurden. Außerdem fällt hier die Verschlechterung des Prototyps gegenüber Original ODE mit durchschnittlich ca. 130% gegenüber ca. 200% deutlich schwächer aus. Somit hat der Zuweisungs-Pushdown ohne Ausdruck in ODE 1.3.5 mit DB2 10.1 ein besseres relatives Laufzeitverhalten als bei den vorigen Messungen. Diese Resultate

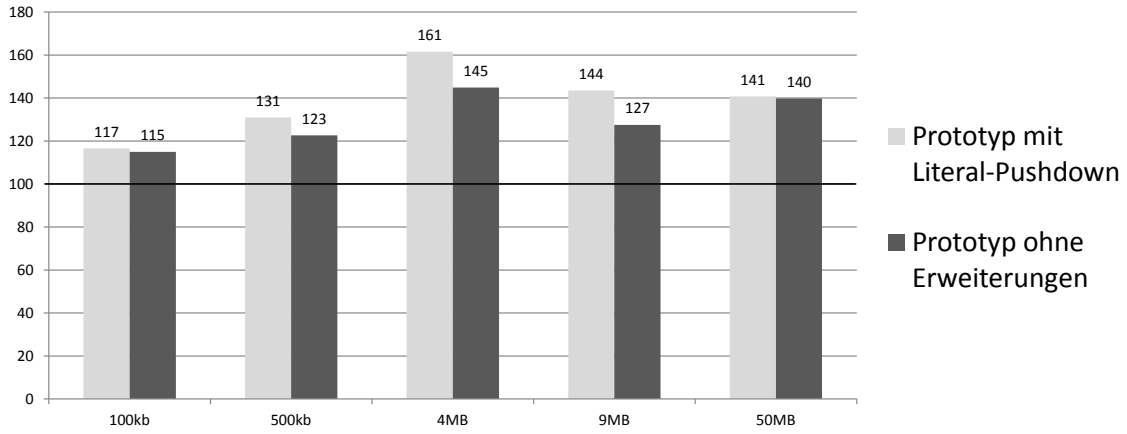

Abbildung 7.1: Instanzlaufzeiten der Testprozesse mit unterschiedlich großen Literalwerten (horizontale Achse) relativ zu Original ODE in Prozent (vertikale Achse) für den Protoyp mit Literal-Pushdown sowie den Prototyp ohne Erweiterungen.

lassen sich nur durch Unterschiede in den verschiedenen Versionen von DB2 (verbesserte Schreiboperationen) bzw. ODE erklären.

Vergleicht man die Laufzeiten der Prototyp-Varianten mit und ohne Literal-Pushdown miteinander, so ergeben sich ähnliche relative Laufzeiten, jedoch mit einer klaren Tendenz zu Geschwindigkeitseinbußen bei eingeschaltetem Literal-Pushdown. Das schlechtere Laufzeitverhalten mit Literal-Pushdown lässt sich implementierungstechnisch begründen. Beim Prototypen ohne Erweiterungen wird der Literalwert in ODE intern zugewiesen und durch Hibernate in die Datenbank geschrieben. Diese Vorgänge laufen offenbar effizienter ab als die Zuweisung des Literalwerts aus der Literaltabelle in die Tabelle BPEL_XML_DATA. Dies könnte mit der Tatsache zusammenhängen, dass für eine Literalzuweisung momentan ein einfacher XPath-Ausdruck verwendet werden muss (s. Abschnitt 6.1, Listing 6.1), um den Wert aus der Literaltabelle auszulesen, anstatt das gesamte Datenfeld in die andere Tabelle zu kopieren. Es lässt sich also abschließend sagen, dass die Implementierung des Literal-Pushdown in ihrer momentanen Form keinen Vorteil bezüglich der Laufzeit von Literalzuweisungen darstellt, jedoch kann und sollte sie optimiert und anschließend erneut evaluiert werden.

7.3.2 Nachrichten-Pushdown

Die Ergebisse der Messungen zum Nachrichten-Pushdown sind in Abbildung 7.2 relativ zur Laufzeit von Original ODE in Prozent dargestellt. Für die Nachrichtengrößen 100 KB und 500 KB wurden jeweils 100 Testdurchläufe durchgeführt, für 4 MB und 9 MB nur 50 Durchläufe. Alle Messungen umspannen auch die Zeit, die das aufrufende Skript zwischen zwei Aufrufen benötigt, um die Rückantwort in Form einer Textdatei auf der Festplatte zu speichern und die nächste Nachricht an ODE zu übermitteln. Bei den Messungen des

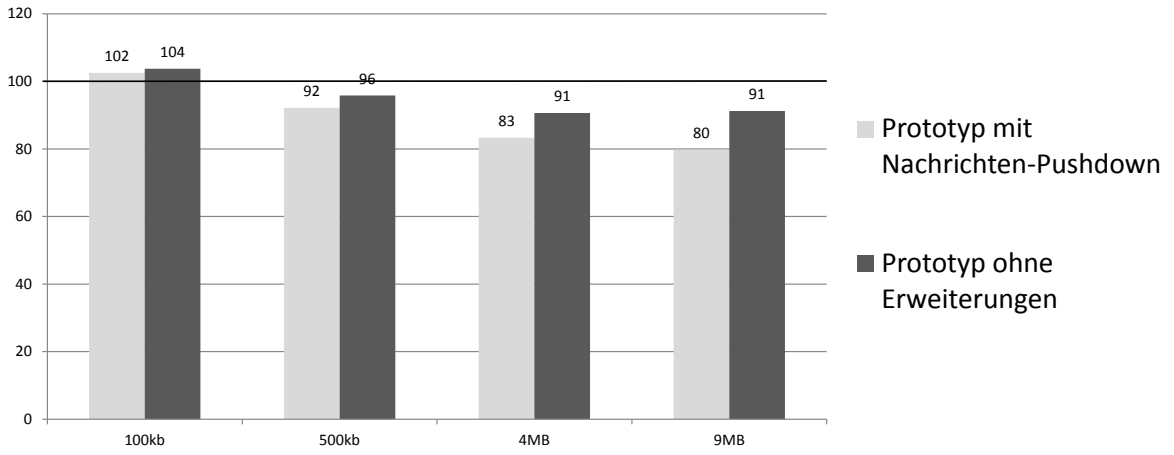

Abbildung 7.2: Instanzlaufzeiten des Testprozesses mit unterschiedlich großen Eingangsnachrichten (horizontale Achse) relativ zu Original ODE in Prozent (vertikale Achse) für den Prototyp mit Nachrichten-Pushdown sowie den Prototyp ohne Erweiterungen.

Nachrichten-Pushdowns wird diese Zeit durch die Größe der Eingangsnachricht beeinflusst und fließt in die Gesamtlaufzeit mit ein.

Die Messungen zeigen zum einen einen Leistungsgewinn durch die Verwendung von Pushdown-Techniken gegenüber Original ODE bei größeren Datenmengen im Allgemeinen, insbesondere aber auch eine Verbesserung durch die Verwendung des Nachrichten-Pushdowns gegenüber des Prototyps ohne Erweiterungen. Der allgemeine Leistungsgewinn bei steigender Datengröße lässt sich durch die Effizienz des DBS beim Zuweisungs-Pushdown während des Kopierens der Eingangsvariable in die Ausgangsvariable erklären. Der Leistungsgewinn durch den eingeschalteten Nachrichten-Pushdown lässt sich damit erklären, dass die Implementierung der Nachricht-zu-Variable Zuweisung innerhalb des DBS effizienter ist, als die Zuweisung innerhalb von ODE und das anschließende Festschreiben in der Datenbank. Da hier die Zuweisung aus einer Datenbanktabelle in eine andere effizienter ist, als die Verarbeitung in ODE, bestärkt dies die Annahme, dass der Literal-Pushdown nach durch weitere Optimierungen ebenfalls zu einem Leistungsgewinn führen kann.

7.3.3 Anwendungsfall

Das Szenario mit dem Testprozess aus dem Gebiet der Proteinmodellierung (vgl. Abschnitt 2.3.1, Abschnitt 3.3) konnte im Rahmen dieser Arbeit aufgrund technischer Komplikationen bei der Implementierung des Literal-Pushdowns nicht evaluiert werden. Erste Versuche haben jedoch gezeigt, dass die Ausführung des Testprozesses unter der Testumgebung mit Original ODE bereits für Dateigrößen von 500 KB nicht korrekt durchläuft. Da eine Auswertung mit dem erweitereten Prototypen zu diesem Zeitpunkt ebenfalls nicht möglich ist, muss die Evaluation der Kombinierten Pushdown-Konzepte anhand des Anwendungsfalls zu einem späteren Zeitpunkt erfolgen.

8 Zusammenfassung und Ausblick

Um große Datenmengen innerhalb von Workflows effizienter und zuverlässiger handhaben zu können, wurden Konzepte zur Verbesserung der Workflowausführung durch Integration eines DBS entwickelt. Dabei entstanden verschiedene Pushdown-Konzepte, die dem Workflow Aufgaben der Datenverarbeitung abnehmen und in das integrierte DBS auslagern. Diese Konzepte wurden bereits implementiert und evaluiert, wobei sich herausstellte, dass durch ihre Umsetzung eine erhöhte Zuverlässigkeit sowie für bestimmte Szenarien ein verbessertes Laufzeitverhalten der Workflowausführung erreicht werden kann (vgl. Abschnitt 3.3.2).

In dieser Arbeit wurden zusätzliche Konzepte entwickelt, die zum Ziel hatten, die bestehenden Pushdown-Konzepte zu erweitern und zu vervollständigen. Dabei wurden die Konzepte *Literal-Pushdown*, *Nachrichten-Pushdown* sowie *XQuery-Pushdown* herausgearbeitet, wobei im Rahmen dieser Arbeit nur der *Literal-Pushdown* und der *Nachrichten-Pushdown* Implementiert und evaluiert wurden.

Die beiden umgesetzten Pushdown-Konzepte wurden im Rahmen dieser Arbeit anhand von Zeitessungen evaluiert. Dabei wurde festgestellt, dass sie im Zusammenspiel mit den bisherigen Pushdown-Konzepten durchaus zu weiteren Leistungssteigerungen der Workflowausführung führen können, sofern die entprechenden Erweiterungen mit einer effizienten Implementierung umgesetzt werden. Bei der Evaluation des Nachrichten-Pushdowns konnten Leistungssteigerungen bereits anhand der prototypischen Implementierung nachgewiesen werden (vgl. Abschnitt 7.3.2). Im Fall des Literal-Pushdowns tendierte dessen prototypische Implementierung zu kleinen, aber erkennbaren Leistungseinbußen, wobei dort aber bereits Ansätze für Optimierungen aufgezeigt wurden. Dennoch konnte dabei aufgrund des verwendeten Testfalls eine allgemeine Leistungsverbesserung bei der Verwendung der Pushdown-Konzepte im Vegleich zur Evaluation mit älteren Versionen von DB2 und ODE festgestellt werden (vgl. Abschnitt 7.3.1).

8.0.4 Ausblick

Aufgrund von technischen Komplikationen bei der Implementierung ist die Evaluation der erweiterten Konzepte nur rudimentär erfolgt. Insbesondere konnte keine Auswertung von Testfällen für einen Anwendungsfall erfolgen, bei dem alle implementierten Konzepte im Zusammenspiel miteinander untersucht weden. Als weiterführende Arbeit ist daher zunächst die Verbesserung der Implementierung vorrangig zu behandeln. Nach Behebung aller bestehenden Probleme an der prototypischen Implementierung (sowie ggf. weiterer Optimierungen), kann eine detaillierte Evaluation anhand von Anwendungsfällen erfolgen.

Von besonderem Interesse sind dabei Messungen am Anwendungsfall aus der Proteinmodellierung, der bereits vom bestehenden ODE-TI-Prototypen zur Evaluation herangezogen wurde (Abschnitt 3.3).

Nach der Verbesserung des erweiterten Prototyps und seiner Evaluation ist der nächste Schritt eine tatsächliche Implementierung und Evaluation des *XQuery-Pushdown*. Dazu gehört neben der Umsetzung innerhalb von Apache ODE auch das Erstellen geeigneter Testfälle für Laufzeitmessungen unter Verwendung von XQuery-Ausdrücken. Je nach Komplexität der dabei verwendeten Ausdrücke sowie der Größe der Daten ist durch die Auswertung innerhalb des DBS, ähnlich wie beim bisherigen Pushdown mit XPath-Ausdrücken, ein Vorteil gegenüber der konventionellen Implementierung innerhalb von ODE zu erwarten.

Die bereits durch [Wag11] (dort Kapitel 8) vorgestellen konzeptionellen Erweiterungen bleiben von dieser Arbeit weitestgehend unberührt. Die dort vorgeschlagene Referenzarchitektur, bei der eine Trennung zwischen der DAO-Schicht und einer Pushdown-Schicht stattfindet, sollte langfristiges Ziel bei der engen Integration von DBS in WfMS sein. Insbesondere führt momentan die gemeinsame Nutzung von Datenbanktabellen durch Pushdownlogik und Persistenzmanager zu gewissen Einschränkungen. So kann momentan beispielsweise eine Variable erst von einer Pushdown-Aktivität verwendet werden, nachdem diese durch den Persistenzmanager in der Datenbank erzeugt worden ist. Idealerweise sollte eine Pushdown-Implementierung komplett unabhängig von der DAO-Schicht eines gegebenen WfMS sein.

Literaturverzeichnis

[AIL98] A. Ailamaki, Y. E. Ioannidis, M. Livny. Scientific workflow management by database management. In *Tenth Int Scientific and Statistical Database Management Conf*, pp. 190–199. 1998. (Zitiert auf Seite 31)

[AMA06] A. Akram, D. Meredith, R. Allan. Evaluation of BPEL to Scientific Workflows. *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*, 2006. (Zitiert auf Seite 22)

[bpe] Web Services Business Process Execution Language Version 2.0 - OASIS Standard. URL http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf. (Zitiert auf den Seiten 21, 23 und 48)

[dom] Document Object Model (DOM). URL http://www.w3.org/DOM/. (Zitiert auf Seite 14)

[G+11] K. Görlach, et al. *Guide to e-Science*, chapter Conventional Workflow Technology for Scientific Simulation, pp. 323–352. Springer, 2011. (Zitiert auf den Seiten 5, 17 und 20)

[KE06] A. Kemper, A. Eickler. *Datanbanksysteme - Eine Einführung*. Oldenbourg Verlag München Wien, 6 edition, 2006. (Zitiert auf den Seiten 9 und 23)

[KKL+05] M. Kloppmann, D. Koenig, F. Leymann, G. Pfau, A. Rickayzen, C. von Riegen, P. Schmidt, I. Trickovic. WS-BPEL Extension for People - BPEL4People, 2005. (Zitiert auf Seite 21)

[LR00] F. Leymann, D. Roller. *Production Workflow - Concepts and Techniques*. Prentice Hall PTR, 2000. (Zitiert auf den Seiten 9 und 20)

[ode] ODE - Architectural Overview. URL http://ode.apache.org/architectural-overview.html. (Zitiert auf den Seiten 5 und 44)

[RRS+11] P. Reimann, M. Reiter, H. Schwarz, D. Karastoyanova, F. Leymann. SIMPL - A Frameworrk for Accessing External Data in Simulation Workflows, 2011. (Zitiert auf den Seiten 20 und 26)

[RSM] P. Reimann, H. Schwarz, B. Mitschang. DATA PROVISIONING TECHNIQUES FOR SIMULATION WORKFLOWS. *Unveröffentlichter Bericht des Instituts für Parallel und Verteilte Systeme*. (Zitiert auf den Seiten 5, 6, 26 und 27)

[RSM11] P. Reimann, H. Schwarz, B. Mitschang. Design, Implementation, and Evaluation of a Tight Integration of Database and Workflow Engines. *Journal of Information and Data Management*, 2(3), 2011. (Zitiert auf den Seiten 5, 10, 18, 19, 28, 29, 30, 31, 32, 33 und 35)

[sql] ISO/IEC 9075. (Zitiert auf Seite 24)

[TDG07] I. Taylor, E. Deelman, D. Gannon. *Workflows for e-Science - Scientific Workflows for Grids*. Springer, 2007. (Zitiert auf den Seiten 9 und 17)

[VSRM08] M. Vrhovnik, H. Schwarz, S. Radeschütz, B. Mitschang. An Overview of SQL Support in Workflow Products. In *IEEE 24th Int. Conf. Data Engineering ICDE*, pp. 1287–1296. 2008. (Zitiert auf Seite 26)

[Wag11] F. Wagner. *Nutzung einer integrierten Datenbank zur effizienten Ausführung von Workflows*. Diplomarbeit, Institut für Parallele und Verteilte Systeme, Universität Stuttgart, 2011. (Zitiert auf den Seiten 5, 6, 10, 31, 36, 43, 45, 46, 47, 49, 51, 52, 55, 56, 57, 58, 60, 74, 75 und 80)

[xmla] Extensible Markup Language (XML) 1.0 (Fifth Edition). URL http://www.w3.org/TR/REC-xml/. (Zitiert auf Seite 13)

[xmlb] XML Schema. URL http://www.w3.org/XML/Schema. (Zitiert auf Seite 13)

[xpa] XML Path Language (XPath) Version 1.0 - W3C Recommendation 16 November 1999. URL http://www.w3.org/TR/xpath/. (Zitiert auf Seite 15)

[xpd] XPDL Support and Resources. URL http://www.wfmc.org/xpdl.html. (Zitiert auf Seite 17)

[xqu] XQuery 1.0: An XML Query Language (Second Edition). URL http://www.w3.org/TR/xquery/. (Zitiert auf den Seiten 14 und 15)

Angegebene Links wurden zuletzt überprüft am: 06.12.2012

Erklärung

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

(Christian Ageu)