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

Studienarbeit Nr. 2341

Erweiterung von SIMPL und BPEL-DM zur Unterstützung weiterer Typen von Datenquellen

Henrik Andreas Pietranek

Studiengang:	Informatik
Prüfer:	Prof. Dr.-Ing. habil. Bernhard Mitschang
Betreuer:	Dipl.-Inf. Peter Reimann
begonnen am:	09. Juni 2011
beendet am:	09. Dezember 2011
CR-Klassifikation:	D.2.11, H.2.5, H.4.1, I.6.7

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

1. Einleitung

In den vergangenen Jahren haben sich im unternehmerischen Umfeld Workflows zur Beschreibung und Ausführung von (Geschäfts-)Prozessen durchgesetzt [LR99]. Seit kurzem wird diese Technologie auch in der Wissenschaft eingesetzt. Z.B. werden Simulationsabläufe als Workflows modelliert. Charakteristisch für solche Simulationen bzw. Simulationsabläufe sind komplexe mathematische Berechnungen sowie verschiedene Aufgaben im Bereich der Datenverwaltung und Datenbereitstellung. Oftmals müssen große Datenmengen, die in proprietären Formaten vorliegen, aus verschiedenen Quellen verarbeitet werden. Damit diese Daten durch einen Simulationsworkflow und den von ihm eingebundenen Programmen und Diensten verarbeitet werden können, müssen sie in passende Eingabeformate transformiert werden. Dies hat einen erhöhten Arbeitsaufwand für den Modellierer zur Folge, der typischerweise der Wissenschaftler selbst ist. Dieser muss zum einen adäquate Quellen finden und zum anderen benötigte Transformationen implementieren oder manuell durchführen. Gerade bei umfangreichen Simulationen, die eine Vielzahl an Datenquellen benötigen, führt dies aufgrund der enormen Komplexität zu Problemen. Einerseits kann sich der Wissenschaftler nicht mehr auf sein Kernproblem (die eigentliche Simulation) konzentrieren, andererseits steigt oftmals die Fehlerrate bei der Datenverarbeitung, da auch mit benötigtem Fachwissen solche Transformationen fehleranfällig sind [RRS+11].

Um diese Probleme zu lösen, wurde das SIMPL-Rahmenwerk (SimTech - Information Management, Processes and Languages) entwickelt. Das SIMPL-Rahmenwerk ist in ein Scientific Workflow Management System eingebettet und schafft eine Abstraktionsebene für die Definition des Datenmanagements. SIMPL bietet einheitliche Zugriffsmethoden, um, aus einem Simulationsworkflow heraus, auf beliebige Datenquellen zuzugreifen. Zur Prozessdefinition wird die Business Process Execution Language (BPEL) in einer erweiterten Form verwendet. Die Business Process Execution Language extension for Data Management (BPEL-DM) erweitert BPEL um zusätzliche Datenmanagement-Aktivitäten. Dadurch ist es möglich, das Datenmanagement direkt in Workflow-Modellen zu definieren [RRS+11]. Das SIMPL-Rahmenwerk wurde prototypisch umgesetzt und unterstützt bisher den Zugriff auf relationale Datenbanken sowie das Windows Dateisystem.

Im Rahmen dieser Studienarbeit sollen Unterschiede, die sich beim Zugriff auf unterschiedliche Datenquellen oder Typen von Datenquellen ergeben, soweit wie möglich bzw. sinnvoll aufgelöst werden. Des Weiteren sollen neue Datenquellen bzw. Typen von Datenquellen integriert werden. Dazu wird im Folgenden das SIMPL-Rahmenwerk detailliert vorgestellt und in einem weiteren Schritt die bestehende Implementierung analysiert und erweitert bzw. angepasst.

Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Grundlagen: Die Grundlagen dieser Arbeit werden hier beschrieben.

Kapitel 3 – SIMPL-Rahmenwerk Das SIMPL-Rahmenwerk wird in diesem Kapitel detailliert vorgestellt.

Kapitel 4 – Bestandsaufnahme Die bestehende Implementierung von SIMPL wird in diesem Kapitel erläutert.

Kapitel 5 – Analyse Eine Analyse der bestehenden Implementierung findet in diesem Kapitel statt. Ziel ist es Gemeinsamkeiten bzw. Unterschiede, die sich beim Zugriff auf die einzelnen Datenquellen ergeben, herauszufinden. Des Weiteren werden die neu zu integrierenden Datenquellen hinsichtlich ihrer Gemeinsamkeiten analysiert.

Kapitel 6 – Umsetzung Auf die konkrete Umsetzung der im vorherigen Kapitel vorgestellten Ideen wird in diesem Kapitel eingegangen.

Kapitel 7 – Zusammenfassung und Ausblick Die Ergebnisse dieser Arbeit werden hier zusammengefasst. Des Weiteren werden nicht gelöste Probleme dargelegt.

2. Grundlagen

In diesem Kapitel werden die Grundlagen, die für das Verständnis der vorliegenden Arbeit nötig sind, erläutert. Es werden die Themen XML, Serviceorientierte Architektur, Workflow, Datenbanken und Sensornetze behandelt.

2.1. XML

XML, die Extensible Markup Language, ist eine Auszeichnungssprache. Mithilfe einer Auszeichnungssprache können Dokumentformate bzw. deren Inhalte beschrieben werden. XML wurde unter der Schirmherrschaft des World Wide Web Consortiums (W3C) entwickelt und 1998 als *Recommendation* veröffentlicht. Im Jahr 2008 wurde die fünfte Version der XML-Spezifikation [BPSM+08] veröffentlicht. Heute wird XML insbesondere als standardisiertes Datenaustauschformat, welches anwendungs- und plattformunabhängig ist, verwendet. Die folgenden Abschnitte basieren auf [KE11], [Sch03] und [Seb10].

2.1.1. Tags, Elemente und Attribute

Listing 2.1 zeigt einen Ausschnitt eines XML-Dokuments. Die Zeichenfolgen zwischen den spitzen Klammern heißen Tags. Zwei Tags bilden jeweils ein Paar. Es gibt ein Start- und ein End-Tag. Ein End-Tag besitzt immer die gleichen Zeichenfolgen wie das zugehörige Start-Tag und wird zusätzlich durch ein „/“eingeleitet. In unserem Beispiel ist *<Erster_Vorname>* ein Start- und *</Erster_Vorname>* das zugehörige End-Tag. Zwischen den einzelnen Tags werden die eigentlichen Information gespeichert. Die Zeichenfolge *<Erster_Vorname>Henrik</Erster_Vorname>*, also ein Start- und End-Tag sowie der Inhalt dazwischen, bildet ein Element. Das Wurzelelement ist das oberste Element der Hierachie, die vom XML-Dokument gebildet wird. Elemente können einfachen Text, andere Elemente oder eine Mischung aus beidem beinhalten. So ist es möglich, dass ein Element beliebig viele Unterelemente beinhaltet. Im gegebenen XML-Fragment ist das Element Student das Wurzelelement, da es alle anderen Elemente umschließt.

Listing 2.1 Ein einfaches XML-Fragment

```
<Student>
  <Erster_Vorname>Henrik</Erster_Vorname>
  <Zweiter_Vorname>Andreas</Zweiter_Vorname>
  <Nachname Geburtsjahr="1986">Pietranek</Nachname>
</Student>
```

Zusätzlich können für ein Element noch ein oder mehrere Attribute angegeben werden. Mithilfe von Attributen werden zusätzliche Elementeigenschaften beschrieben. Attribute werden immer im Start-Tag eines Elementes positioniert. Das Element Nachname im Beispiel enthält noch zusätzlich das Attribut Geburtsjahr.

2.1.2. Aufbau eines XML-Dokuments

Ein XML-Dokument besteht aus den folgenden drei Teilen:

- optionaler Prolog
- optionales Schema
- ein einziges Wurzelelement

Der optionale Prolog enthält Informationen über die zugrundeliegende XML-Version sowie den Zeichensatz zur Speicherung des Dokumentinhalts und deklariert ein Dokument als XML-Dokument. Obwohl der Prolog optional ist, kann er die weitere Verarbeitung eines Dokuments vereinfachen, da dadurch bekannt ist, um welches Format es sich handelt. Das optionale Schema legt Anforderungen fest, die das Dokument erfüllen muss, wie z.B. die Struktur des Dokumentinhalts. Es gibt zwei Möglichkeiten solche Anforderungen zu definieren. Die Erstellung einer Document Type Definition (DTD) oder die Verwendung des neueren XML Schema. Abschnitt 2.1.4 beschäftigt sich mit diesen beiden Möglichkeiten. Das Wurzelelement, das oberste Element der Hierachie, beinhaltet alle weiteren Unterelemente.

Ein XML-Dokument heißt wohlgeformt, wenn es den syntaktischen Anforderungen von XML genügt. Dazu gehört unter anderem, dass es nur ein einziges Wurzelemement gibt und, dass ein Element keine zwei Attribute mit demselben Namen beinhaltet. Gültig oder valide heißt ein XML-Dokument, wenn es wohlgeformt ist und zusätzlich die in einer DTD oder in einem XML Schema definierten Anforderungen erfüllt.

2.1.3. XML-Namensräume

Ein einzelnes Wort kann mehrere Bedeutungen haben. Listing 2.2 zeigt ein XML-Fragment, in dem das Element *Titel* zweimal vorkommt.

Listing 2.2 Mehrdeutigkeiten in XML

```
<Vorlesungen>
  <Vorlesung VorNr="111">
    <Titel>Datenbanksysteme</Titel>
    <Dozent>
      <Name>Mustermann</Name>
      <Titel>Professor</Titel>
    </Dozent>
    <SWS>4</SWS>
  </Vorlesung>
</Vorlesungen>
```

Im ersten Fall wird der Name einer Vorlesung und im zweiten Fall der akademische Grad des Dozenten beschrieben. In *einem* XML-Dokument werden also zwei verschiedene Vokabulare verwendet.

Eine mögliche Lösung ist die Verwendung von Namensräumen. Jedem Element wird ein eindeutiger Name zugewiesen, der aus dem Namensraum und dem eigentlichen Namen des Elementes besteht. Die Identifizierung des zugehörigen Namensraums erfolgt durch eine global eindeutige URI (Uniform Resource Identifier). Listing 2.3 zeigt den Inhalt von Listing 2.2 in modifizierter Fassung. Standardmäßig wird der Namensraum *Universität* verwendet. Bei der Beschreibung des Dozenten wird auf den Namensraum *Person* zurückgegriffen. Auf diese Weise können Mehrdeutigkeiten beseitigt werden.

Listing 2.3 Namensräume in XML

```
<Universitaet xmlns="http://www.BeispielUniversitaet.de/universitaet">
  <Vorlesungen>
    <Vorlesung VorNr="111">
      <Titel>Datenbanksysteme</Titel>
      <Dozent>
        <person:Name>Mustermann</person:Name>
        <person:Titel>Professor</person:Titel>
      </Dozent>
      <SWS>4</SWS>
    </Vorlesung>
  </Vorlesungen>
</Universitaet>
```

2.1.4. Document Type Definiton und XML Schema

XML wird insbesondere zum Daten- bzw. Informationsaustausch eingesetzt. Deshalb ist es wichtig, dass beide Kommunikationspartner dasselbe Verständnis von den ausgetauschten Informationen haben. Das verwendete Vokabular wird durch Namensräume definiert. Eine DTD oder ein XML Schema legt weiterhin die Struktur eines Dokuments fest.

Listing 2.4 Dokument Type Definition (in Anlehnung an [Sch03])

```
<!DOCTYPE Vorlesung[
  <!ELEMENT Vorlesung (VorName, Dozent+, SWS)>
  <!ATTLIST Vorlesung VorNr CDATA #REQUIRED>
  <!ELEMENT VorName (#PCDATA)>
  <!ELEMENT Dozent (#PCDATA)>
  <!ELEMENT SWS (#CDATA)
]>

<Vorlesung VorNr="111">
  <VorTitel>Datenbanksysteme</VorTitel>
  <Dozent>Mustermann</Dozent>
  <SWS>4</SWS>
</Vorlesung>
```

Eine DTD beginnt stets mit der Zeichenfolge „<!DOCTYPE“. Anschließend folgen für jeden Dokumenttyp Elementtyp- und Attributtypdefinitionen. Listing 2.4 zeigt eine DTD für Informationen zu Vorlesungen. Jede Vorlesung hat als Attribut eine Vorlesungsnummer, als Elemente einen Vorlesungsnamen und eine Anzahl an Semesterwochenstunden (SWS) und kann von einem oder beliebig vielen Dozenten gehalten werden. Würde man das + bei „Dozent“durch ein * ersetzen, könnte die Vorlesung auch von keinem Dozenten gehalten werden. Das Element Vorlesung muss das Attribut *Vorlesung-NR* enthalten, da es als *REQUIRRED* angegeben ist. Optionale Attribute können durch den Zusatz *IMPLIED* definiert werden. Es ist nicht genau festgelegt, welche Eigenschaften eines Elementes als Attribut, und welche als Unterelement modelliert werden sollten. Hier geben die eigenen Vorlieben den Ausschlag.

Jedoch besitzt eine Document Type Definiton nur eine begrenzte Ausdruckskraft. [KE11] bemängelt die folgenden Aspekte:

> „So lassen sich keine komplexen Integritätsbedingungen angeben und auch die Datentypen sind sehr eingeschränkt (im Wesentlichen gibt es nur Strings, d.h. PCDATA).“

Aus diesem Grund wurde vom World Wide Web Consortium das XML Schema entwickelt. Nach [Seb10] gibt es vier Anwendungsgebiete, die durch XML Schema unterstützt werden:

- Validierung
- Abfrageunterstützung
- Datenbindung und Editierung
- Dokumentation

Das Anwendungsgebiet Validierung kann in die beiden Teilgebiete Strukturvalidierung und Datenvalidierung unterteilt werden. Bei der Strukturvalidierung wird überprüft, ob definierte Elemente und Attribute im Dokument vorkommen. Der Inhalt der Elemente und Attribute wird bei der Datenvalidierung bewertet. Das Element *Telefonnummer* darf z.B. nur Zahlen enthalten (Validierung). XPath 2.0 [BBC+11] und XQuery 1.0 [BCF+10] bieten Unterstützung von XML Schema Datentypen. Auf diese Weise können Dokumente genauer analysiert und Inhalte abgefragt werden (Abfrageunterstützung). Neuere XML-Editoren können für ein gegebenes XML Schema Instanzdokumente erzeugen (Datenbindung und Editierung). Des Weiteren kann ein XML Schema zur Dokumentation verwendet werden. Das verwendete Vokabular kann festgehalten oder ein Schema durch einen Editor in eine für Menschen verständliche Fassung (Visualisierung) überführt werden (Dokumentation).

Listing 2.5 zeigt einen Ausschnitt aus einem XML Schema zur Beschreibung von Universitäten und deren Fakultäten. Zur einfacheren Darstellung wurden benötigte Definitionen teilweise weggelassen. Das oberste Element der Hierachie ist das Element *Universitaet*, welches vom Typ *tUniversitaet* ist. Dieser Typ wird anschließend definiert. Er besteht aus den zwei Elementen *NameUndOrt* und *Fakultaeten*. Das Element *NameUndOrt* ist wiederum ein *complexType*, der aus den beiden Elementen *NameDerUniversitaet* und *OrtDerUniversitaet*

besteht. Dem Typ für das Element *NameUndOrt* wurde jedoch kein Name zugewiesen. Stattdessen wurde eine lokale, anonyme Typdefinition erstellt.

Listing 2.5 Ausschnitt aus einem XML Schema

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="Universitaet" type ="tUniversitaet"/>

  <xs:complexType name="tUniversitaet">
    <xs:sequence>
      <xs:element name="NameUndOrt">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="NameDerUniversitaet" type="xs:string"/>
            <xs:element name="OrtDerUniversitaet" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="Fakultaeten">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Fakultaet" minOccurs="1" maxOccurs="unbounded"
                type="tFakultaet"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="tFakultaet">
    <xs:sequence>
      <xs:element name="NameDerFakultaet" type="xs:String"/>
      <xs:element name="VorlesungenDerFakultaet" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Vorlesung" type="tVorlesung"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="NummerDerFakultaet" type="xs:ID"/>
  </xs:complexType>

  <xs:complexType name="tVorlesung">...</xs:complexType>

</xs:schema>
```

Die einzelnen Fakultäten, als Unterelemente des Elementes *Fakultaeten* definiert, sind vom Typ *tFakultaet*. Da für das Element *Fakultaet minOccurs=1* angegeben wurde, muss eine Universität mindestens eine Fakultät haben. Werden für *minOccurs* und *maxOccurs* keine Werte angegeben, werden die beiden Werte standardmäßig als 1 interpretiert. Jede Fakultät

hat eine Fakultätsnummer. Diese Nummer wird mit Hilfe des Attributs *NummerDerFakultaet* festgehalten. Dieses Attribut ist vom Typ *ID*. Das bedeutet, dass diese Nummer eindeutig sein muss. Es darf im gesamten XML-Dokument kein anderes Attribut vom Typ *ID* mit demselben Wert geben, so dieses Attribut einen Primärschlüssel darstellt. Mithilfe des Elementes *NameDerFakultaet* wird der Fakultätsname gespeichert. Die Vorlesungen, die eine Fakultät anbietet, werden als Kindelemente des Elementes *VorlesungenDerFakultaet* gespeichert. Für detailliertere Informationen wird auf [KE11] verwiesen.

2.2. Serviceorientierte Architektur

Der Begriff *Serviceorientierte Architektur (SOA)* steht für ein Architekturparadigma und keine konkrete Technik bzw. Implementierungsmethode. Bisher gibt es keine einheitliche Definition. Teilweise beziehen Definitionen Aspekte mit ein, die bei anderen Definition vollständig vernachlässigt werden. [Mel10] definiert den Begriff Serviceorientierte Architektur folgendermaßen:

> „Unter einer SOA versteht man eine Systemarchitektur, die vielfältige, verschiedene und eventuell inkompatible Methoden oder Applikationen als wiederverwendbare und offen zugreifbare Dienste repräsentiert und dadurch eine plattform- und sprachenunabhängige Nutzung und Wiederverwendbarkeit ermöglicht."

Bei einer Serviceorientierten Architektur steht der Begriff Service im Mittelpunkt. Im weiteren Verlauf dieser Arbeit wird die deutsche Übersetzung *Dienst* verwendet.

2.2.1. Merkmale einer SOA

Nach [Mel10] zählen unter anderem die folgenden Punkte zu den grundlegenden Merkmalen einer SOA: *Lose Kopplung* bedeutet, dass Dienste erst bei Bedarf von Anwendungen oder anderen Diensten dynamisch gesucht und eingebunden werden. Bei der Übersetzung einer Anwendung steht noch nicht fest, welcher Dienst eine anfallende Aufgabe übernimmt. Dazu wird ein *Verzeichnisdienst* benötigt, in dem alle verfügbaren Dienste registriert sind. Bei Bedarf sucht eine Anwendung einen passenden Dienst in diesem Verzeichnissdienst. Dazu ist es nötig, dass *offene Standards* verwendet werden und Beschreibungen von Schnittstellen in einer maschinenlesbaren Form vorliegen. Ein weiteres Merkmal einer SOA ist die *Orchestrierung* von Diensten. Unter dem Aspekt der Wiederverwendbarkeit können die angebotenen Dienste zu größeren Diensten, z.B. den Geschäftsprozessen eines Unternehmens, zusammengesetzt (orchestriert) werden. Geschäftsprozesse werden meist von externen *Ereignissen* angetrieben. Die Komponenten einer SOA können auf solche Ereignisse reagieren und bestimmte Aktionen auslösen.

2.2.2. Beteiligte und deren Aktionen

Innerhalb einer SOA agieren die folgenden drei Beteiligten:

- Dienstnutzer
- Dienstanbieter
- Verzeichnisdienst

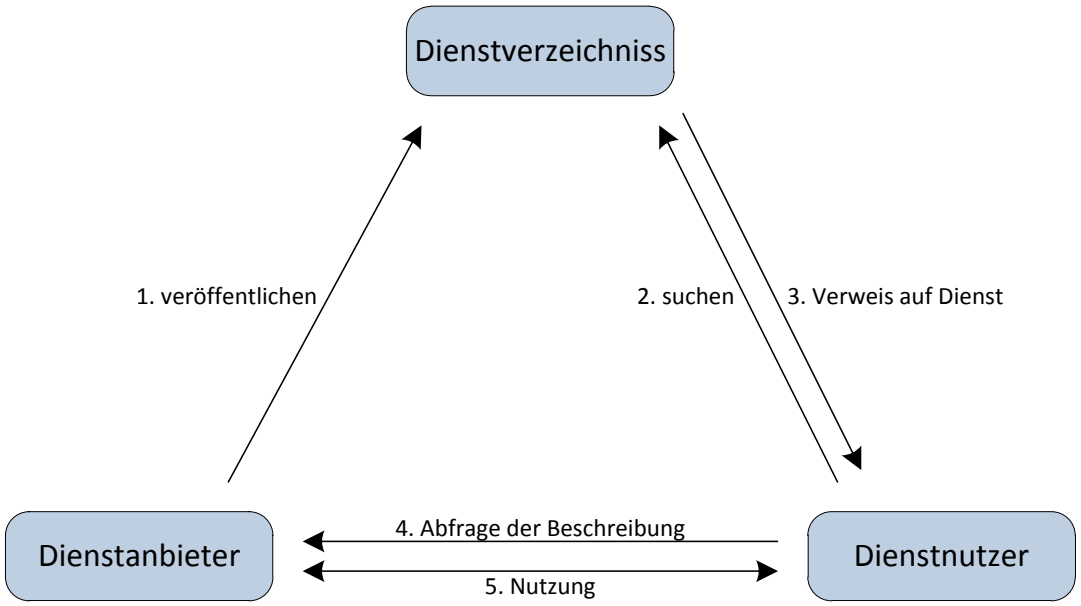

Abbildung 2.1.: Beziehungen innerhalb einer SOA (in Anlehnung an [Mel10])

Abbildung 2.1 zeigt das Zusammenspiel aller Beteiligten. Nach [Mel10] erstellt der Dienstanbieter eine *Service Description* für seine Anwendung bzw. Softwarekomponente. Diese Service Description liegt in maschinenlesbarer Form vor und beinhaltet, wie bereits erwähnt, die Beschreibung der eigentlichen Funktion, sowie die Beschreibung der öffentlichen Schnittstellen. Oftmals werden auch noch nichtfunktionale Anforderungen, wie z.B. maximale Antwortzeiten, festgehalten. Des Weiteren betreibt der Dienstanbieter die benötigte Infrastruktur, wie z.B. Rechenzentren, und übernimmt Aufgaben, die im Bereich der Quality of Service (wie z.B. Verfügbarkeit und Datenschutz) liegen. Der Dienstanbieter installiert (deployt) den Dienst innerhalb seiner Infrastruktur und registriert diesen anschließend im Dienstvereichnis. Der Dienstnutzer kann nun im Verzeichnissdienst nach einem passenden Dienst suchen. Ist dieser gefunden, wird die Schnittstellenbeschreibung des Dienstes vom Dienstanbieter abgefragt und mit diesem über Rahmenbedingungen, wie z.B. über Zertifikate, verhandelt. Anschließend kann der gewählte Dienst genutzt werden.

2.2.3. Web Service

Implementierungen einer SOA sind unter anderem *RESTful Services* oder *Web Services* [BHM+04]. Die folgende Definition für den Begriff Web Service stammt aus [BKNT10]:

> „Ein Web Service ist eine durch einen URI eindeutige identifizierte Softwareanwendung, deren Schnittstellen als XML-Artefakte definiert, beschrieben und gefunden werden können. Ein Web Service unterstützt die direkte Interaktion mit anderen Softwareagenten durch XML-basierte Nachrichten, die über Internetprotokolle ausgetauscht werden."

Grundlage bilden die drei Standards: SOAP, WSDL und UDDI. Die folgenden Abschnitte basieren auf [Mel10].

SOAP Der Begriff SOAP beschreibt ein XML-basiertes Nachrichtenformat, das insbesondere zur Kommunikation mit und zwischen Web Services eingesetzt wird. Es ist unäbhängig von einem Transportprotokoll. 1999 wurde Version 1.0 veröffentlicht. Im Jahr 2003 wurde Version 1.2 als *Recommendation* veröffentlicht. Eine SOAP Nachricht ist ein XML-Dokument, das aus den folgenden drei Teilen besteht:

- SOAP Envelope
- SOAP Header
- SOAP Body

Der SOAP Envelope (Umschlag) dient als Container für die gesamte Nachricht. Er beinhaltet die beiden Teile Header und Body sowie Informationen über die verwendete SOAP Spezifikation.

Listing 2.6 Beispiel einer SOAP Nachricht [GHM+07]

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
  <env:Header>
    <n:alertcontrol xmlns:n="http://example.org/alertcontrol">
      <n:priority>1</n:priority>
      <n:expires>2001-06-22T14:00:00-05:00</n:expires>
    </n:alertcontrol>
  </env:Header>
  <env:Body>
    <m:alert xmlns:m="http://example.org/alert">
      <m:msg>Pick up Mary at school at 2pm</m:msg>
    </m:alert>
  </env:Body>
</env:Envelope>
```

Der erste Teil einer SOAP Nachricht ist der Header. Er ist optional und der Inhalt wird durch die SOAP Spezifikation nicht genau definiert. Er kann zusätzliche Informationen für den Empfänger beinhalten, oder aber auch, da Nachrichten meist über Zwischenstationen (Intermediäre) zum Empfänger gelangen, Informationen für Zwischenstationen beinhalten.

Für SOAP Header Elemente können zusätzliche Attribute angegeben werden. Das Attribut *role* spezifiziert die Rolle, in der sich der Empfänger bzw. die Zwischenstation befinden muss, um das Header Element verarbeiten zu dürfen. Das Attribut *mustUnderstand* kann *true* oder *false* sein. Ist es *true*, muss die Station, die die Nachricht empfangen hat, das Header Element verarbeiten können. Wenn die Station die Nachricht nicht verarbeiten kann, wird sie nicht weitergeleitet/weiterverarbeitet. Listing 2.6 zeigt eine SOAP Nachricht. Der SOAP Header enthält Informationen über die Dringlichkeit sowie Gültigkeit der Nachricht. Der zweite Teil einer SOAP Nachricht ist der Body. Der Body ist verpflichtend und enthält die eigentlichen Nutzdaten (payload), die wiederum ein wohlgeformtes XML-Dokument bilden müssen. Einzig der Prolog ist an dieser Stelle nicht erlaubt.

WSDL WSDL, die *Web Service Description Language*, existiert in Version 1.1 seit 2001. 2007 veröffentlichte das W3C Version 2.0. Mit Hilfe von WSDL können Schnittstellen von Web Services beschrieben werden. Nach [Mel10] werden Web Services in WSDL aus zwei Blickwinkeln betrachtet:

> „abstrakt auf der Ebene der Funktionalität und konkret auf der Ebene der technischen Details. Hierzu wird die Beschreibung der Funktionalität, die ein Web Service anbietet, von den technischen Details über den Ort und die Art und Weise, wie ein Dienst angeboten wird, getrennt."

Listing 2.7 zeigt den prinzipiellen Aufbau einer WSDL Beschreibung. Eine WSDL Beschreibung beinhaltet die Komponenten *Types*, *Interface (WSDL 1.1 portType)*, *Binding* und *Service*.

Listing 2.7 Komponenten einer WSDL Beschreibung [CMRW07]

```
<description
     targetNamespace="xs:anyURI" >
  <documentation />*
  [ <import /> | <include /> ]*
  <types />?
  [ <interface /> | <binding /> | <service /> ]*
</description>
```

Das Element Description bildet das Wurzelelement der WSDL Beschreibung und beinhaltet die gerade genannten Komponenten als Kindelemente. Die Schnittstelllenbeschreibung erfolgt im Abschnitt Interface. Ein Interface bietet eine Menge an Operationen an. Für jede Operation wird festgelegt, welche Nachrichten zwischen Dienstanbieter und Dienstnutzer ausgetauscht werden müssen. Die Definition der Nachrichten erfolgt über die im Abschnitt Types festgelegten Datentypen. Die Abschnitte Types und Interface beschreiben einen Web Service auf abstrakte Weise. Konkreter wird es im Abschnitt Binding. Dort wird festgelegt, welche Protokolle für die Kommunikation verwendet werden können. Im letzten Abschnitt, dem Abschnitt Service, wird definiert, wie der Web Service physikalisch erreicht werden kann, z.B. über eine URL. Weitere Informationen zu WSDL sind in [CMRW07] zu finden.

UDDI UDDI steht für *Universal Description Discovery & Integration*. Es handelt sich hierbei um einen Standard für den Aufbau eines Verzeichnisdiensts, der im Wesentlichen aus den White Pages, Yellow Pages und Green Pages besteht. Die White Pages enthalten Informationen

über die einzelnen Dienstanbieter, die ihre Dienste im Verzeichnisdienst registriert haben und werden deshalb oftmals mit einem Telefonbuch verglichen. Eine Kategorisierung der einzelnen Dienste bilden die Yellow Pages. Aus diesem Grund spricht man auch von einem Branchenverzeichnis. Die Green Pages beinhalten die Schnittstellenbeschreibungen der einzelnen Web Services.

Die in Abbildung 2.1 gezeigte Rollenverteilung kann auf Web Services übertragen werden. Der Anbieter eines Dienstes erstellt eine entsprechende WSDL Beschreibung für seinen Dienst und registriert diesen im Verzeichnisdienst, der z.B. über UDDI aufgebaut wird. Der Dienstnutzer sucht nach einem passenden Dienst und fordert, wenn dieser gefunden wurde, die WSDL-Beschreibung an. Er erhält dazu ein URI, die auf die eigentliche WSDL-Beschreibung verweist, und kann damit den gewünschten Dienst einbinden.

2.3. Workflow

Ein Workflow, die deutsche Übersetzung lautet Arbeitsablauf, beschreibt eine Reihe von Aktivitäten, die in einer bestimmten Reihenfolge ausgeführt werden müssen. Einzelne Komponenten werden unter dem Aspekt der Wiederverwendbarkeit zu größeren Komponenten zusammengesetzt. Der Vorteil des Komponenten-basierten Aufbaus ist die größere Flexibilität. Ändert sich ein definierter Ablauf, können die einzelnen Komponenten neu zusammengesetzt werden. Ziel ist es, einen definierten Workflow weitestgehend zu automatisieren. Bisher wurden Workflows meist im unternehmerischen Umfeld eingesetzt. Mittlerweile findet die Workflow-Technologie auch im wissenschaftlichen Umfeld Einsatz.

2.3.1. Grundlagen der Workflow-Technologie

Nach [LR99] beschreibt ein Prozessmodell die Struktur eines Prozesses bzw. Geschäftsprozesses in der realen Welt. Es beschreibt die einzelnen Schritte, die nötig sind, um einen Prozess abzuarbeiten. Aus diesem Modell können einzelne Instanzen gebildet werden, also konkrete Ausführungen des Prozessmodells. Nicht alle Teile eines Prozessmodells müssen computerunterstützt ausgeführt werden. Das Workflowmodell spezifiziert die Teile eines Prozessmodells, die auf einem Computer ausgeführt werden. Aus diesem Workflowmodell können wiederum Instanzen gebildet werden. In der Bankenbranche gibt es z.B. Prozessmodelle zur Überprüfung der Bonität eines Kunden. Soll die Bonität eines bestimmten Kunden überprüft werden, wird von diesem Prozessmodell eine Instanz gebildet. Werden alle Teile eines Prozessmodells auf einem Computer ausgeführt, werden die Begriffe Prozessmodell und Workflowmodell oftmals synonym verwendet. Abbildung 2.2 stellt die Beziehung zwischen den einzelnen Begriffen grafisch dar.

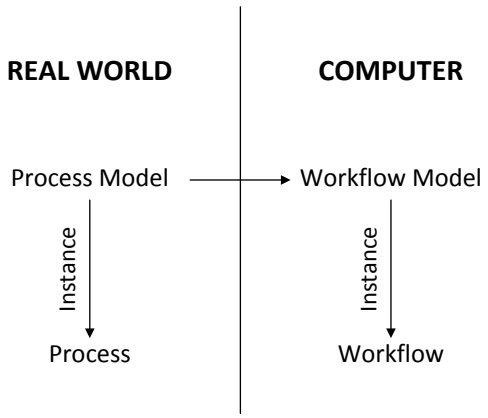

Abbildung 2.2.: Prozess und Workflow [LR99]

Prozesse werden nach [LR99] mithilfe der folgenden drei Dimensionen beschrieben: **WHAT**, **WHO** und **WITH**. Die **WHAT**-Dimension beschreibt die Aktivitäten, die ausgeführt werden müssen. Mithilfe der **WHO**-Dimension wird beschrieben, welcher Bereich einer Organisation (Abteilung, konkrete Person) die anfallende Aufgabe übernimmt. Die benötigten Ressourcen werden mit der **WITH**-Dimension beschrieben.

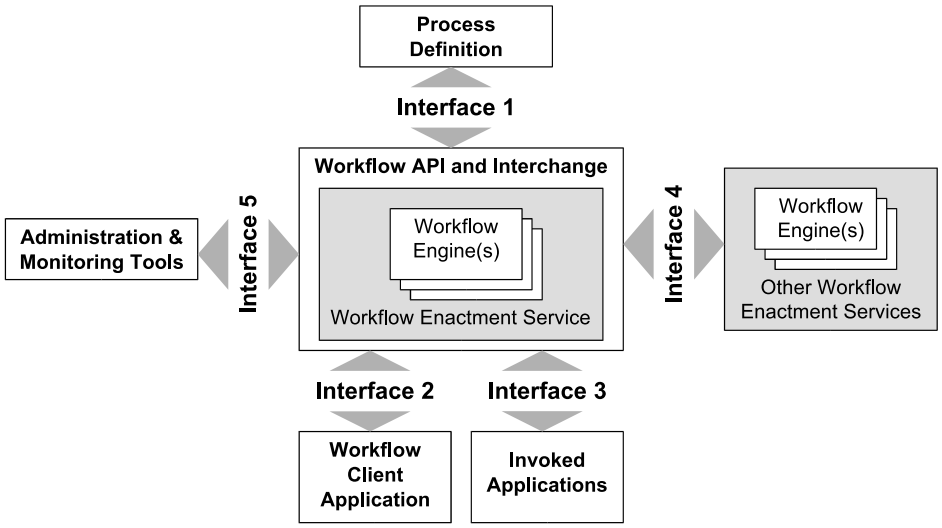

Abbildung 2.3.: WfMC-Workflow-Referenz-Modell [Mel10]

Das in Abbildung 2.3 dargestellte Worklow-Referenzmodell wurde von der Workflow Management Coalition (WfMC) entwickelt. Die WfMC wurde 1993 ins Leben gerufen und verfolgt das Ziel größtmögliche Systemunabhängigkeit und Interoperabilität zu schaffen. Die folgenden Abschnitte basieren auf [Mel10] und betrachten die einzelnen Teile des Referenzmodells genauer.

Process Definition Mit dieser Komponente werden die einzelnen Prozesse modelliert. Zur einfacheren Modellierung können grafische Anwendungsprogramme in Anspruch genommen werden.

Workflow Engine Die fertige Prozessdefinition wird von der Workflow Engine (Execution Engine) ausgeführt. Apache ODE [Foua] ist eine solche Engine und wird vom SIMPL-Rahmenwerk verwendet.

Workflow Client Application Die Workflow Client Application stellt dem Nutzer Funktionen zur Interaktion mit dem Workflow Management System bereit.

Invoked Applications Die Komponente Invoked Applications fasst alle Anwendungen bzw. deren Funktionen zusammen, die vom definierten Workflow eingebunden werden. Dadurch soll eine Unabhängigkeit vom Zielsystem realisiert werden.

Other Worflow Engines Mithilfe dieser Komponente kann mit anderen Workflow-Systemen kommuniziert werden. Somit können bereits vorhandene Prozesse in den eigenen Workflow integriert werden.

Administration & Monitoring Tool Diese Komponente überwacht und protokolliert die Ausführung der einzelnen Instanzen.

Mittlerweile wird die Workflow-Technologie auch in wissenschaftlichen Bereichen eingesetzt. In den letzten Jahren hat sich dafür der Begriff **Scientific Workflow** etabliert. Nach [GSK+11] bietet der Einsatz von Scientific Workflows die folgenden Vorteile:

- Die Wissensweitergabe kann durch Workflows unterstützt werden, da diese als Services bereit stehen.
- Resultate können durch die Community bewertet und analysiert werden.
- Workflows können mit großen Datenmengen umgehen.
- Eine Ausführung in verteilten und heterogenen Umgebungen ist möglich (Unterstützung verschiedener Plattformen und Programmiersprachen).
- Durch die Automatisierung der Schritte zwischen der Design- und Ausführungsphase können sich Wissenschaftler auf ihre eigentliche Problemstellung konzentrieren.
- Wissenschaftliche Simulationen können parallelisiert und automatisiert ausgeführt werden.

2.3.2. WS-BPEL

Die WS-Business Process Execution Language (WS-BPEL, ehemals BPEL4WS) [JE07] ist eine XML-basierte Sprache zur Beschreibung von Workflows. Mittlerweile existiert Version 2.0. Der Kerngedanke ist, dass Web Services in einer bestimmten Reihenfolge aufgerufen werden. Dabei nutzt WS-BPEL die XML-basierten Spezifikationen WSDL 1.1, XML Schema 1.0, XPath 1.0, XSLT 1.0 und Infoset. WS-BPEL übernimmt die Grundkonzepte von WSDL. [JE07] beschreibt WS-BPEL folgendermaßen:

> „The definition of a WS-BPEL business process follows the WSDL model of separation between the abstract message contents used by the business process and deployment information (messages and port type versus binding and address information). In particular, a WS-BPEL process represents all partners and interactions with these partners in terms of abstract WSDL interfaces (port types and operations); no references are made to the actual services used by a process instance. WS-BPEL does not make any assumptions about the WSDL binding. Constraints, ambiguities, provided or missing capabilities of WSDL bindings are out of scope of this specification.“

Listing 2.8 Erste Ebene eines WS-BPEL Dokuments (nach [Mel10])

```
<process name="ProcessName">
  <partnerLinks>..</partnerLinks>
  <variables>..</variables>
  <correlationSets>..</correlationSets>
  <faultHandlers>..</faultHandlers>
  <compensationHandlers>..</compensationHandlers>
  <eventHandlers>..</eventHandlers>
  <!-- genau eine activity -->
</process>
```

Listing 2.8 zeigt die erste Ebene eines WS-BPEL Dokuments. Einen genaueren Überblick über die Struktur der Sprache gibt Anhang A.1. Die folgende Beschreibung basiert auf [Mel10]. Wurzelelement ist immer das Element *Process*. Ein Prozess ruft während seiner Ausführung externe Dienste (Web Services) auf. Das Element *PartnerLinks* definiert, wie ein Prozess mit seinen Partnern kommuniziert. Mithilfe des Elementes *Variables* können Variablen definiert werden. Es stehen die Datentypen WSDL Message Type, XML Schema Type und XML Schema Element zur Verfügung. *CorrelationSets* enthalten verschiedene Eigenschaften, die von Nachrichten einer korrelierten Gruppe geteilt werden. Mithilfe der Ausprägungen dieser Eigenschaften kann bestimmt werden, welcher Partner oder welche Prozessinstanz eine Nachricht bekommt. Somit kann sicher gestellt werden, dass eine Nachricht auch zum richtigen Empfänger kommt. *FaultHandlers* bieten die Möglichkeit auf geworfene *Exceptions*, also Fehler während der Ausführung, zu reagieren. Bereits vollständig ausgeführte Aktivitäten können durch *CompensationHandlers* rückgängig gemacht bzw. kompensiert werden. Mithilfe von *EventHandlers* kann auf Ereignisse bzw. Alarme reagiert werden. Jeder Prozess muss genau eine Aktivität, die wiederum andere Aktivitäten beinhalten kann, beinhalten. Die zur Verfügung stehenden Aktivitäten können in zwei Kategorien unterteilt werden:

- Basisaktivitäten
- Strukturierte Aktivitäten

Im Folgenden werden die einzelnen Aktivitäten kurz erläutert. Grundlage bilden die Quellen [JE07] und [Mel10]. Für syntaktische Details wird auf diese beiden Quellen verwiesen.

Zu den Basisaktivitäten gehören: *receive*, *reply*, *invoke*, *assign*, *throw*, *exit*, *wait*, *empty*, *compensate*, *compensateScope*, *rethrow*, *validate* und *extensionActivity*. Eine *receive* Aktivität erlaubt es einem Prozess auf das Eintreffen einer Nachricht zu warten. Trifft eine Nachricht ein, so wird der Prozess gestartet bzw. weitergeschaltet. Durch die *reply* Aktivität kann der Prozess eine Nachricht an einen aufrufenden Partner zurückgeben (Listing 2.9).

Listing 2.9 Beispiel einer receive sowie reply Aktivität [Mel10]

```
<receive name="BerechnungReceive"
  partner="caller" portType="tns:BerechnungPT" operation="berechne"
  variable="request" createInstance="yes">
</receive>

<reply name="BerechnungReply"
  partner="caller" portType="tns:BerechnungPT" operation="berechne"
  variable="response">
</reply>
```

Durch eine *invoke* Aktivität können externe Web Services aufgerufen werden. Invoke kann sowohl für synchrone (request/response) als auch für asynchrone Aufrufe verwendet werden. Synchrone Aufrufe benötigen sowohl eine Input- als auch eine Output-Variable (Listing 2.10).

Listing 2.10 Beispiel einer synchronen invoke Aktivität [Mel10]

```
<invoke name="ErsterBerechnungsSchritt"
  partner="meinAddierer" portType="addiererNS:Addierer"
  operation="addiere"
  inputVariable="ersteSummanden" outputVariable="erstesErgebnis">
</invoke>
```

Eine *assign* Aktivität weist einer Variablen einen Wert zu bzw. kann eine Menge solcher Variablenzuweisungen beinhalten. Eine Fehler kann durch eine *throw* Aktivität ausgelöst werden. Existiert für diesen Fehler ein „FaultHandler“, wird eine Fehlerbehandlung gestartet. Dazu ist es notwendig einen eindeutigen Bezeichner für den Fehler anzugeben. Durch eine *exit* Aktivität kann die Ausführung eines Prozesses vollständig beendet werden. Die *wait* Aktivität hält einen Prozess für eine Weile an. Durch das Attribut „for“ wird eine entsprechende Zeitspanne festgelegt. Das Attribut „until“ ermöglicht es auf das Eintreten eines bestimmten Ereignisses zu warten. Die *empty* Aktivität entspricht der leeren Anweisung und kann z.B. als Platzhalter für noch nicht definierte Aktivitäten genutzt werden. Die *compensate* Aktivität setzt bzw. kompensiert fertiggestellte Aktivitäten. Dabei werden alle inneren *scopes* berücksichtigt. Eine *compensate* Aktivität darf nur in einem „FaultHandler“ oder „CompensationHandler“ stehen. *CompensateScope* ermöglicht es genau einen *scope* zu kompensieren. Durch *rethrow* kann eine ausgelöste und behandlelte Ausnahme erneut ausgelöst und somit weitergeleitet werden. Die Aktivität *validate* überprüft, ob Variablenwerte ihrer Definition genügen. Mithilfe des Elementes *extensionActivity* können neue Aktivitäten erzeugt werden.

Die strukturierten Aktivitäten umfassen: *sequence*, *if*, *while*, *repeatUntil*, *forEach*, *pick*, *flow* und *scope*. Aktivitäten innerhalb einer *sequence* Anweisung werden nacheinander in ihrer lexikalischen Anordnung ausgeführt. Die *switch* Anweisung wurde in WS-BPEL Version 2.0 durch eine *if* Anweisung ersetzt. Diese ermöglicht es, Bedingungen zu definieren, nach denen ein bestimmter Pfad im Workflow durchlaufen wird oder nicht. Durch eine *while* Anweisung werden Aktivitäten solange ausgeführt, solange eine definierte Bedingung erfüllt ist. Bei der *repeatUntil* Anweisung wird die definierte Bedingung im Gegensatz zur *while* Anweisung erst nach der Ausführung des ersten Schleifendurchlaufs geprüft. Die innere Aktivität wird somit mindestens einmal ausgeführt. Bei der *ForEach* Aktivität wird solange iteriert, bis eine bestimmte Anzahl an Schleifendurchläufen zu Ende ist. Die einzelnen Schleifendurchläufe können entweder parallel oder sequentiell ausgeführt werden. Wenn interne oder externe Ereignisse auftreten, bietet *pick* die Möglichkeit eine geeignete Aktivität zu starten. Eine *flow* Anweisung ermöglicht es Aktivitäten parallel oder nach einem bestimmten Ablaufgraphen auszuführen. Durch das Element *scope* können Gültigkeitsbereiche für z.B. Variablen und „PartnerLinks“ festgelegt werden.

2.4. Datenbanken

Die folgenden Absätze basieren auf [KE11]. Ein Datenbanksystem (DBS) dient zur Speicherung großer Mengen an Informationen (Daten) und wird heutzutage in allen großen Organisationen zur Informationsverwaltung eingesetzt. Es besteht aus zwei Komponten: Das Datenbankverwaltungssystem (DBVS) ist ein standardisertes Softwaresystem mit dessen Hilfe die Daten in einer Datenbank verwaltet und ausgewertet werden können. Die eigentlichen Daten befinden sich in der Datenbank, die oftmals auch als Datenbasis bezeichnet wird.

Für den Einsatz eines Datenbanksystems sprechen nach [KE11] unter anderem die folgenden Gründe: Bei der Verwendunng von Dateisystemen werden zu speichernde Informationen oftmals redundant, d.h. mehrfach, gespeichert. Datenbanken dagegen bieten einen gewissen Grad an *Redundanzfreiheit*. Des Weiteren können Dateien nur schwer miteinander verknüpft werden und es treten Probleme im Mehrbenutzerbetrieb auf. Dies kann zu ungewünschten Annomalien führen. Datenbanken hingegen bieten mächtige *Auswertungsoperationen* und einen kontrollierten *Mehrbenutzerbetrieb*. Ein weiteres Problem ist der mögliche Datenverlust. Bei der Verwendung von Dateisystemen kann im Fehlerfall nur schwer ein konsistenter Zustand wiederhergestellt werden. Datenbanken dagegen besitzen entsprechende *Recovery-Komponenten*. Zu den weiteren Vorteilen zählen unter anderem: *Daten- und Anwendungsunabhängigkeit*, die Festlegung von *Integritätsbedingungen* und die Möglichkeit einer *Zugriffskontrolle*.

Jedem Datenbanksystem liegt ein Datenmodell zu Grunde. Das Datenmodell legt die Modellierungskonstrukte fest, mit denen Datenobjekte erzeugt und verändert werden können. Im Rahmen dieser Arbeit werden relationale Datenbanken nur kurz erläutert. Das Augenmerk liegt auf XML-Datenbanken.

2.4.1. Relationale Datenbanken

Dieses Kapitel basiert auf [KE11]. Eine relationale Datenbank besteht aus Relationen. Aus mathematischer Sicht wird eine Relation folgendermaßen definiert:

> Gegeben seien n Domänen (Wertebereiche) D_1, $D_2 \ldots D_n$. Diese müssen nicht zwangsläufig unterschiedlich sein, dürfen jedoch nur atomare Werte beinhalten. Eine Relation R ist dann eine Teilmenge des kartesischen Produkts der n Domänen:
>
> $$R \subseteq D_1 \times D_2 \times \cdots \times D_n$$

Die n Domänen beschreiben das Schema einer Relation. Die aktuelle Ausprägung des Schemas ist durch die Teilmenge des Kreuzprodukts gegeben und wird auch als Instanz bezeichnet. Ein Element der Menge R heißt Tupel.

Student

MatrikelNr	Nachname	Vorname
001	Doe	John
002	Doe	Jane

Vorlesung

VorlesungsNr	Name	SWS
111	DB1	4
222	DB2	4

hatPruefung

VorlesungsNr	MatrikelNr	Note
111	001	1.0
222	001	1.7
222	002	1.3

Tabelle 2.1.: Relationale Datenbank: Datensätze innerhalb einer Tabelle

Bildlich gesehen besteht eine relationale Datenbank aus Tabellen. Tabelle 2.1 zeigt die drei Relationen *Student*, *Vorlesung* und *hatPruefung*. Die einzelnen Spalten repräsentieren die Attribute. Eine Zeile entspricht einem Tupel. Die Relation *Student* hat die drei Attribute *MatrikelNr*, *Nachname* und *Vorname*, die z.B. vom Typ *integer*, *string* und *string* sein können. Jede Relation hat einen *Primärschlüssel*. Dieser muss eindeutig sein, darf keine *NULL*-Werte beinhalten und wird typischerweise durch Unterstreichen gekennzeichnet. Attribute bzw. Attributkombinationen, die auf eine andere Relation verweisen, werden Fremdschlüssel genannt. In obiger Tabelle ist das Attribut *VorlesungsNr* der Releation *hatPruefung* ein Fremdschlüssel. Es verweist auf die Relation *Vorlesung* und stellt die entsprechenden Tupel beider Relationen in Beziehung.

Um Informationen aus einer Datenbank zu extrahieren, werden geeignete Sprachen benötigt. Für relationale Datenbanken gibt es zwei formale Abfragesprachen: die relationale Algebra sowie das Relationenkalkül. SQL ist eine *deklarative* Abfragesprache und baut auf diesen beiden Sprachen auf. Deklarativ bedeutet, dass der Nutzer nur definiert, **welche** Daten er haben möchte. Die eigentliche Auswertung übernimmt das Datenbanksystem.

2.4.2. XML-Datenbanken

In Abschnitt 2.1 wird auf XML eingegangen. Oftmals ist es nötig XML-Dokumente, z.B. aus rechtlichen Gründen, dauerhaft zu speichern. XML-Datenbanksysteme unterstützen diese Aufgabe. [Sch03] beschreibt drei Grundmodelle: Beim **monolithischem Ansatz** repräsentiert ein einziges XML-Dokument die gesamte Datenbank. Indem dieses *eine* Dokument verändert wird, werden Informationen hinzugefügt bzw. gelöscht. Dadurch ist die Datenbank vollständig geordnet. Probleme gibt es, wenn z.B. ganze Dokumente gespeichert werden sollen. So ist es nicht möglich einen Prolog oder eine Document Type Definition unterhalb eines Wurzelelementes zu speichern. Der **fragmentorientierte Ansatz** wird von [Sch03] folgendermaßen beschrieben:

> „Bei diesem Ansatz besteht die Datenbank aus Daten, die zunächst einmal (aus XML Sicht) nicht miteinander verknüpft sind. Vielmehr werden Daten hier erst beim Extrahieren aus der Datenbank zu XML-Fragmenten kombiniert. Analog werden XML-Fragmente (ggf. ganze XML-Dokumente) beim Abspeichern möglicherweise aufgelöst. Die Einheit der Operationen wird also im Allgemeinen dynamisch definiert."

Relationale Datenbanken, die die Speicherung von XML-Dokumenten unterstützen, verfolgen oftmals diesen Ansatz. Problematisch ist, dass der eigentliche Zusammenhang der einzelnen Teile innerhalb der Datenbank nicht sichtbar ist. Beim **dokumentorientierten Ansatz** besteht die Datenbank aus einzelnen XML-Dokumenten. Es werden immer ganze Dokumente eingefügt bzw. gelöscht. Mehrere Dokumente können in sogenannten *Collections* zusammengefasst werden. Dies kann automatisch aufgrund festgelegter Regeln erfolgen oder manuell durch den Nutzer.

Relationale Datenbanksysteme, wie z.B. Mircosoft SQL Server, die mit XML-Dokumenten umgehen können, werden als XML-enabled Datenbanksysteme bezeichnet. Native XML-Datenbanksysteme, wie z.B. MonetDB/XQuery oder Exist, verfolgen den dokumentorientierten Ansatz.

Neben allgemeinen Anforderungen, wie z.B. Effizienz, Mehrbenutzerbetrieb, Skalierbarkeit und Verfügbarkeit, die auch für andere Datenbanksysteme gelten, müssen XML-Datenbanksysteme noch spezielle Anforderungen erfüllen. [Sch03] nennt unter anderem die Folgenden:

- Standardkonformität: Ein XML-Datenbanksystem sollte die wichtigsten XML-Spezifikationen und Empfehlungen berücksichtigen.
- Dokumentenbehandlung: Es sollte möglich sein *ganze* XML-Dokumente (inklusive Prolog und Document Type Definition) zu speichern und diese auch unverändert wiederzugeben.
- Schema-Unabhängigkeit: Es sollte möglich sein beliebige wohlgeformte XML-Dokumente zu speichern. Wenn eine schematische Beschreibung gegeben ist, muss deren Übereinstimmung geprüft werden.

- Schemavorgabe: Es sollte möglich sein schematische Beschreibungen vorzugeben. Gegen diese werden zu speichernde XML-Dokumente dann validiert.
- Kodierung und Internationalisierung: Da XML-Dokumente insbesondere zum Datenaustausch verwendet werden, sollten möglichst viele Kodierungen verstanden werden.
- XML-spezifische Schnittstelle: Eine formale Abfragesprache sollte unterstützt werden und das Ergebnis wiederum wohlgeformtes XML sein.
- Originaltreue: XML-Dokumente sollten unverändert zurückgegeben werden.

2.4.3. XPath

Um Informationen aus einer XML-Datenbank zu extrahieren, wird eine ausdrucksstarke, deklarative Abfragesprache benötigt. XPath ist eine vom World Wide Web Consortium (W3C) standardisierte Sprache und dient zur Adressierung von Teilen von XML-Dokumenten. Dieser Abschnitt basiert auf [KE11] und [Sch03].

Ein XML-Dokument wird als Baum dargestellt. Der Wurzelknoten ist ein virtueller Knoten und kommt im eigentlichen XML-Dokument nicht vor. XPath unterstützt die folgenden Knotentypen: Elementknoten, Attributknoten, Namensraumknoten, Verarbeitungsanweisungsknoten, Kommentarknoten und Textknoten. Mit Hilfe von *Lokalisierungspfaden* kann durch ein XML-Dokument navigiert werden. Ein Lokalisierungspfad besteht aus aneinandergereihten *Lokalisierungsschritten*, welche durch ein „/“-Zeichen getrennt werden. Eine Knotenmenge wird ausgehend von einem Referenzknoten, durch einen Lokalisierungsschritt selektiert. Im nächsten Schritt dient jeder dieser selektierten Knoten wiederum als Referenzknoten. Die auf diese Weise selektierten Knotenmengen werden vereinigt und bilden die Ergebnismenge. Ein Lokalisierungsschritt besteht aus drei Teilen:

axis::node-test[predicate]

Die *Achsen* geben die Richtung vor, in der durch die Baumstruktur navigiert wird:

- self: der Referenzknoten selbst
- attribute: die Attribute des Referenzknotens
- parent: der Vaterknoten des Referenzknotens
- child: alle direkt untergeordneten Knoten
- descendant: alle untergeordneten Knoten
- descendant-or-self: alle untergeordneten Knoten inkl. Referenzknoten
- ancester: alle übergeordneten Knoten
- ancester-or-self: alle übergeordneten Knoten inkl. Referenzknoten
- following: alle Knoten, die nach dem Referenzknoten im XML Dokument vorkommen (außer direkte und indirekte Nachfolger)

- following-sibling: alle Knoten, die nach dem Referenzknoten im XML Dokument vorkommen und in der gleichen Hierachie-Ebene liegen
- preceding: alle Knoten, die vor dem Referenzknoten im XML Dokument vorkommen (außer direkte und indirekte Vorgänger)
- preceding-sibling: alle Knoten, die vor dem Referenzknoten im XML Dokument vorkommen und in der gleichen Hierachie-Ebene liegen

Mit Hilfe des *Knotentests* kann die Achse eingeschränkt werden. Nur Knoten, die die definierte Bedingung erfüllen, dienen wiederum als Referenzknoten. Komplexere Bedingungen können durch *Prädikate* definiert werden.

Listing 2.11 Ausschnitt aus einem XML-Dokument

```
<Universitaet>
  <Fakultaeten>
    <Falutaet>
      <NameDerFakultaet>Informatik</NameDerFakultaet>
      <Vorlesungen>
        <Vorlesung Vorlesungsnummer=111>
          <NameDerVorlesung>DB1</NameDerVorlesung>
          <SWS>4</SWS>
        </Vorlesung>
      </Vorlesungen>
    </Falutaet>
    <Falutaet>
      <NameDerFakultaet>Mathematik</NameDerFakultaet>
      <Vorlesungen>
        <Vorlesung Vorlesungsnummer=222>
          <NameDerVorlesung>HM1</NameDerVorlesung>
          <SWS>4</SWS>
        </Vorlesung>
      </Vorlesungen>
    </Falutaet>
  </Fakultaeten>
</Universitaet>
```

Listing 2.11 zeigt ein XML-Fragment. Für dieses Fragement werden im Folgenden exemplarisch XPath Ausdrücke und deren Resultate gezeigt. So liefert der Ausdruck:
/child::Universitaet/child::Fakultaeten/child::Fakultaet/child::NameDerFakultaet

das folgende Ergebnis:

<NameDerFakultaet>Informatik</NameDerFakultaet>
<NameDerFakultaet>Mathematik</NameDerFakultaet>

Mit Hilfe eines Prädikats können alle Vorlesungen der Fakultät Informatik extrahiert werden:

/child::Universitaet/child::Fakultaeten/child::Fakultaet[child::NameDerFakultaet= "Informatik"]/ descendant-or-self::Vorlesung/child::NameDerVorlesung

Das Ergebnis wäre in diesem Fall:

<NameDerVorlesung>DB1</NameDerVorlesung>

2.4.4. XQuery

Die gerade beschriebene Sprache XPath bildet die Grundlage von XQuery. Eine Vielzahl an XML-Datenbanksystemen unterstützt XQuery als Abfragesprache. Dieser Abschnitt basiert auf [KE11] und [Sch03]. In XQuery werden Anfragen als *FLWOR*-Ausdrücke formuliert. FLWOR ist ein Apronym für:„**f**or ... **l**et ... **w**here ... **o**rder by ... **r**eturn ... “. Mit *for* werden Variablen sukzessive, also iterativ, gebunden. Für jede durchgeführte Variablenbindung im *for*-Konstrukt, wird im *let*-Konstrukt eine einmalige Bindung durchgeführt. Das *where*-Konstrukt bietet die Möglichkeit Bedingungen zu definieren, mit denen bestimmte Knoten extrahiert werden können. Eine Sortierung kann mit dem *order-by*-Konstrukt realisiert werden. Das *return*-Konstrukt wird nur für diejenigen Knoten ausgewertet, für die die im *where*-Konstrukt definierten Bedingungen erfüllt sind. FLWOR-Ausdrücke können beliebig tief geschachtelt werden und in einem XML-Fragment vorkommen. Auszuwertende XQuery Ausdrücke werden durch geschweifte Klammern kenntlich gemacht. Des Weiteren ist es möglich Joins sowie rekursive Anfragen zu realisieren. Der folgende XQuery Ausdruck würde in Bezug auf obiges XML-Dokument alle Vorlesungen extrahieren:

```
<Vorlesungen>
   {for $v in doc(“example.xml“)//Vorlesungen
   return $v}
</Vorlesungen>
```

2.5. Sensornetze

Sensornetze (von engl. Wireless Sensor Networks) sind eine neuartige Technologie, die nach [ZJ09] unter anderem in den verschiedensten zivilen und militärischen Bereichen eingesetzt werden kann. Ein Sensornetz besteht aus einer Vielzahl an Sensorknoten. Diese können Daten erfassen und drahtlos untereinander austauschen sowie einem Client zur Verfügung stellen. Im Folgenden werden zunächst die Grundlagen von Sensornetzen erläutert, bevor anschließend auf TinyDB, eine deklarative Datenbank für Sensornetze, eingegangen wird.

2.5.1. Grundlagen von Sensornetzen

Dieser Abschnitt basiert auf [AV10] und [ZJ09]. Eine Vielzahl an Sensorknoten gleicher Bauart bildet ein Sensornetz. Ein Sensorknoten weißt unter anderem die folgenden Merkmale auf: er ist relativ klein, effizient in der Energienutzung, kostengünstig und robust gegenüber Umwelteinflüssen. In der Regel besteht ein Sensorknoten aus einer Sende- und Empfangseinheit, einer Batterie, einem Prozessor (Mikrocontroller und Speicher) sowie

aus verschiedenen Sensoren, die auf bestimmte Ereignisse reagieren. So können z.B. die folgenden physikalischen Parameter gemessen bzw. registriert werden:

- Lichtverhältnisse
- Geräuschpegel
- Luftdruck
- Temperatur
- Luft- und Wasserqualität
- Größe, Gewicht, Position und Geschwindigkeit von Objekten

Die erfassten Daten werden vom Sensorknoten vorverarbeitet und anschließend an andere Knoten weitergeleitet. Oftmals werden die erfassten Daten zu einer Basisstation geleitet, die mit dem Sensornetz verbunden ist und eine Schnittstelle für den Nutzer bietet. Die drahtlose Kommunikation zwischen den verschiedenen Sensorknoten bietet signifikante Vorteile gegenüber einer kabelgebundenen Variante. So können Sensorknoten einfacher und kostengünstiger installiert werden, sowie an Orten plaziert werden, wo vorher keine Installation möglich war. Es ergeben sich aber auch neue Probleme. Der kritische Punkt ist die Energieversorung der einzelnen Knoten. Daten müssen unter anderem effizient transformiert und übertragen werden, um Energie zu sparen. Es wird zwischen Low-End- und High-End-Plattformen unterschieden. Die Sensorknoten der Mica-Familie (Mica, Mica2 und MicaZ) [AV10] gehören zur Klasse der Low-End-Plattformen und werden von TinyDB unterstützt. Low-End-Plattformen zeichnen sich durch limitierende Eigenschaften, wie z.B. geringe Speicherkapazität und Prozessorleistung, aus. High-End-Plattformen beseitigen diese limitierenden Eigenschaften und können deshalb auch komplexere Aufgaben, wie z.B. das Netzwerk-Management, erledigen.

2.5.2. TinyDB

Dieser Abschnitt basiert auf [MHH03]. TinyDB läuft innerhalb von TinyOS, einem Softwaresystem zur Verwaltung von drahtlosen Sensornetzen, und bietet die Möglichkeit Daten aus einem Sensornetz zu extrahieren. TinyDB funktioniert standardmäßig mit den von der Universität Berkeley entwickelten Sensorknoten. Mithilfe von TinySQL, einer deklarativen Abfragesprache, können Daten extrahiert werden. Das folgende Datenmodell liegt TinyDB bzw. TinySQL zu Grunde: Alle Queries werden auf einer einzigen unendlichen Tabelle ausgeführt, die das gesamte Netzwerk repräsentiert. Die einzelnen Spalten repräsentieren die definierten Attribute. Zu den Attributen zählen Attribute für die eigentlichen Sensorwerte, die KnotenID sowie andere Eigenschaften, die den Zustand eines Sensorknotens beschreiben. Nicht vorhandene Werte werden durch einen *NULL*-Eintrag gekennzeichnet. Tabelle 2.2 zeigt den Aufbau einer TinySQL Query.

TinySQL und SQL weisen, wie der Name schon andeutet, Gemeinsamkeiten auf. Zu den Gemeinsamkeiten zählen die folgenden Punkte. Bei beiden Sprachen handelt es sich um *deklarative* Abfragesprachen. Die **GROUP BY**-Klausel ist optional. Bei Verwendung der

GROUP BY-Klausel ist die **HAVING**-Klausel, ebenfalls wie bei SQL, optional. Des Weiteren sind arithmetische Ausdrücke in allen Klauseln erlaubt. Zu den Unterschieden zählt, dass in der optionalen **FROM**-Klausel nur die Tabelle *sensors*, die oben schon erläutert wurde, referenziert werden kann. Des Weiteren werden Subqueries nicht unterstützt und Spalten können nicht umbenannt werden. Arithmetische Ausdrücke können nur der Form *column op constant* sein, wobei $op \in \{+, -, *, /\}$.

SELECT *select-list*
[FROM sensors]
WHERE *where-clause*
[GROUP BY *gb-list*
[HAVING *having-list*]]
[TRIGGER ACTION *command-name* [(*param*)]]
[EPOCH DURATION *integer*]

Tabelle 2.2.: Aufbau einer TinySQL Anweisung [MHH03]

Epoch	nodeid	light
3	12	860
4	12	860
5	12	861
8	12	860
9	12	879
11	12	860

Tabelle 2.3.: Beispiel Query: SELECT nodeid, light EPOCH DURATION 1024 [MHH03]

Die **SELECT**-Klausel wählt die Spalten aus, die extrahiert werden soll. Die **FROM**-Klausel referenziert die Tabelle *sensors*. Eine Suchbedingung kann mithilfe der **WHERE**-Klausel spezifiziert werden. Die **GROUP BY**-Klausel ermöglicht es, die resultierende Tabelle zu sortieren. Z.B. werden alle Zeilen, die in einer Spalte den gleichen Wert enthalten, zu einer Gruppe zusammengefasst. Mithilfe der **HAVING**-Klausel können weitere Zeilen aussortiert werden. Mithilfe der **TRIGGER**-Klausel kann eine Aktion ausgeführt werden. Sobald die in der **WHERE**-Klausel definierte Suchbedingung erfüllt ist, wird die spezifizierte Aktion ausgeführt. Beispielsweise kann ein Alarm ausgelöst werden, wenn ein Sensor eine bestimmte Temperatur misst bzw. überschreitet. Das Ergebnis einer Abfrage zeigt den Zustand zu einem bestimmten Zeitpunkt. Mithilfe von **EPOCH DURATION** kann das Intervall spezifiziert werden, in dem die Abfrage erneut evaluiert wird. Tabelle 2.3 zeigt exemplarisch eine mögliche Ergebnistabelle. Komplexe Anfragen können auf unterschiedliche Arten ausgeführt bzw. ausgewertet werden. Der Anfrageoptimierer wählt stets den Plan, der den Energieverbrauch minimiert, da dies der kritische Punkt in einem Sensornetzwerk ist.

3. SIMPL-Rahmenwerk

SIMPL (SimTech - Information Management, Processes and Languages) ist ein erweiterbares Rahmenwerk zur Einbindung externer, heterogener Datenquellen in Simulationsworkflows. Im Folgenden werden zunächst bestehende Probleme in Simulationsworkflows erläutert, bevor anschließend das SIMPL-Rahmenwerk detailliert betrachtet wird. Im weiteren Verlauf dieser Arbeit steht die Abkürzung *DM* für *Datenmanagement* (von engl. data management). Dieses Kapitel basiert auf [RRS+11].

3.1. Gründe für SIMPL

Mithilfe des PANDAS Rahmenwerks[1] (Porous Media Adaptive Nonlinear finite element solver based on Differential Algebraic Systems) können Knochenverformungen simuliert werden. Solche Simulationsabläufe werden oftmals als Simulationsworkflows modelliert. Innerhalb dieser Workflows werden Daten bereitgestellt und externen Programmen übergeben. Diese führen dann Berechnungen auf den Datensätzen durch. Die benötigten Programme werden als Web Services bereitgestellt und durch das Scientific Workflow Management System (sWfMS) entsprechend aufgerufen. Charakteristisch für solche Simulationen bzw. Simulationsabläufe sind komplexe mathematische Berechnungen sowie verschiedene Aufgaben im Bereich der Datenverwaltung und Datenbereitstellung. Die Vorteile der Workflow-Technologie werden in Abschnitt 2.3 erläutert. Problematisch ist jedoch, dass sehr große Datenmengen aus verschiedenen Quellen verarbeitet werden müssen, die oftmals in proprietären Formaten vorliegen. Diese müssen dann in ein für die Simulation geeignetes Format transformiert werden. Ein sWfMS bietet meist keine integrierte Möglichkeit solche heterogenen Datenquellen zu verwalten. Oftmals muss auf spezielle Dienste, die auf eine einzige Datenquelle und/oder ein einzelnes Simulationsprogramm zugeschnitten sind, zurückgegriffen werden. Dies führt zu einem erhöhten Arbeitsaufwand für den Anwender. Dieser muss unter anderem adäquate Quellen finden sowie nötige Datentransformationen implementieren. Gerade bei komplexen Simulationen, die eine Vielzahl an heterogenen Datenquellen nutzen, führt dies aufgrund der enormen Komplexität zu Problemen.

Wünschenswert wäre eine Abstraktionsebene innerhalb eines sWfMS, welche den Zugriff auf externe Datenquellen sowie deren Verwaltung vereinheitlicht und somit auch vereinfacht. An dieser Stelle setzt das SIMPL-Rahmenwerk an. SIMPL bietet einheitliche Zugriffsmethoden um, aus Simulationsworkflows heraus, auf beliebige externe Datenquellen zuzugreifen.

[1] http://www.get-pandas.com/

Metadaten beschreiben die hierfür notwendigen Zuordnungen zwischen vereinheitlichten Zugriffsmethoden und konkreten Zugriffsoperationen. Dem Modellierer stehen dafür zusätzliche Workflowaktivitäten zur Verfügung (vgl. Abschnitt 3.3). Zusätzlich beinhaltet das SIMPL-Rahmenwerk Datenmanagement Patterns, um z.B. ETL Operationen einfacher definieren zu können (vgl. Abschnitt 3.4).

3.2. Architektur des SIMPL-Rahmenwerks

Das SIMPL-Rahmenwerk erweitert ein sWfMS. Aus diesem Grund wird nun kurz auf die Architektur einen solchen Systems eingegangen. Die in Abbildung 3.1 dargestellte Architektur basiert auf der in [LR99] definierten Technologie für Geschäfts- und Produktionsworkflows und unterscheidet zwischen GUI Komponenten und Laufzeitkomponenten. Im Folgenden werden nun zunächst die GUI Komponenten erläutert:

- Der **Function Catalog** enthält eine Liste der verfügbaren Dienste oder vorgefertigten Workflow-Fragmente, die innerhalb eines Workflows genutzt werden können.
- Der **sWF Modeler** (Scientific Workflow Modeler) unterstützt den Modellierer dabei Workflow-Spezifikationen sowie Deployment-Informationen zu erstellen.
- Die **Monitor** Komponente bietet die Möglichkeit die Ausführung von Workflows zu überwachen.
- Die **Result Display** Komponente stellt dem Nutzer Zwischen- und Endergebnisse der Simulationsberechnungen zur Verfügung.

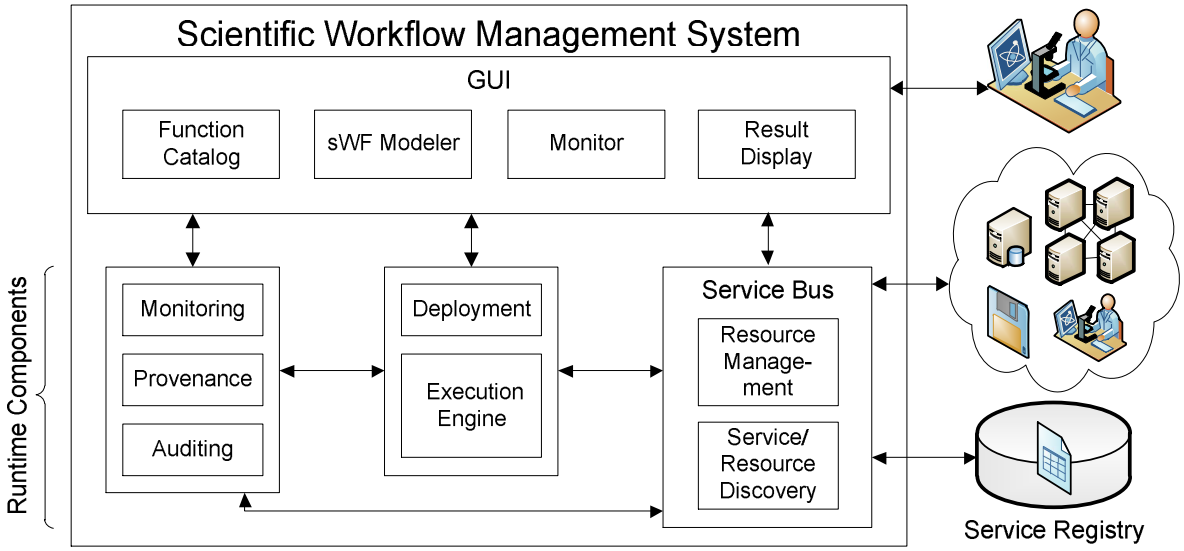

Abbildung 3.1.: Architektur eines sWfMS [GSK+11]

Zu den Laufzeitkomponenten zählen die Folgenden:

- Die **Deployment** Komponente transformiert ein Workflowmodell in eine für die **Execution Engine** verständliche Form, die dann Instanzen davon ausführt.
- Die **Auditing** Komponente zeichnet Ereignisse, die zur Laufzeit auftreten, auf.
- Die **Monitoring** Komponente nutzt die Ergebnisse der Auditing Komponente, um den Zustand eines laufenden Workflows anzuzeigen.
- Die **Provenance** Komponente erfasst weitere Daten, um detailliertere Informationen über die Ausführung eines Workflows zur Verfügung zu stellen und damit die Nachvollziehbarkeit und Wiederholbarkeit solcher Ausführungen sicher zu stellen.
- Die **Service Bus** Komponente sucht und wählt passende Dienste aus, die die Workflow Aktivitäten implementieren. Des Weiteren leitet sie Nachrichten weiter und transformiert Daten.
- Das **Resource Management** enthält Informationen über externe Ressourcen sowie Dienste.
- Die **Service/Resource Discovery** Komponente nutzt die im Resource Management hinterlegten Metadaten, um eine Liste mit zu den Nutzeranforderungen passenden Diensten bzw. Ressourcen zur Verfügung zu stellen.

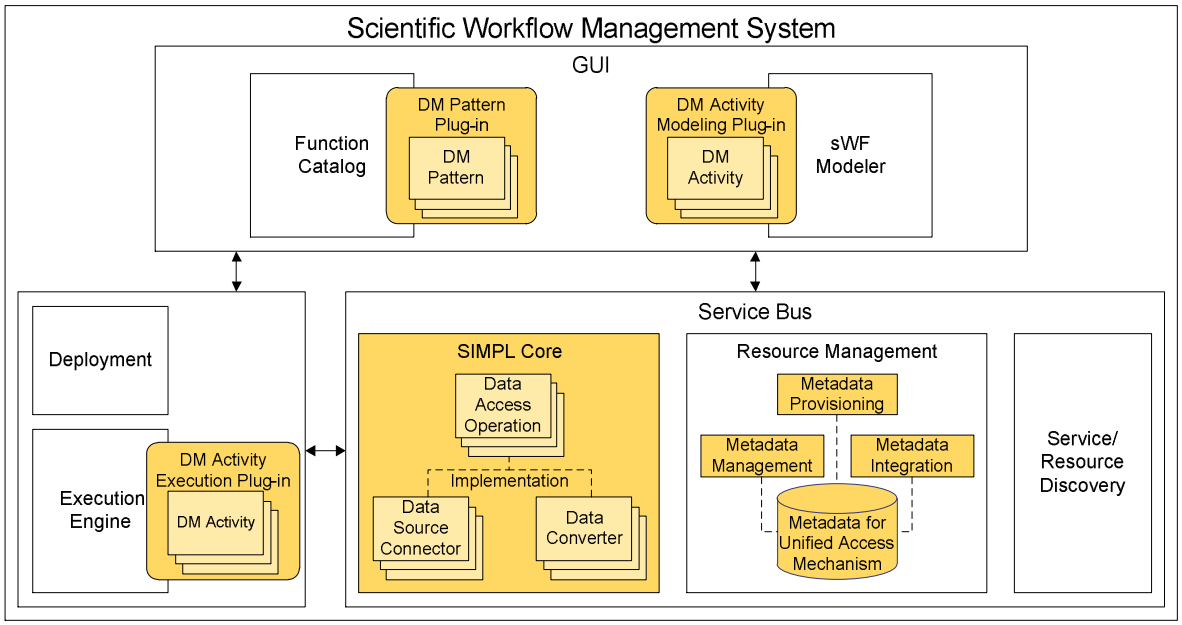

Abbildung 3.2.: Das SIMPL-Rahmenwerk eingebettet in ein sWfMS [RRS+11]

Das SIMPL-Rahmenwerk erweitert nun diese Architektur um zusätzliche Komponenten. Abbildung 3.2 zeigt die geänderte Architektur. Die **SIMPL Core** Komponente wurde im Service Bus eingebettet und bietet vereinheitlichte logische Schnittstellen, um auf beliebige externe Datenquellen zuzugreifen. Die **Resource Management** Komponente wurde dahingehend erweitert, dass Metadaten über die Beziehung zwischen logischen Schnittstellen und konkreten Zugriffsmechanismen gespeichert werden können. Das **DM Activity Modeling Plug-in** erweitert den sWF Modeller um zusätzliche Aktivitäten. Dem Modellierer stehen somit alle Aktivitäten aus BPEL-DM (vgl. Abschnitt 3.3) zur Verfügung. Das **DM Activity Execution Plug-in** erweitert die Execution Engine und sorgt dafür, dass die neu eingeführten Aktivitäten auch zur Laufzeit ausgeführt werden können. Das **DM Pattern Plug-in** erweitert den Function Catalog und unterstützt den Nutzer dahingehend, dass dieser vorgefertigte Patterns, welche komplexe Operationen darstellen, übernehmen kann (vgl. Abschnitt 3.4).

3.3. BPEL-DM

In Abschnitt 2.3.2 wurde bereits auf BPEL, die Business Process Execution Language, eingegangen. BPEL-DM (Business Process Execution Language extension for Data Management) erweitert BPEL um zusätzliche Aktivitäten, die sogenannten DM Aktivitäten. Diese sind:

- IssueCommand
- RetrieveData
- WriteDataBack

Jede dieser Workflow Aktivitäten ruft die entsprechende Operation des SIMPL Cores auf. Dieser ist wiederum für die Ausführung der eigentlichen Anweisung auf der Datenquelle verantwortlich. Im weiteren Verlauf dieser Arbeit bezeichnet der Begriff *Datenquelle* ein System, welches Daten speichern und verwalten kann. Dabei kann es sich um Datenbanken oder aber auch um Dateisysteme handeln. Die eigentlichen Anweisungen, die auf der Datenquelle ausgeführt werden, werden *DM commands* genannt. Für relationale Datenbanken wären dies z.B. SQL Konstrukte.

Die neu definierten DM Aktivitäten erhalten als Eingabeparameter eine spezielle BPEL Variable. Diese Variable referenziert die Datenquelle auf der die eigentliche Anweisung ausgeführt werden soll, und wird *data source reference variable* genannt. Solch eine Referenz ist ein *logical data source descriptor*, der entweder den logischen Namen der Datenquelle oder eine Anforderungsbeschreibung mit funktionalen und nicht funktionalen Anforderungen beinhaltet. Der logische Name einer Datenquelle referenziert genau eine Datenquelle, deren Beschreibung im Resource Management hinterlegt ist. Mithilfe einer Anforderungsbeschreibungen ist es möglich, eine geeignete Datenquelle erst zur Laufzeit zu suchen und einzubinden.

Jede Datenquelle verwaltet mehrere Container (*datacontainer*). In einer relationalen Datenbank wären dies z.B. die einzelnen Tabellen. Mithilfe von *data container reference variables* können einzelne Container über logische Namen referenziert werden. Das Resource Management speichert die dafür notwendigen Zuordnungen zwischen logischem Namen und konkretem

Identifikator. Des Weiteren benötigt das SIMPL-Rahmenwerk noch *data set variables*. Diese Variablen dienen als Zielcontainer, um Daten in einen Workflow zu laden. Die Struktur dieser Variablen wird durch XML Schema Definitionen beschrieben. Dabei müssen die einzelnen Definitonen die Besonderheiten der unterschiedlichen Datenquellen berücksichtigen. Tabellenbasierte Daten, wie sie in relationalen Datenbanken oder CSV-basierten Dateien vorkommen, werden z.B. in einer XML RowSet Struktur gespeichert.

Im Folgenden werden nun die einzelnen DM Aktivitäten im Detail beschrieben. Die *IssueCommand* Aktivität bietet die Möglichkeit Daten zu manipulieren sowie Datenstrukturen, wie z.B. die Struktur einer Datenbanktabelle, zu erstellen. Als Eingabeparameter erwartet diese Aktivität neben einer Referenz auf eine Datenquelle noch einen *DM command*. Die spezifizierte Anweisung wird auf der gewählten Datenquelle ausgeführt. Die Execution Engine erwartet eine Bestätigung über den Erfolg bzw. Misserfolg der Operation. Bei einem Misserfolg kann eine Fehlerbehandlung bzw. Fehlerkompensation gestartet werden. Im Gegenzug wird bei einem Erfolg der Workflow weiter ausgeführt.

Die *RetrieveData* Aktivität erhält genau wie die IssueCommand Aktivität einen *DM command*. Diese Anweisung muss Daten generieren. Bei einer erfolgreichen Ausführung der Anweisung, werden die generierten Daten an die Execution Engine zurückgeschickt. Zusätzlich wird bei dieser Aktivität eine *data set variable* angegeben. Die generierten Daten werden dann in dieser Variable gespeichert. Im Fehlerfall kann wiederum eine entsprechende Fehlerbehandlung gestartet werden.

Die *WriteDataBack* Aktivität ist das Gegenstück der RetrieveData Aktivität. Daten aus dem Workflow Kontext können in einer Datenquelle gespeichert werden. Als Eingabeparameter werden dazu eine *data set variable* sowie eine *data container reference variable* benötigt. Die durch die erste Variable spezifizierten Daten werden im Container, der durch die zweite Variable referenziert wird, gespeichert. Die Execution Engine erwartet analog zur IssueCommand Aktivität eine Benachrichtigung über den Erfolg bzw. Misserfolg der Ausführung.

Des Weiteren bietet das SIMPL-Rahmenwerk die Möglichkeit *data container reference variables* als Parameter in *DM commands* einzubauen. So können diese z.B. in der FROM Klausel einer SQL Anweisung genutzt werden. Dasselbe gilt für BPEL Variablen. Variablen vom Type String oder Integer können z.B. in Vergleichsoperationen eingebunden werden. Um eine Variable innerhalb einer Anweisung kenntlich zu machen, wird der Variablenname mit „#"-Zeichen umschlossen. Die Execution Engine erkennt diese Kennzeichnung und löst sie auf. Der Variablenwert wird gelesen und an entsprechender Stelle eingefügt. Bei *data container reference variables* ist das der logische Name des referenzierten Containers.

3.4. Data Management Patterns

Wie bereits in Abschnitt 3.2 erwähnt, können *DM Patterns* bei der Modellierung verwendet werden. Dabei handelt es sich um vorgefertigte Datenmanagement-Operationen, die nur noch parametrisiert werden müssen. Dazu wurde der Function Catalog des sWfMS um das *DM Pattern Plug-in* erweitert, welches eine Liste der verfügbaren Patterns zur Verfügung

stellt. Dadurch wird eine neue Abstraktionsebene geschaffen. Der Modellierer wird bei der Konkretisierung eines gewählten Patterns in einem semi-automatischen Prozess unterstützt und muss keine konkreten Datenmanagement-Operationen definieren.

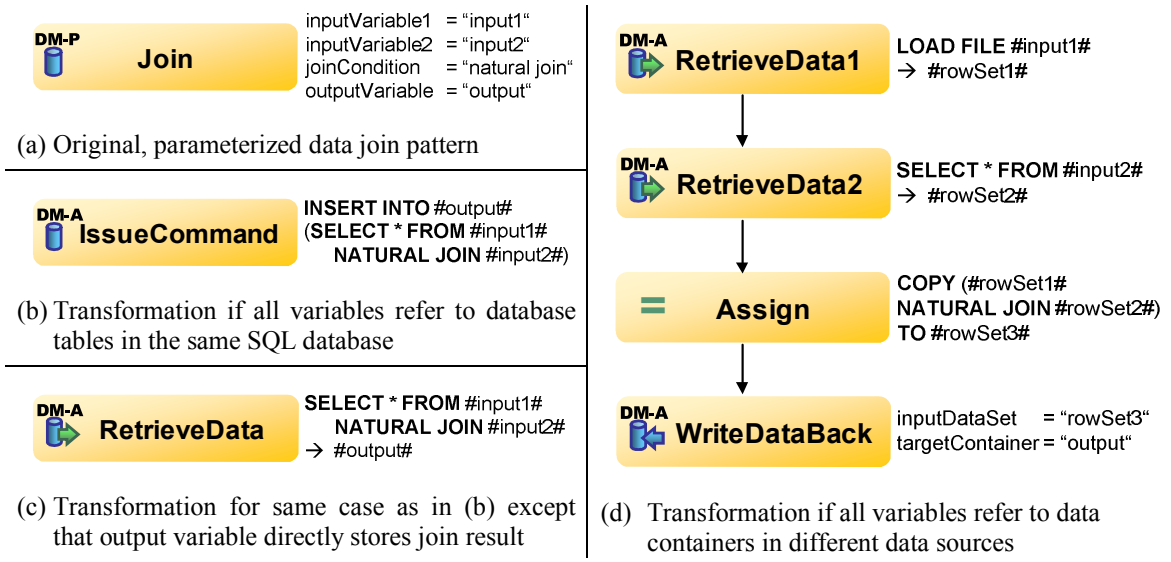

Abbildung 3.3.: Beispiel eines Join Patterns [RRS+11]

Abbildung 3.3 zeigt ein Pattern, das einen Join zwischen zwei Datensätzen realisiert. Der Modellierer muss lediglich spezielle Parameter setzen. In diesem Fall muss er zwei Input-Variablen, eine Output-Variable sowie eine Join-Condition angeben. Umformungsregeln überführen das abstrakte Pattern (vgl. Abbildung 3.3 (a)) in ausführbare Workflow-Fragmente. Welche Aktivitäten anschließend im Speziellen ausgeführt werden, bestimmen die gewählten Eingabeparameterwerte. Handelt es sich bei den benötigten Variablen um *data container reference variables*, die auf Datenbanktabellen in derselben Datenbank verweisen, reicht eine IssueCommand Aktivität. Diese führt dann den Join innerhalb der Datenbank aus (vgl. Abbildung 3.3 (b)). Sollen die generierten Daten im Workflow bereitgestellt werden, muss eine RetrieveData Aktivität verwendet werden (vgl. Abbildung 3.3 (c)). Verweisen alle drei Variablen auf Container in unterschiedlichen Datenquellen, werden zwei RetrieveData Aktivitäten sowie ein WriteDataBack Aktivität benötigt (vgl. Abbildung 3.3 (d)). Die nötige Umformung kann erst dann geschehen, wenn die benötigten Datenquellen feststehen. Bei einer statischen Beschreibung der Datenquellen, kann die Umformung kurz vor dem Deployment geschehen. Bei einem Late Binding der Datenquellen erst zur Laufzeit.

3.5. SIMPL Core

Der SIMPL Core bietet generische Operationen, um auf beliebige externe Datenquellen zu zugreifen. Die Spezifikationen sind unabhängig von den darunter liegenden Datenquel-

len. Die Operationen, die der SIMPL Core bietet, übernehmen die Namen der BPEL-DM Aktivitäten: IssueCommand, RetrieveData und WriteDataBack. Jede BPEL-DM Aktivität ruft die zugehörige SIMPL Core Operation auf. Abbildung 3.4 zeigt die auszutauschenden Daten zwischen Execution Engine und SIMPL Core. Der SIMPL Core leitet *DM commands*, generierte Daten und Benachrichtigungen nur weiter und führt selbst keine komplexen Transformationen oder Analysen durch.

Jede SIMPL Core Operation erwartet einen *logical data source descriptor* als Eingabe, um die entsprechende Datenquelle zu finden. Die IssueCommand und RetrieveData Operationen benötigen des Weiteren jeweils einen *DM command*. Bei der RetrieveData Operation muss dieser Daten generieren. Die WriteDataBack Operation erwartet ein *data set* sowie eine Referenz auf einen Container als Eingabe. Die IssueCommand und WriteDataBack Operationen benachrichtigen die Execution Engine über Erfolg bzw. Misserfolg. Die RetrieveData Operation sendet nach erfolgreicher Ausführung die generierten Daten an die Execution Engine. Im Fehlerfall wird wiederum eine entsprechende Benachrichtigung gesendet.

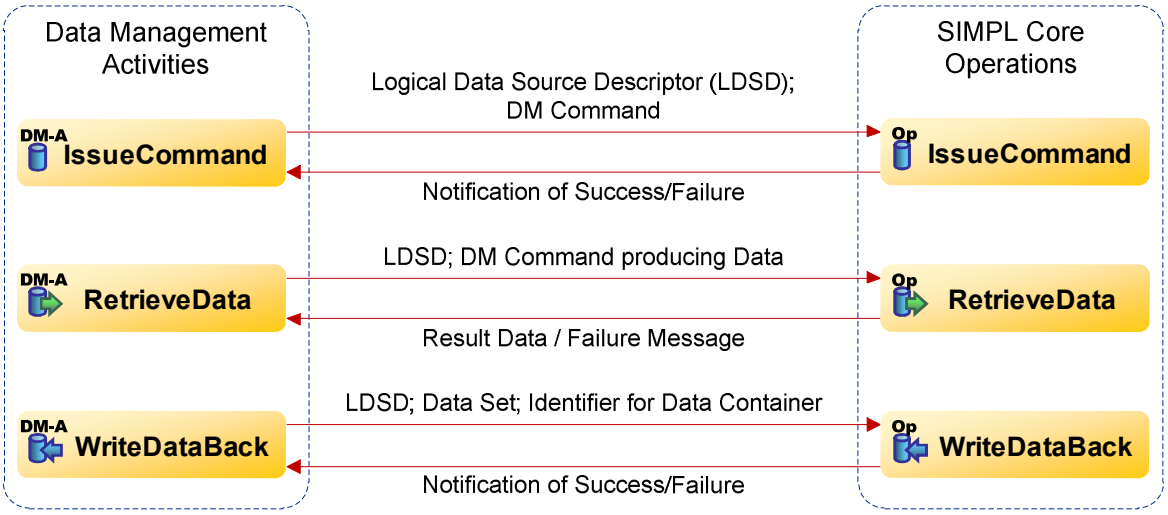

Abbildung 3.4.: Datenaustausch zwischen Execution Engine und SIMPL Core [RRS+11]

Für den Zugriff auf beliebige Datenquellen müssen die generischen Operationen des SIMPL Cores für konkrete Datenquellen oder für bestimmte Typen von Datenquellen implementiert werden. *Data source connectors* übernehmen diese Aufgabe. So könnte es z.B. einen Konnektor für relationale Datenbanken, die JDBC zur Kommunikation nutzen, geben, sowie einen Konnektor für Dateisysteme. Ein Konnektor muss jedoch nicht alle generischen Operationen unterstützten bzw. implementieren. Bei Sensornetzen z.B. macht eine WriteDataBack Operation keinen Sinn. Zusätzlich zu den *data source connectors* gibt es noch *data converter*. Diese Konverter transformieren Daten aus dem Ausgabeformat des Konnektors in ein XML-basiertes Format, welches vom Workflow unterstützt wird. So könnte es z.B. einen Konverter geben, der ein JDBC Result Set in eine XML RowSet Darstellung überführt.

3.6. Resource Management

Im Resource Management werden benötigte Metadaten gespeichert. Diese Metadaten werden zur Abbildung der generischen Operation des SIMPL Cores auf konkrete Zugriffsmechanismen benötigt. Dazu können im Resource Management die folgenden vier Objekte hinterlegt werden:

- Data Sources
- Data Containers
- Data Source Connectors
- Data Converters

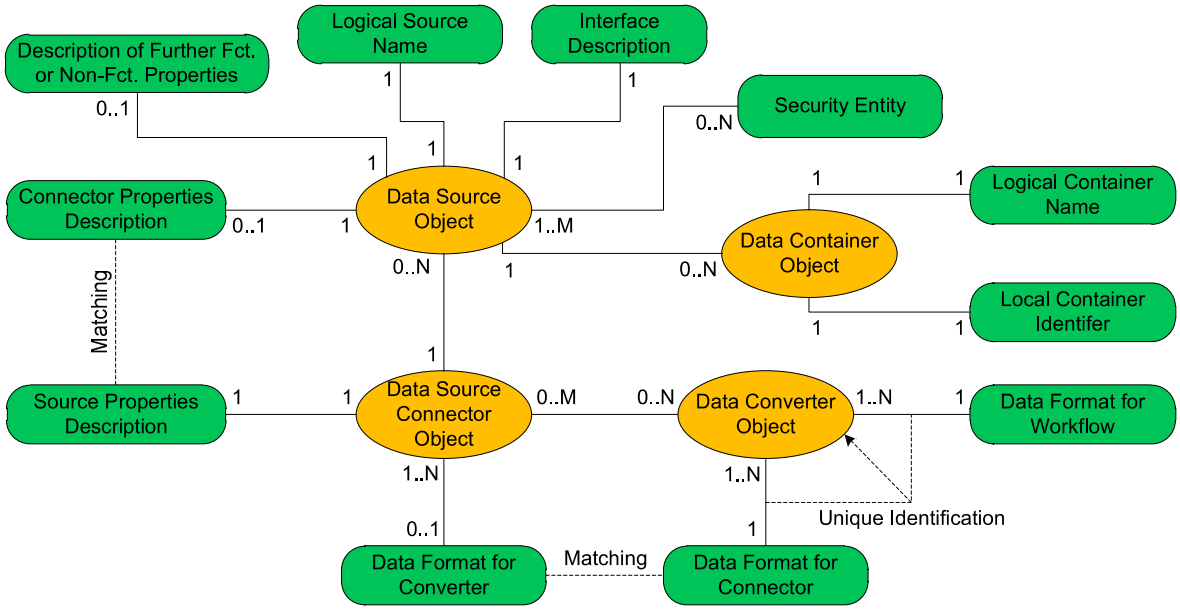

Abbildung 3.5.: Zusammenhang zwischen den verschiedenen Metadaten [RRS+11]

Abbildungg 3.5 zeigt die Beziehung der verschiedenen Objekte untereinander. Eine Datenquelle wird über Attribute beschrieben. Für jede Datenquelle gibt es einen einzigartigen *logical source name* der zur Identifizierung innerhalb des SIMPL-Rahmenwerks dient. Dieser Name kann innerhalb eines Workflows als *logical datasource descriptor*, z.B. in einer *data source reference variable*, genutzt werden. Die *Interface Description* beinhaltet Informationen über das Interface der Datenquelle, z.B. unter welcher Addresse die Datenquelle erreicht werden kann. Das Attribut *Security entitiy* enthält Authentifizierungsinformationen, wie z.B. Benutzername und Passwort. Das Attribut *description of further functional or non-functional properties* beinhaltet Information über die Eigenschaften einer Datenquelle, wie z.B. die maximale Antwortzeit. Diese Informationen werden für ein Late Binding benötigt. Ein *data container object* beschreibt

die Container die von einer Datenquelle verwaltet werden. Die einzelnen Container erhalten einen *logical container name*, der innerhalb eines Workflows als Containerreferenz genutzt werden kann. Der *logical container name* wird mit einem *local container identifier* gepaart, der eindeutig den entsprechenden Container in der Datenquelle identifiziert.

Wenn eine Datenquelle im Resource Management registriert oder verändert wird, kann der Nutzer direkt einen passenden *data source connector* angeben. Oder aber es kann eine *connector properties description* hinterlegt werden, die besagt, welche Eigenschaften ein geeigneter Konnektor haben muss. Jeder *data source connector* erhält dafür eine *source properties description*, die im Gegenzug besagt, welche Eigenschaften der Konnektor von einer Datenquelle erwartet. Darauf aufbauend kann dann ein passender Konnektor anhand der *connector properties description* und *source properties description* ausgewählt werden.

Das Attribut *data format for connector* beinhaltet eine Formatbeschreibung. Diese legt fest, welches Ein- und Ausgabeformat der Konnektor unterstützt. Konverter teilen diese Informationen. Dadurch können passende Paare von Konnektor und Konverter gebildet werden. Des Weiteren benötigen Konverter noch eine weitere Formatbeschreibung (*data format for workflow*). Diese legt fest, in welchem Format der Konverter die Eingabedaten, z.B. als XML RowSet, aus dem Workflow erwartet, und in welchem Format er die Ausgabedaten wiederum zurückschickt.

3.7. Ablauf einer DM Aktivität

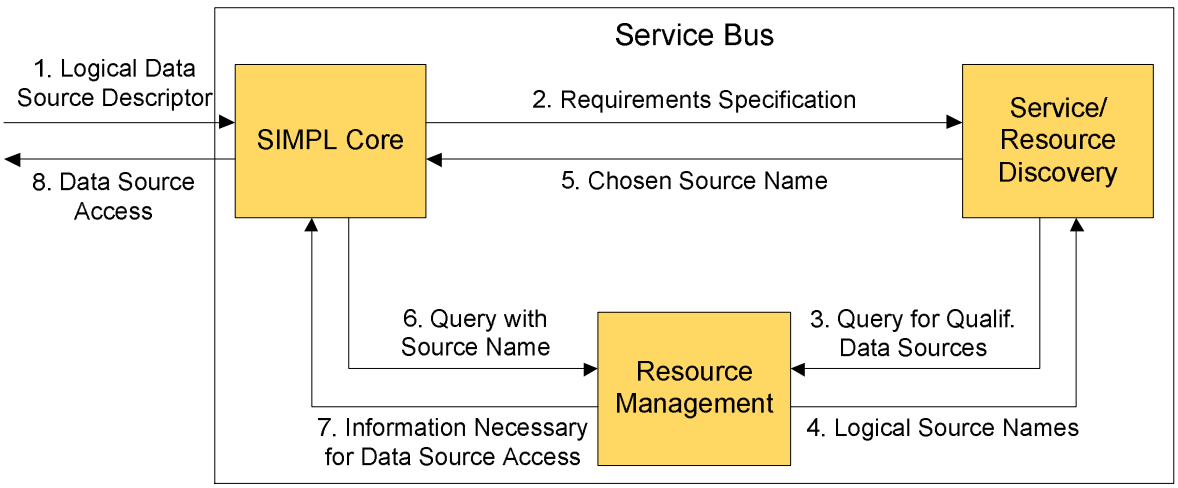

Abbildung 3.6.: Interaktion zwischen den verschiedenen Service Bus Komponenten [RRS+11]

Greift eine DM Aktivität mithilfe eines *logical data source descriptors* auf eine Datenquelle zu, muss der SIMPL Core alle benötigten Informationen über die referenzierte Datenquelle

bereitstellen. Dazu zählen unter anderem die *Interface Description*, die Zugangsdaten sowie geeignete Konnektoren und Konverter. Des Weiteren müssen logische Containerreferenzen aufgelöst werden. Abbildung 3.6 zeigt exemplarisch die Interaktion der beteiligten Service Bus Komponenten. Der SIMPL Core erhält einen *logical data source descriptor* (1). Verweist die Referenz auf keine konkrete Datenquelle, sonderen beinhaltet eine Anforderungsbeschreibung, leitet der SIMPL Core diese an die Service/Resource Discovery Komponente weiter (2). Diese sucht wiederum in der Resource Management Komponente nach geeigneten Datenquellen und wählt eine davon aus (3 und 4). Der logische Name dieser Datenquelle wird der SIMPL Core Komponente mitgeteilt (5), welche wiederum in der Resource Management Komponente die für den Zugriff auf die Datenquelle notwendigen Informationen abfragt (6 und 7). Die entsprechende SIMPL Core Operation kann nun mithilfe der gesammelten Informationen auf die Datenquelle zugreifen (8). Bei einer RetrieveData oder WriteDataBack Aktivität sendet die Execution Engine noch zusätzlich den Datentyp der zugehörigen BPEL Variablen, in der die Daten abgelegt sind/werden, mit. Mithilfe dieser Information kann dann ein passender Konverter gewählt werden.

4. Bestandsaufnahme

Das SIMPL-Rahmenwerk wurde bisher in weiten Teilen prototypisch umgesetzt und wird sukzessive weiterentwickelt [SIM]. Die Implementierung ist in ein Workflow Management System eingebunden. Dieses basiert auf BPEL [JE07], sowie auf der BPEL-Engine Apache ODE (Orchestration Director Engine) [Foua] und dem Modellierungstool Eclipse BPEL Designer [Foub]. Im Folgenden werden nun der erweiterte Eclipse BPEL Designer, das Resource Management sowie die Umsetzung des SIMPL Cores erläutert. Das nächste Kapitel beschäftigt sich dann mit den Gemeinsamkeiten bzw. Unterschieden, die sich beim Zugriff auf die in dieser Arbeit betrachteten verschiedenen Datenquellen oder Typen von Datenquellen ergeben. Die in Abschnitt 3.4 erläuterten DM Patterns wurden bisher nur konzeptionell entwickelt. Aus diesem Grund werden DM Patterns im weiteren Verlauf dieser Arbeit nicht mehr betrachtet.

4.1. Der erweiterte Eclipse BPEL Designer

Der Eclipse BPEL Designer wurde im Zuge der Implementierung des SIMPL-Rahmenwerks erweitert. Abbildung 4.1 zeigt die resultierende Oberfläche. Exemplarisch wurde eine RetrieveData Aktivität modelliert. Die Palette (grüne Markierung) beinhaltet die neuen DM Aktivitäten. Diese sind:

- QueryData
- IssueCommand
- RetrieveData
- WrtieDataBack
- TransferData

Die Funktionen der IssueCommand, RetrieveData und WriteDataBack Aktivitäten werden in Abschnitt 3.3 erläutert. Aus diesem Grund werden nun nur noch die übrigen Aktivitäten beschrieben.

Mithilfe einer **QueryData** Aktivität können Daten extrahiert und in einem referenzierten Container abgelegt werden. Dieser Container muss jedoch, im Gegensatz zur TransferData Aktivität, in der selben Datenquelle vorliegen. Als Eingabeparameter werden dazu eine *data source reference variable* sowie eine *data container reference variable* benötigt. Die zu extrahierenden Daten werden über ein *DM command* (im weiteren Verlauf der Arbeit auch als Statement bezeichnet) spezifiziert.

Die **TransferData** Aktivität bietet die Möglichkeit Daten aus einer Datenquelle zu extrahieren und diese wiederum in einer anderen Datenquelle zu speichern. Dazu werden folgende Eingabeparameter benötigt: Die beiden *data source reference variables* referenzieren die beiden Datenquellensysteme (Datenquelle und Datensenke), zwischen denen die Daten ausgetauscht werden sollen. Der Container in dem die Daten in der Datensenke gespeichert werden sollen, wird mithilfe einer *data container reference variable* referenziert. Ein Statement spezifiziert wiederum die eigentlichen Daten, die aus der Datenquelle extrahiert und zur Datensenke übertragen werden.

Die Parametrisierung der einzelnen Aktivitäten geschieht in der Property-View (rote Markierung). Dort können entsprechende Variablen ausgewählt und Statements erzeugt werden. Die vom Nutzer erzeugten Variablen werden im Bereich Variables angezeigt (blaue Markierung). Dort können auch neue Variablen erstellt werden. Jede dieser fünf Aktivitäten nutzt die generischen Operationen des SIMPL Cores, um die eigentliche Operation auf der Datenquelle auszuführen.

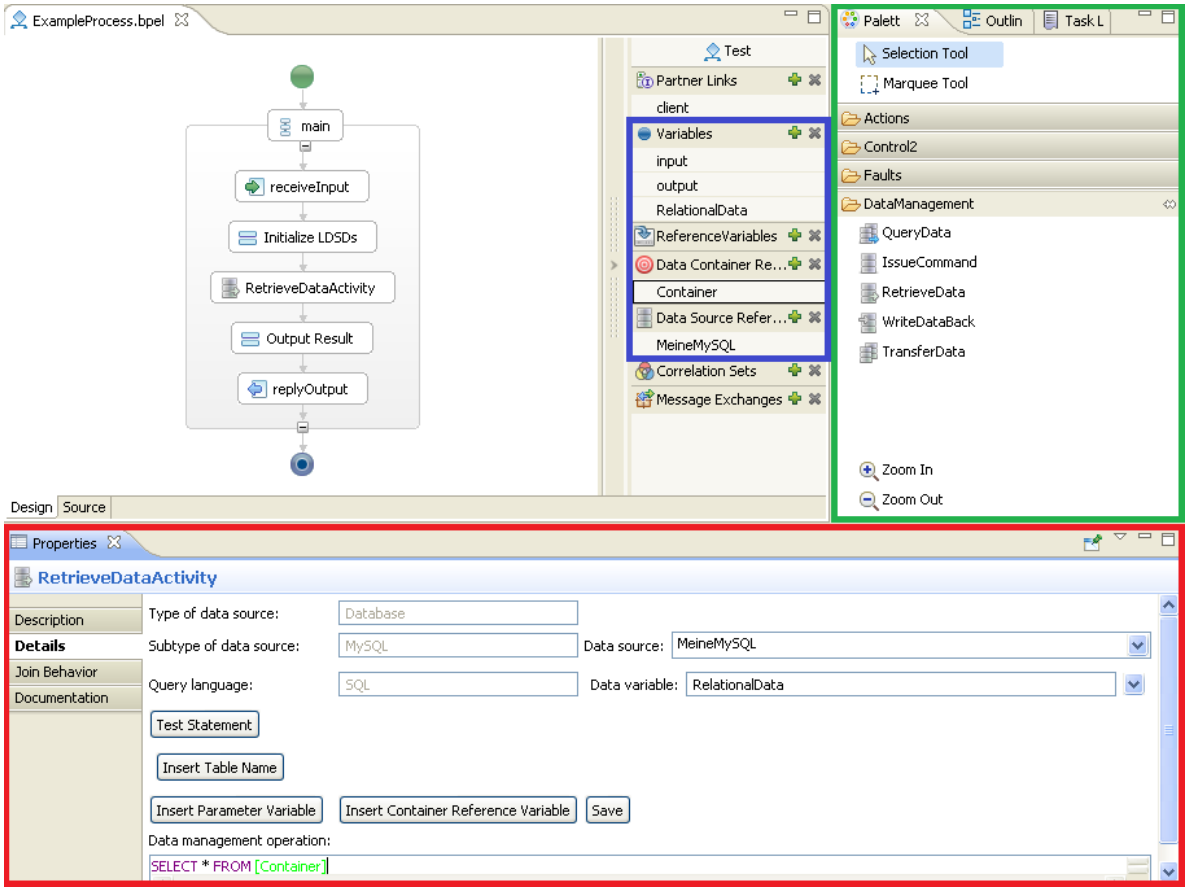

Abbildung 4.1.: Der erweiterte Eclipse BPEL Designer

4.2. Resource Management

In Abschnitt 3.6 wird auf die Aufgabe des Resource Managements eingegangen. In der bestehenden Implementierung läuft das Resource Management als Web Service (vgl. Abschnitt 2.2.3) in einer Axis2 Laufzeitumgebung. Zusätzlich gibt es die Möglichkeit direkt über das Java Interface auf das Resource Management zuzugreifen. Beim SIMPL Core kann der Nutzer einstellen, welche der beiden Optionen genommen wird. Der erweiterte Eclipse BPEL Designer nutzt immer den Web Service zur Kommunikation mit dem Resource Management.

Der SIMPL Core nutzt das Resource Management, z.B. um Referenzen auf Datenquellen aufzulösen oder geeignete Konnektoren zu finden. Über ein Web Interface kann der Nutzer neue Objekte im Resource Management registrieren und bestehende Objekte anpassen oder löschen. Dazu bietet das Web Interface die folgenden Funktionen:

- **Data Source Management** Hier werden alle registrierten Datenquellen angezeigt. Des Weiteren besteht die Möglichkeit neue Datenquellen zu registrieren sowie registrierte Datenquellen anzupassen bzw. zu löschen.
- **Connector Management** Hier werden registrierte Konnektoren angezeigt. Auch hier besteht die Möglichkeit neue Konnektoren zu registrieren sowie registrierte Konnektoren anzupassen bzw. zu löschen.
- **Data Converter Management** Hier werden registrierte Konverter angezeigt. Es besteht ebenfalls die Möglichkeit neue Konverter zu registrieren sowie registrierte Konverter anzupassen bzw. zu löschen.
- **Data Transformation Service Management** Hier werden registrierte Data Transformation Services angezeigt. Mithilfe dieser Dienste können Daten in andere Formate transformiert werden. So ist es z.B. möglich Daten aus dem CSVDataFormat in das RDBDataFormat zu transformieren. Auch hier besteht die Möglichkeit Data Transformation Services zu registrieren, sowie registrierte Data Transformation Services anzupassen bzw. zu löschen.

Die eigentlichen Daten, die vom Resource Management verwaltet werden, sind in einer PostgreSQL Datenbank [Gro] gespeichert. Bei der Initialisierung des Resource Managements werden die folgenden Tabellen in dieser Datenbank angelegt (Version 1.0.7):

- **Connectors(**
  Id, Dataconverter_Id, Name, Input_Datatype, Output_Datatype, Implementation, Properties_Description**)**
- **Dataconverters(**
  Id, Name, Input_Datatype, Output_Datatype, Workflow_Dataformat, Direction_Output_Workflow, Direction_Workflow_Input, Implementation, XML_Schema**)**
- **Datasources(**
  Id, Connector_Id, Logical_Name, Security_Username, Security_Password, Interface_Description, Properties_Description, Connector_Properties_Description**)**

- **Datatransformationservices(**
  Id, Name, Connector_Dataformat, Workflow_Dataformat, Direction_Connector_Workflow, Direction_Workflow_Connector, Implementation**)**
- **StrategyPlugins(**
  Id, Name, Implementation**)**
- **Languages(**
  Id, Name, Statement_Description**)**
- **DataSource_Reference_Types(**
  Id, Name, XSD_Type**)**
- **DataContainer_Reference_Types(**
  Id, Name, XSD_Type**)**

Die ersten vier Tabellen speichern im Wesentlichen die in Abschnitt 3.6 erläuterten Metadaten. Zum besseren Verständniss wird auf einige Attribute nun noch einmal eingegangen und der Zusammenhang zwischen den einzelnen Tabellen beschrieben.

Die Tabelle **Connectors** beinhaltet Metadaten über verfügbare Konnektoren. Die Attribute *InputDataType* und *OutputDataType* beschreiben das Ein- und Ausgabeformat des Konnektors. Ein Verweis auf die konkrete Implementierung wird unter dem Attribut *Implementation* gespeichert. Das Attribut *Properties_Description* beinhaltet Eigenschaften, die der jeweilige Konnektor aufweist. Abbildung 4.2 zeigt exemplarisch einen Inhalt dieser Tabelle. Das Attribut *Dataconverter_Id* verweist auf einen geeigneten Konverter in der Tabelle **Dataconverters**. Dort sind wiederum Metadaten über verfügbare Konverter gespeichert. Das Attribut *Workflow_Dataformat* enthält den Namen einer Formatbeschreibung. Diese legt das Format zum Austausch mit dem Workflow fest. Das zugehörige XML Schema wird unter dem Attribut *XML_Schema* gespeichert. Die Attribute *Input_Datatype* und *Output_Datatype* beschreiben auch hier wieder Datenformate. Das Attribut *Direction_Output_Workflow* enthält einen Wahrheitswert. Dieser besagt, ob der Konverter Daten, die im *Output_Dataformat* vorliegen, in das Workflow Datenformat überführen kann. Ob eine Transformation in Rückrichtung, also vom Workflow Datenformat in das *Input_Dataformat* möglich ist, wird mithilfe des Attributs *Direction_Workflow_Input* festgehalten. Das Attribut *Implementation* verweist auf die konkrete Implementierung des jeweiligen Konverters. Ein Inhalt dieser Tabelle ist in Abbildung 4.3 exemplarisch dargestellt. Metadaten über verfügbare Datenquellen werden in der Tabelle **Datasources** gespeichert. Das Attribut *Connector_Id* verweist auf einen geeigneten Konnektor für die jeweilige Datenquelle. Der logische Name einer Datenquelle muss eindeutig sein. Über diesen kann dann die Datenquelle im Workflow referenziert werden. Die restlichen Attribute speichern Authentifizierungsinformationen, Eigenschaften der Datenquelle sowie Eigenschaften, die ein geeigneter Konnektor aufweisen muss (vgl. Abschnitt 3.6). Metadaten über Data Transformation Services werden in der Tabelle **Datatransformationservices** gespeichert. Der Names des Dienstes wird mithilfe des Attributs *Name* festgehalten. Die Attribute *Connector_Dataformat* und *Workflow_Dataformat* beschreiben wiederum zwei Datenformate. Bietet der Data Transformation Service die Möglichkeit Daten, die im *Connector_Dataformat* vorliegen, in das *Workflow_Dataformat* zu transformieren, wird mithilfe des Attributs *Direction_Connector_Workflow* ein positiver Wahrheitswert gespeichert.

Wird solch eine Transformation hingegen nicht unterstützt wird ein negativer Wahrheitswert gespeichert. Das Attribut *Direction_Workflow_Connector* enthält ebenfalls einen Wahrheitswert. Dieser besagt, ob eine Transformation in die Rückrichtung, also vom *Workflow_Dataformat* in das *Connector_Dataformat* möglich ist. Werden Datenquellen erst zur Laufzeit gesucht und eingebunden, bestimmt die Suchstrategie welche in Frage kommende Datenquelle gewählt wird. Die Tabelle **StrategyPlugins** enthält Verweise auf implementierte Suchstrategien. Abfragesprachen, wie z.B. SQL, die vom SIMPL-Rahmenwerk unterstützt werden, werden in der Tabelle **Languages** gespeichert. Das Attribut *Statement_Description* enthält ein XML-Dokument. Dieses beschreibt, wie Ausdrücke der jeweiligen Sprache aussehen dürfen. Die Tabelle **DataSource_Reference_Types** enthält XML Schema Typen. Diese Typen definieren die Struktur einer Referenz auf eine Datenquelle. In den Abschnitten 3.3 und 3.6 wurde bereits erläutert, dass jede Datenquelle verschiedene Container (*data container*) verwaltet. WriteDataBack und QueryData Aktivitäten erhalten als Eingabeparameter jeweils eine *data container reference variable*. Diese Variable referenziert den Ort, an dem Daten gespeichert bzw. bereitgestellt werden sollen. Die Idee ist, dass ein Container über einen logischen Namen referenziert werden kann. Metadaten über die einzelnen Container werden im Resource Management gespeichert. Das Resource Management bildet diesen logischen Namen dann auf einen konkreten Identifikator bzw. lokalen Bezeichner ab, z.B. auf einen Schemanamen und einen Tabellennamen bei SQL Datenbanken. Bei der bestehenden Implementierung ist dies allerdings noch nicht möglich. Bei der Initialisierung des Resource Managements wird bisher lediglich die Tabelle **DataContainer_Reference_Types** erzeugt. Diese Tabelle enthält, genau wie die Tabelle *DataSource_ReferenceTypes*, XML Schema Typen. Diese Typen definieren die Struktur von Referenzen auf einzelne Container. Bisher gibt es jedoch noch keine Möglichkeit Container im Resource Management zu registrieren. Auch die dafür benötigte Tabelle wird noch nicht erzeugt. Bisher kann bei der Parametrisierung einer DM Aktivität lediglich eine *data container reference variable* angegeben werden, die direkt den lokalen Bezeichner beinhaltet. Des Weiteren geht dies auch nur für relationale Datenbanken. Auf diese Weise kann eine Tabelle zumindest über den Schema- und Tabellenname referenziert werden. An der Umsetzung der eigentlichen Idee wird momentan gearbeitet.

	id [PK] serial	dataconverter_id integer	name character varying(255)	input_datatype character varyin	output_datatype character varying(	implementation character varying	properties_description xml
1	1	1	DB2RDBConnector	List<String>	RDBResult	org.simpl.core.plugins	<properties_description xml
2	2	1	DerbyRDBConnector	List<String>	RDBResult	org.simpl.core.plugins	<properties_description xml
3	3	1	EmbDerbyRDBConnector	List<String>	RDBResult	org.simpl.core.plugins	<properties_description xml
4	4	1	MySQLRDBConnector	List<String>	RDBResult	org.simpl.core.plugins	<properties_description xml
5	5	1	PostgreSQLRDBConnector	List<String>	RDBResult	org.simpl.core.plugins	<properties_description xml
6	6	3	WindowsLocalFSConnector	File	RandomFile	org.simpl.core.plugins	<properties_description xml

Abbildung 4.2.: Metadaten über implementierte Konnektoren

	id [PK] serial	name character varying(255)	input_dataty character var	output_datat character var	workflow_dataforma character varying(25	direction_out character(5)	direction_wo character(5)	implementatior character varyi	xml_schema xml
1	1	RDBDataConverter	List<String>	RDBResult	RDBDataFormat	true	true	org.simpl.core.plu	<?xml version="1.0
2	2	CSVDataConverter	File	RandomFile	CSVDataFormat	true	true	org.simpl.core.plu	<?xml version="1.0
3	3	RandomFileDataConverter	File	RandomFile	RandomFileDataFormat	true	true	org.simpl.core.plu	<?xml version="1.0

Abbildung 4.3.: Metadaten über implementierte Konverter

Die erläuterten Tabellen bilden die Datenbasis des Resource Managements. Der Resource Management Web Service bietet zur Verwaltung der einzelnen Objekte CRUD (Create, Read, Update und Delete) Operationen. Für Datenquellen gibt es z.B. die folgenden Operationen:

- **addDataSource** Diese Operation registriert eine neue Datenquelle. Die notwendigen Informationen werden in die PostgreSQL Datenbank geschrieben.
- **getDataSourceByName** Diese Operation extrahiert alle Informationen bzgl. einer Datenquelle aus der PostgreSQL Datenbank. Zur Referenzierung des jeweiligen Tupels wird der logische Name der Datenquelle verwendet.
- **getDataSourceById** Diese Operation extrahiert ebenfalls alle Informationen bzgl. einer Datenquelle. Zur Referenzierung des jeweiligen Tupels wird bei dieser Operation jedoch die Id der Datenquelle verwendet.
- **updateDataSource** Diese Operation aktualisiert die Informationen, die über eine Datenquelle in der PostgreSQL Datenbank gespeichert sind.
- **deleteDataSource** Diese Operation löscht alle Informationen, die bzgl. einer Datenquelle in der PostgreSQL Datenbank gespeichert sind.

4.3. SIMPL Core

Der SIMPL Core bildet zusammen mit dem Resource Management das Herzstück des SIMPL-Rahmenwerks. Der SIMPL Core wurde ebenfalls wie das Resource Management als Web Service (vgl. Abschnitt 2.2.3) implementiert. Zusätzlich ist ein Zugriff über das Java Interface möglich. Somit kann Apache ODE wahlweise über den Web Service oder das Java Interface auf den SIMPL Core zugreifen. Der SIMPL Core bietet die folgenden öffentlichen Operationen:

- IssueCommand
- RetrieveData
- WriteDataBack
- QueryData
- GetMetaData

Die in Abschnitt 4.1 erläuterte TransferData Aktivität baut auf diesen Operationen auf. Eine RetrieveData Operation extrahiert die spezifizierten Daten aus der Datenquelle. Eine WriteDataBack Operationen speichert diese Daten dann wiederum in der Datensenke. Zusätzlich bietet der SIMPL Core noch die Operation GetMetaData. Diese Operation stellt Metadaten zur Verfügung. Bei relationalen Datenbanken sind dies die Strukturen der einzelnen Tabellen, die in dieser Datenbank gespeichert sind. Bei einem Dateisystem werden alle Befehle, die auf Kommandozeilenebene ausgeführt werden können, zurückgegeben. Im weiteren Verlauf dieser Arbeit werden nun nur noch die ersten vier SIMPL Core Operationen betrachtet, da

sich die Implementierung der GetMetaData Operation nur dann zischen den einzelnen Datenquellen unterscheidet, wenn sich auch die Implementierungen der anderen Operationen maßgeblich unterscheiden. In solchen Fällen ist ohnehin ein unterschiedlicher Konnektor notwendig. Diese Operationen wurden jeweils auf zwei Arten implementiert. Der Unterschied zwischen den beiden Implementierungen soll nun anhand der RetrieveData Operation verdeutlicht werden. Wird eine Datenquelle über einen logischen Namen referenziert, können über diesen alle benötigten Informationen im Resource Management abgefragt werden. Das übergebene Statement kann dann auf der referenzierten Quelle ausgeführt werden. Für diesen Zweck bietet der SIMPL Core die Operation *retrieveDataByDataSourceName*. Die Operation *retrieveDataByDataSource* wird hingegen verwendet, wenn eine Datenquelle nicht über ihren logischen Namen referenziert wird. In diesem Fall wird zuerst überprüft, ob die für ein Late Binding benötigten Informationen vorliegen. Eine WS-Policy Beschreibung [BBC+07] spezifiziert die Eigenschaften, die eine in Frage kommende Datenquelle aufweisen muss. Welche Datenquelle schließlich ausgewählt wird, wird durch die Suchstrategie bestimmt. Die WS-Policy Beschreibung und die Suchstrategie werden der Resource Discovery Komponente übergeben. Wurde eine passende Datenquelle gefunden, kann anschließend das spezifizierte Statement auf dieser ausgeführt werden.

Listing 4.1 Interface Beschreibung eines Konnektors [SIM]

```
public interface Connector<S, T> {

        public boolean issueCommand(DataSource dataSource, String statement)
                        throws ConnectionException;

        public T retrieveData(DataSource dataSource, String statement)
                        throws ConnectionException;

        public boolean writeDataBack(DataSource dataSource, S data, String target)
                        throws ConnectionException;

        public boolean queryData(DataSource dataSource, String statement,
                        String target) throws ConnectionException;

        public DataObject getMetaData(DataSource dataSource, String filter)
                        throws ConnectionException;

        public boolean createTarget(DataSource dataSource, DataObject dataObject,
                        String target) throws ConnectionException;
}
```

Dazu wird ein geeigneter Konnektor benötigt, der die SIMPL Core Operationen implementiert. Metadaten über die einzelnen Konnektoren sind im Resource Management gespeichert. Bisher unterstützt das SIMPL-Rahmenwerk die folgenden Datenquellen:

- IBM DB2 (DB2RDBConnector)
- Apache Derby (DerbyRDBConnector)
- Apache Derby Embedded Mode (EmbDerbyRDBConnector)

- Sun MySQL (MySQLRDBConnector)
- PostgreSQL (PostgreSQLRDBConnector)
- Windows Dateisystem (WindowsLocalFSConnector)

Bei den ersten fünf Datenquellen handelt es sich um relationale Datenbanken. Der Konnektor für das Windows Dateisystem funktioniert größtenteils auch unter Linux. Die einzelnen Konnektoren implementieren das in Listing 4.1 gezeigte Interface. Die generischen Variablentypen *S* und *T* legen das Ein- bzw. Ausgabeformat des Konnektors fest. Der Konnektor für PostgreSQL Datenbanken erwartet z.B. als Eingabe eine Liste mit Strings (SQL Konstrukte), um Daten mit der WriteDataBack Operation in einer Datenbank zu speichern. Als Ausgabe erzeugt die RetrieveData Operation ein RDBResult (JDBC ResultSet + zugehörige Metdaten).

Eine DM Aktivität übergibt bzw. erwartet Daten in einem XML-basierten Format. Konverter implementieren die benötigten Transformationen von den Formaten eines Konnektors zu diesen XML-basierten Formaten und umgekehrt. Das vom Workflow gewünschte Format (im weiteren Verlauf dieser Arbeit als Workflow Datenformat bezeichnet) ist als XML Schema Definition im Resource Management hinterlegt. Bisher wurden die folgenden Konverter umgesetzt:

- **RDBDataConverter** Dieser Konverter wird für relationale Datenbanken verwendet.
- **CSVDataConverter** Mithilfe des CSVDataConverters kann der Inhalt aus CSV-Dateien verarbeitet werden.
- **RandomFileDataConverter** Dieser Konverter kann mit einer beliebigen Datei bzw. einem Ordner mit mehren Dateien umgehen. Dabei wird die spezifische Datenstruktur der Datei nicht berücksichtigt.

Listing 4.2 Interface Beschreibung eines Konverters [SIM]

```
public interface DataConverter<S, T> {

        public DataObject toSDO(S data);

        public T fromSDO(DataObject data);

        public DataObject getSDO();

        public String getDataFormat();
}
```

Die zugehörige Interface-Beschreibung ist in Listing 4.2 dargestellt. Als Datenstruktur werden *Service Data Objects* [AAB+06] verwendet. Dadurch können Daten aus unterschiedlichen Quellen einfacher gehandhabt werden. Die generischen Variablentypen *S* und *T* legen das Ein- bzw. Ausgabeformat des Konverters fest. Zur Ausführung einer RetrieveData oder WriteDataBack Operation auf einer Datenquelle wird ein Konnektor sowie ein passender

Konverter benötigt. Das Ausgabeformat des Konnektors muss dem Eingabeformat des Konverters entsprechen. Dasselbe gilt für die andere Richtung.

Die *toSDO* Funktion wird genutzt, um Daten, die im Ausgabeformat eines Konnektors vorliegen, in ein SDO zu überführen. Diese Struktur kann wiederum ohne großen Aufwand in eine XML Darstellung überführt und dem Workflow zur Verfügung gestellt werden. Mithilfe der *fromSDO* Funktion kann ein SDO in das Eingabeformat eines Konnektors überführt werden. Eine Analyse der einzelnen Workflow Datenformate erfolgt im nächsten Kapitel.

5. Analyse

Die vorliegende Studienarbeit verfolgt zwei Ziele. Zum einen sollen Unterschiede, die sich beim Zugriff auf die verschiedenen Datenquellen oder Typen von Datenquellen ergeben, soweit wie möglich bzw. sinnvoll aufgelöst werden. Zum anderen soll das Rahmenwerk um die Unterstützung weiterer Datenquellen erweitert werden. Dieses Kapitel beschreibt die Ergebnisse der Analyse der bisherigen Umsetzung des SIMPL-Rahmenwerks, bevor anschließend auf die Anforderungen bei der Integration der neuen Datenquellen eingegangen wird.

5.1. Analyse der bisherigen Umsetzung

Nachdem im vorherigen Kapitel die Aufgaben der Konverter und Konnektoren schon grob erläutert wurden, werden nun die einzelnen Umsetzungen genauer betrachtet. Ziel ist es Gemeinsamkeiten bzw. Unterschiede bzgl. der jeweiligen Datenquellen und Datenformate herauszuarbeiten.

5.1.1. Konverter

Eine Workflow Aktivität erwartet bzw. übergibt Daten in einem XML-basierten Format. Bisher wurden die folgenden zwei Formate definiert: das *RelationalDataFormat* und das *RandomFileDataFormat*. Die Konverter RDBDataConverter und CSVDataConverter nutzen das RelationaleDataFormat. Der RandomFileDataConverter hingegen nutzt das RandomFile-DataFormat. Die vollständigen Definitionen sind in Anhang B.1 und B.2 zu finden. Die durch die einzelnen Konverter realisierten Transformationen werden nun detailiert betrachtet.

Werden Daten mithilfe einer RetrieveData Operation aus einer relationalen Datenbank (vgl. Abschnitt 2.4.1) extrahiert, muss die Ergebnisrelation durch den **RDBDataConverter** in das RelationalDataFormat überführt werden. Abbildung 5.1 zeigt Auszüge aus dem zugehörigen XML Schema. Da eine Relation als Tabelle dargestellt werden kann, wird im weiteren Verlauf der Begriff Tabelle bevorzugt. Informationen über das verwendete Datenformat werden über das Attribut *dataFormat* des Elementes *RelationalDataFormat* gespeichert. Das Element *tDataFormatMetaData* dient als Container für Metadaten wie z.B. Informationen über die externe Datenquelle. Das Kindelement *dataSource* beinhaltet den Namen der Datenquelle. Mithilfe des Elementes *table* wird die durch eine RetrieveDataOperation extrahierte Relation bzw. Tabelle in diese Struktur aufgenommen. Metadaten über die eigentliche Tabelle werden mithilfe des Elementes *rdbTableMetaData* festgehalten. Über Attribute werden der

Schemaname, der Tabellenname sowie der zugehörige Katalog gespeichert. Für jede Spalte der Tabelle wird zudem ein Element *columnType* erzeugt. Als Attribut wird der Spaltenname und als Inhalt des Elementes der entsprechende Datentyp gespeichert. Um den eigentlichen Inhalt der Tabelle zu speichern, wird für jede Zeile der Tabelle ein Element *row* erzeugt. Die Werte der einzelnen Spalten einer Zeile werden durch Kindelemente *column* in diese Struktur aufgenommen. Listing 5.1 zeigt exemplarisch ein resultierendes XML-Dokument. Die ursprüngliche Tabelle hatte drei Zeilen. Jede Zeile hatte wiederum drei Spalten.

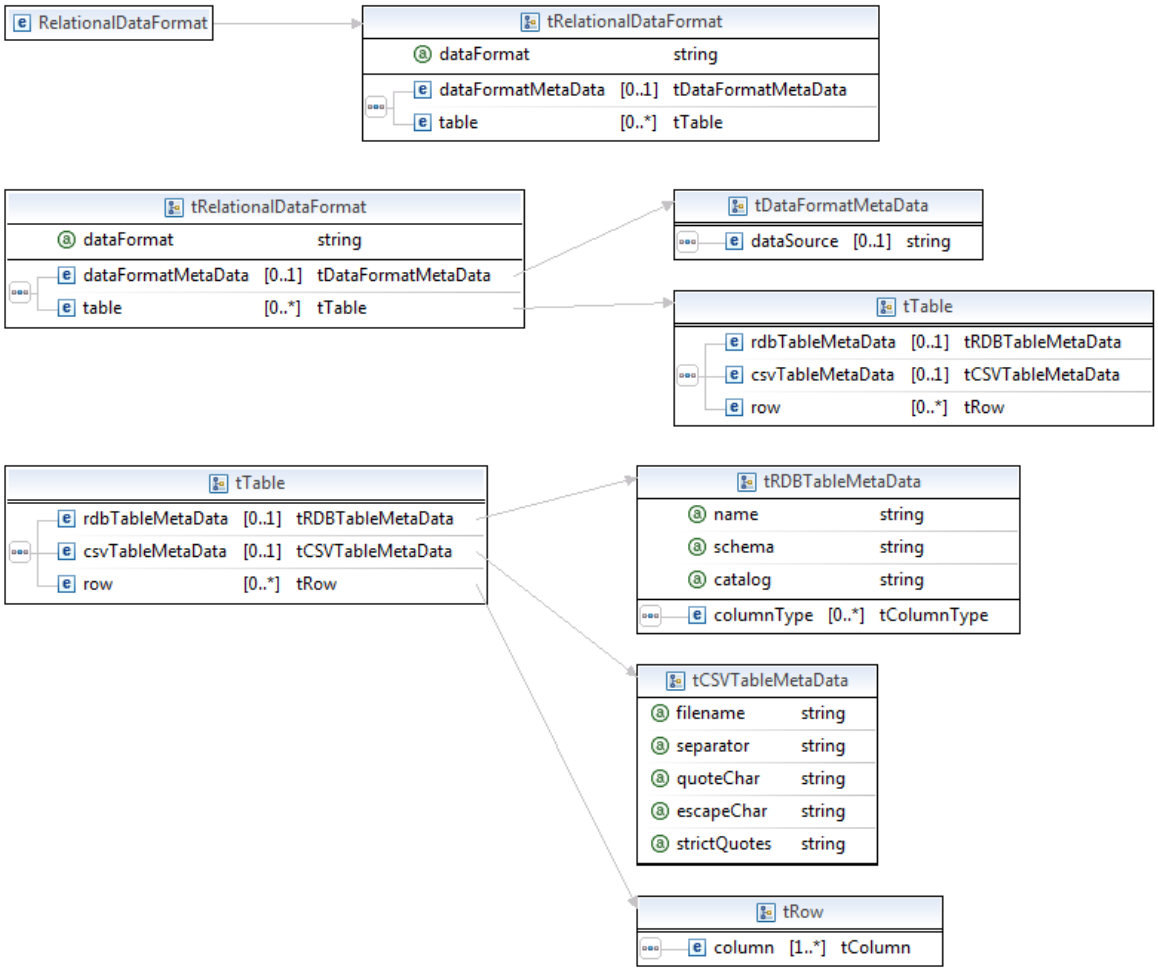

Abbildung 5.1.: Auszüge aus der RelationalDataFormat Definition

Sollen Daten, welche im RelationalDataFormat vorliegen, nun wiederum durch eine WriteDataBack Operation in eine relationale Datenbank geschrieben werden, muss erneut eine Transformation stattfinden. Der RDBDataConverter erzeugt für jede Zeile einer Tabelle SQL Anweisungen, die die entsprechende Zeile in der Datenbank ändert bzw. einfügt. Für jede

Tabelle werden also mehrere UPDATE- und INSERT-Ausdrücke erzeugt. Die einzelnen Anweisungen haben die folgende Struktur:

- **UPDATE** *Schemaname.Tabellenname* **SET** *Spaltenname* **=** *Wert* **WHERE** *1=1* **AND** *Primärschlüssel* **=** *Wert_Prim*
- **INSERT INTO** *Schemaname.Tabellenname* **(***Spaltenname_1, . . . , Spaltenname_N***) VALUES (***Wert_1, . . . , Wert_N***)**

Listing 5.1 Daten aus einer relationalen Datenbank in einer XML-Strukur

```
<result dataFormat="RDBDataFormat" type="format:tRelationalDataFormat">
  <dataFormatMetaData>
    <dataSource>PostgresSQL</dataSource>
  </dataFormatMetaData>
  <table>
    <rdbTableMetaData catalog="" name="personen" schema="public">
      <columnType columnName="ID" isPrimaryKey="true">int4(10)</columnType>
      <columnType columnName="VORNAME">varchar(256)</columnType>
      <columnType columnName="NACHNAME">varchar(256)</columnType
    </rdbTableMetaData>
    <row>
      <column name="ID">1</column>
      <column name="VORNAME">henrik</column>
      <column name="NACHNAME">p.</column>
    </row>
    <row>
      <column name="ID">2</column>
      <column name="VORNAME">conny</column>
      <column name="NACHNAME">s.</column>
    </row>
    <row>
      <column name="ID">3</column>
      <column name="VORNAME">bernd</column>
      <column name="NACHNAME">s.</column>
    </row>
  </table>
</result>
```

Für die in Listing 5.1 dargestellte Struktur würde der RDBDataConverter unter anderem die folgenden Statements generieren:

- **UPDATE** *public.personen* **SET** *ID=1* **WHERE** *1=1* **AND** *ID=1*
- **UPDATE** *public.personen* **SET** *VORNAME=henrik* **WHERE** *1=1* **AND** *ID=1*
- **UPDATE** *public.personen* **SET** *NACHNAME=p.* **WHERE** *1=1* **AND** *ID=1*
- **INSERT INTO** *public.personen* **(***ID, VORNAME, NACHNAME***) VALUES (***1, henrik, p.***)**

Die UPDATE-Ausdrücke werden benötigt, um einen Datensatz innerhalb einer Tabelle zu aktualisieren. Durch die INSERT-Ausdrücke kann ein Datensatz neu in eine Tabelle eingefügt werden. Der RDBDataConverter erzeugt beide Varianten (UPDATE- und INSERT-Ausdrücke).

Bei beiden Typen von Ausdrücken ist es notwendig das eigentliche Ziel zu referenzieren. Diese Informationen sind als Attribute des Elementes *rdbTableMetaData* gespeichert (vgl. letzter Abschnitt). Ein geeigneter Konnektor führt diese Statements dann aus. Beinhaltet die Variable *target* (vgl. Abschnitt 5.1.2) keinen Wert, wird die ursprüngliche Tabelle mithilfe der UPDATE-Ausdrücke aktualisiert. Andernfalls wird das referenzierte Ziel in die INSERT-Ausdrücke eingefügt bevor diese anschließend ausgeführt werden. Die einzelnen Datensätze werden in eine Tabelle neu eingefügt.

Der **CSVDataConverter** implementiert die benötigten Transformationen für CSV-Dateien. Eine CSV-Datei besteht aus mehreren Zeilen. Eine Zeile wird wiederum durch ein festes Zeichen in Spalten unterteilt. Eine Zeile mit den jeweiligen Werten der einzelnen Spalten entspricht genau einem Datensatz. Der CSVDataConverter basiert auf der Annahme, dass die erste Zeile Metadaten beinhaltet. Diese Zeile wird auch als Kopfdatensatz bezeichnet und beinhaltet die Namen der einzelnen Spalten. Soll eine CSV-Datei in ein XML-basiertes Format überführt werden, wird erneut das RelationalDataFormat genutzt. Jedoch erzeugt der CSVDataConverter kein Element vom Typ *tRDBTableMetaData*. Stattdessen werden Metadaten über die CSV-Datei mithilfe des Elementes *csvTableMetaData* festgehalten. Bisher wird lediglich der Dateiname unter dem Attribut *filename* gespeichert. Die anderen Attribute (*separator*, *quoteChar*, *escapeChar* und *strictQuotes*) werden derzeit nicht genutzt. Für jede Zeile wird ein Element *row* erzeugt. Als Kindelemente werden die einzelnen Spalten und ihre Werte gespeichert. Besteht eine Zeile aus drei Spalten, werden für jedes Element vom Typ *tRow* drei Kindelemente vom Typ *tColumn* erzeugt. Der Inhalt eines solchen Kindelementes ist der eigentliche Wert der entsprechenden Spalte. Als Attribut wird zusätzlich der Spaltenname festgehalten, der aus der Kopfzeile der CSV-Datei ausgelesen wird.

Sollen die einzelnen Datensätze aus dieser Struktur wieder extrahiert werden, wird eine neue Datei erzeugt. Der Dateiname wird aus dem Inhalt des Attributs *filename* ausgelesen. Für jedes Element vom Typ *tRow* wird eine neue Zeile generiert. Den Inhalt einer Zeile bilden die einzelnen Spaltenwerte, die durch ein festes Zeichen voneinander getrennt werden. Ein *row* Element beinhaltet als Kindelemente die einzelnen Spalten und ihre Werte. Jedes *column* Element speichert als Attribut den zugehörigen Spaltennamen. Deshalb wird, wenn das erste *row* Element verarbeitet wird, zusätzlich der Kopfdatensatz erzeugt und als erste Zeile der neuen Datei eingefügt. Anschließend kann die erzeugte Datei durch die WriteDataBack Operation eines geeigneten Konnektors weiter verarbeitet werden.

Der dritte Konverter, der **RandomFileDataConverter**, wird für den Zugriff auf Dateisysteme und das Laden beliebiger Dateiformate benötigt. Durch eine RetrieveData Operation wird beim Zugriff auf ein Dateisystem entweder eine einzelne Datei oder ein Ordner mit mehreren Dateien extrahiert. Der RandomFileDataConverter überführt diese Daten in das XML-basierte RandomFileDataFormat. Ausschnitte aus dem zugehörigen XML Schema sind in Abbildung 5.2 dargestellt.

Das Element *RandomFileDataFormat* beinhaltet ebenfalls Attribute. Zum einen werden erneut Informationen über das verwendete Datenformat gespeichert. Zum anderen wird ein Pfad festgehalten. Dieser Pfad verweist entweder auf die einzelne Datei oder auf den übergeordneten Ordner. Für jede Datei wird ein Element vom Typ *tFile* erzeugt. In jeweils einem

Attribut wird der Name der Datei sowie eine HTML-Kodierung des Inhalts gespeichert. Sollen die einzelnen Dateien aus dieser Struktur wieder extrahiert werden, wird ein temporäres Verzeichnis erstellt. In diesem Verzeichnis werden dann die einzelnen Dateien gespeichert, jeweils mit dem Namen und dem Dateiinhalt aus den Attributen *name* bzw. *content*. Das erzeugte Verzeichnis kann anschließend von der geeigneten WriteDataBack Operation eines geeigneten Konnektors weiter verarbeitet werden.

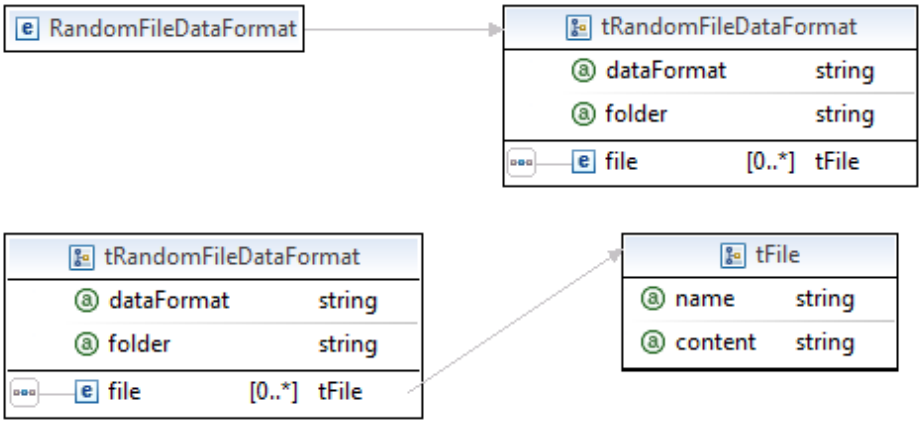

Abbildung 5.2.: Auszüge aus der RandomFileDataFormat Definition

Betrachtet man die verschiedenen Konverter bzw. Datenformate hinsichtlich ihrer Gemeinsamkeiten und Unterschiede, fallen zwei Punkte auf. Zum einen werden bei beiden Formaten Informationen über das vervendete Datenformat als Attribut des Wurzelelementes gespeichert. Zum anderen nutzen der RelationalDataConverter und der CSVDataConverter dasselbe Datenformat. Lediglich zur Speicherung der Metadaten werden unterschiedliche Elemente erzeugt. Gemeinsamkeiten zwischen den verschiedenen Formaten wurden demzufolge schon teilweise zusammengefasst. Nun stellt sich die Frage, wie eine weitere Vereinheitlichung aussehen könnte. Ziel ist es, eine zukünftige Erweiterung der Implementierung von SIMPL um weitere Datenformate (z.B. für neue Datenquellen) zu vereinfachen. Im Rahmen dieser Studienarbeit wird die folgende Idee umgesetzt. Für das Workflow Datenformat wird ein Basistyp entwickelt. Dieser fasst die Gemeinsamkeiten zwischen den verschiedenen Datenformaten zusammen. Bei Bedarf kann dieser Basistyp erweitert werden. So wird es Erweiterungen für das RelationalDataFormat sowie das RandomFileDataFormat geben. Die konkrete Umsetzung dieser Idee wird in Abschnitt 6.1 beschrieben.

5.1.2. Konnektoren

Bisher wurden sechs Konnektoren (vgl. Abschnitt 4.3) umgesetzt. Mithilfe dieser Konnektoren kann auf relationale Datenbanken sowie auf das Windows Dateisystem zugegriffen werden. Der Konnektor für das Windows Dateisystem unterscheidet sich grundlegend von

den Konnektoren für relationale Datenbanken. Aus diesem Grund werden nur die Konnektoren für relationale Datenbanken hinsichtlich ihrer Gemeinsamkeiten bzw. Unterschiede analysiert.

Alle für relationale Datenbanken implementierten Konnektoren basieren auf JDBC (Java Database Connectivity) [Ora]. JDBC bietet eine einheitliche Schnittstelle, um auf Datenbanken verschiedener Hersteller zuzugreifen und ist mit ODBC (Open Database Connectivity) vergleichbar. Mithilfe von JDBC können Datenbankverbindungen aufgebaut und verwaltet werden. Desweiteren werden SQL-Anweisungen an die Datenbank weitergeleitet und Ergebnisse dem aufrufenden Programm zur Verfügung gestellt. Für jede Datenbank müssen vom Hersteller geeignete Treiber, die die benötigten Funktionen implementieren, bereitgestellt werden. Im Folgenden werden nun die vier DM Operationen IssueCommand, RetrieveData, WriteDataBack und QueryData betrachtet. Alle weiteren von den Konnektoren implementierten Operationen, wie z.B. GetMetaData, unterscheiden sich nicht zwischen den verschiedenen Datenbanken. Daher werden diese Operationen hier nicht weiter betrachtet.

Listing 5.2 Implementierung der IssueCommand Operation [SIM]

```
public boolean issueCommand(DataSource dataSource, String statement)
    throws ConnectionException {
  boolean success = false;
  Connection conn = openConnection(dataSource.getAddress(), dataSource
      .getAuthentication().getUser(), dataSource.getAuthentication().getPassword());
  try {
    Statement stat = conn.createStatement();
    stat.execute(statement);
    conn.commit();
    success = true;
    stat.close();
  } catch (Throwable e)
    try {
      conn.rollback();
    } catch (SQLException e1) {
      e1.printStackTrace();
    }
  }
  closeConnection(conn);
  return success;
}
```

Die **IssueCommand** Operation ist bei allen Konnektoren auf dieselbe Weise implementiert. Als erstes wird eine Verbindung mit der Datenbank hergestellt. Anschließend wird ein Statement-Objekt erzeugt. Durch dieses Objekt können SQL-Anweisungen der Datenbank übergeben werden. Die vom Nutzer spezifizierte Anweisung wird übergeben und anschließend wird ein *COMMIT* ausgeführt. Danach wird die Verbindung geschlossen. Sollte während der Ausführung der Operation ein Fehler auftreten, wird ein *ROLLBACK* versucht. Listing 5.2 zeigt exemplarisch den Quellcode einer IssueCommand Operation.

Listing 5.3 Implementierung der RetrieveData Operation [SIM]

```
public RDBResult retrieveData(DataSource dataSource, String statement)
    throws ConnectionException {
  Connection connection = openConnection(dataSource.getAddress(), dataSource
      .getAuthentication().getUser(), dataSource.getAuthentication().getPassword());
  Statement connStatement = null;
  ResultSet resultSet = null;
  RDBResult rdbResult = null;
  try {
    connStatement = connection.createStatement();
    resultSet = connStatement.executeQuery(statement);
    rdbResult = new RDBResult();
    rdbResult.setDbMetaData(connection.getMetaData());
    rdbResult.setResultSet(resultSet);
    rdbResult.setDataSource(dataSource);
  } catch (SQLException e) {
    e.printStackTrace();
  }
  if (rdbResult == null) {
    try {
      connStatement.close();
      connection.close();
    } catch (SQLException e) {
      e.printStackTrace();
    }
  }
  return rdbResult;
}
```

Die **RetrieveData** Operation ist ebenfalls bei allen Konnektoren gleich. Es wird eine Verbindung mit der Datenbank aufgebaut und ein Statement-Objekt erzeugt. Die spezifizierte Anweisung wird der Datenbank übergeben. Da es sich um eine RetrieveData Operation handelt, werden Daten generiert. Das Ergebnis wird in einem JDBC ResultSet-Objekt gespeichert. Der zugehörige Konverter erwartet die Daten wiederum als RDBResult. Dieses Objekt wird anschließend erzeugt und beinhaltet das eigentliche ResultSet-Objekt sowie weitere Metadaten. Sind durch die vom Nutzer spezifizierte Anweisung keine Daten generiert wurden, wird die Verbindung geschlossen. Ansonsten erledigt dies der zugehörige Konverter zu einem späteren Zeitpunkt. Der Quellcode einer RetrieveData Operation ist exemplarisch in Listing 5.3 dargestellt.

Auch bei der **WriteDataBack** Operation gibt es keine wesentlichen Unterschiede. Als erstes wird eine Verbindung mit der Datenbank aufgebaut und wiederum ein Statement-Objekt erzeugt. In Abschnitt 5.1.1 wird erläutert, dass der RDBDataConverter ein Liste mit UPDATE- und INSERT-Ausdrücken erzeugt, um Daten in eine Datenbank zu schreiben. Die Variable *target* referenziert das Ziel bzw. die Tabelle, die verändert werden soll. Diese Liste wird nun abgearbeitet. Beinhaltet die Variable *target* keinen Wert, so wird mithilfe der UPDATE-Ausdrücke die ursprüngliche Tabelle aktualisiert. Andernfalls wird das referenzierte Ziel (Schemaname.Tabellenname) in die einzelnen INSERT-Ausdrücke eingefügt, bevor diese der Datenbank übergeben werden. Zuletzt werden ein *COMMIT* ausgeführt und die Verbindung

geschlossen. Sollte ein Fehler auftreten, wird genau wie bei der IssueCommand Operation ein *ROLLBACK* versucht.

Bei der **QueryData** Operation hingegen gibt es Unterschiede bei den einzelnen Implementierungen. Mithilfe dieser Operation werden Daten über eine SQL Anweisung extrahiert und das Ergebnis wiederum an einem referenzierten Ort (*data container*) bereitgestellt. Dazu muss die benötigte Tabelle erzeugt und mit Inhalt gefüllt werden. An dieser Stelle unterscheiden sich die verschiedenen Implementierungen, ansonsten sind sie, analog zu den anderen Operationen, gleich. Der DB2RDBConnector, der DerbyRDBConnector sowie der EmbDerbyRDBConnector erzeugen jeweils zwei SQL Anweisungen, die nacheinander ausgeführt werden:

- **CREATE TABLE** *target***AS (***statement***) WITH NO DATA**
- **INSERT INTO** *target statement*

Durch die erste SQL Anweisung wird die benötigte Tabelle erzeugt. Die übergebene SQL Anweisung (*statement*) wird angegeben, um die benötigten Attribute zu erzeugen. Der Ausdruck *WITH NO DATA* sorgt dafür, dass die Ergebnismenge noch nicht berechnet und damit nicht gespeichert wird. Die zweite SQL Anweisung füllt diese Tabelle dann mit dem von der übergebenen SQL Anweisung (*statement*) erzeugten Inhalt. Soll beispielsweise der Inhalt einer Tabelle *TAB1* in die noch nicht vorhandene Tabelle *TAB2* kopiert werden, sehen die beiden SQL Anweisungen folgendermaßen aus:

- **CREATE TABLE** *TAB2* **AS (***SELECT * FROM TAB1***) WITH NO DATA**
- **INSERT INTO** *TAB2 SELECT * FROM TAB1*

Der MySQLRDBConnector benötigt hingegen nur eine SQL Anweisung. Durch diese *eine* SQL Anweisung wird die benötigte Tabelle erzeugt und direkt mit Inhalt gefüllt. Die Anweisung hat die folgende Struktur:

- **CREATE TABLE** *target statement*

In Bezug auf obiges Beispiel würde die Anweisung folgendermaßen aussehen:

- **CREATE TABLE** *TAB2 SELECT * FROM TAB1*

Auch der PostgreSQLRDBConnector benötigt nur eine SQL Anweisung. Jedoch wird zusätzlich ein *AS* eingefügt:

- **CREATE TABLE** *target* **AS** *statement*

Der PostgreSQLRDBConnector würde für obiges Beispiel folgende SQL Anweisung erzeugen:

- **CREATE TABLE** *TAB2* **AS** *SELECT * FROM TAB1*

Alle Konnektoren erzeugen zuerst die benötigten Anweisungen und bauen anschließend eine Verbindung mit der Datenbank auf. Auch hier wird anschließend ein Statement-Objekt erzeugt, mit dem die generierten Anweisungen der Datenbank übergeben werden. Danach wird ein *COMMIT* ausgeführt. Im Fehlerfall wird auch hier ein *ROLLBACK* versucht. Listing 5.4 zeigt exemplarisch die Implementierung der QueryData Operation für PostgreSQL Datenbanken.

Listing 5.4 Implementierung der QueryData Operation für PostgreSQL Datenbanken [SIM]

```
public boolean queryData(DataSource dataSource, String statement, String target)
    throws ConnectionException {
  boolean success = false;
  StringBuilder createTableStatement = new StringBuilder();
  createTableStatement.append("CREATE TABLE");
  createTableStatement.append(" ");
  createTableStatement.append(target);
  createTableStatement.append(" AS ");
  createTableStatement.append(statement);
  Connection conn = openConnection(dataSource.getAddress(), dataSource
      .getAuthentication().getUser(), dataSource.getAuthentication().getPassword());
  try {
    Statement createState = conn.createStatement();
    createState.execute(createTableStatement.toString());
    conn.commit();
    createState.close();
    closeConnection(conn);
    success = true;
  } catch (Throwable e) {
      e.printStackTrace();
    try {
      conn.rollback();
    } catch (SQLException e1) {
      e1.printStackTrace();
    }
  }
  return success;
}
```

Listing 5.5 Laden des JDBC Treibers [SIM]

```
  Connection connect = null;
  try {
    Class.forName("org.postgresql.Driver");
    StringBuilder uri = new StringBuilder();
    uri.append("jdbc:postgresql://");
    uri.append(dsAddress);
    try {
      connect = DriverManager.getConnection(uri.toString(), user, password);
      connect.setAutoCommit(false);
    } catch (SQLException e) {
        e.printStackTrace();
    }
```

Ein Unterschied zwischen den einzelnen Konnektoren wird oben noch nicht erläutert. Bevor die Verbindung zu einer Datenbank aufgebaut werden kann, muss ein geeigneter JDBC Treiber geladen werden. Die Operation OpenConnection lädt den geeigneten Treiber und stellt die Verbindung mit der Datenbank her. Der Name des Treibers ist bei jeder Datenbank anders. Listing 5.5 zeigt den Verbindungsaufbau für PostgreSQL Datenbanken. Dort wurde der Treiber *org.postgresql.Driver* geladen.

Im Großen und Ganzen sind die einzelnen Konnektoren sehr ähnlich. Bis auf die Query-Data Operation und das Laden des JDBC Treibers sind nahezu alle Operationen identisch. Aufgrund dessen werden im Rahmen dieser Studienarbeit die Konnektoren für relationale Datenbanken durch einen einzigen Konnektor ersetzt. Dieser implementiert die DM Operationen für alle relationalen Datenbanken, deren Kommunikation auf JDBC aufbaut. Auf die konkrete Umsetzung wird in Abschnitt 6.2 eingegangen.

Der Vollständigkeit halber und da das Thema wieder in Kapitel 6.5 aufgegriffen wird, wird nun noch kurz auf *data container references* eingegangen. Bei der WriteDataBack und QueryData Operation referenziert die Variable *target* einen Container, in dem die jeweiligen Daten gespeichert werden sollen. Da es bisher nicht möglich ist einzelne Container über logische Namen zu referenzieren (vgl. Abschnitt 4.2), wird diesen beiden Operationen der in der Workflow Aktivität vom Nutzer festgelegte lokale Bezeichner des Containers übergeben. Bei relationalen Datenbanken ist dies der Tabellenname in Verbindung mit dem zugehörigen Schemanamen. Der Konnektor für Dateisysteme wurde aufgrund der großen Unterschiede im Rahmen dieser Studienarbeit nicht analysiert. Dort könnte eine Datei bzw. ein Ordner über den vollständigen Pfad eindeutig referenziert werden.

5.2. Analyse der neuen Datenquellen

Die folgenden Datenbanken werden neu in das SIMPL-Rahmenwerk integriert:

- MonetDB (Version 5)
- TinyDB
- Exist
- XML Transaction Coordinator (XTC)
- MonetDB/XQuery

MonetDB ist ein hauptspeicherbasiertes relationales Datenbanksystem (vgl. Abschnitt 2.4.1). Mithilfe von TinyDB (vgl. Abschnitt 2.5.1) können Daten aus einem Sensornetz extrahiert werden. Bei den restlichen drei Datenbanken handelt es sich um XML-Datenbanken (vgl. Abschnitt 2.4.2). In diesem Kapitel werden die einzelnen Datenbanken analysiert. Es wird untersucht, wie aus Java Programmen heraus ein Zugriff stattfinden kann und ob bereits vorhandene Datenformate oder Konnektoren für die Realisierung der SIMPL Core Operationen genutzt werden können.

5.2.1. MonetDB (Version 5)

Analog zu den bisher integrierten relationalen Datenbanken geschieht die Kommunikation auch bei MonetDB über JDBC. Um diese Datenbank zu integrieren, muss demzufolge kein neues Workflow Datenformat definiert werden. Datensätze können durch den RDB-DataConverter in das RelationalDataFormat überführt werden. In Bezug auf die bisherige Umsetzung des SIMPL-Rahmenwerks müsste ein neuer Konnektor implementiert werden. Die Zugriffsmechanismen wurden erst einmal prototypisch implementiert, ohne sie in das SIMPL-Rahmenwerk zu integrieren. Die für die SIMPL Core Operationen relevanten Aspekte dieser Zugriffsmechanismen haben dabei aber keine neuen Unterschiede zu den anderen relationalen Datenbanken aufgewiesen. Deswegen kann der in Abschnitt 5.1.1 vorgestellte einheitliche Konnektor, auch für MonetDB verwendet werden. Die konkrete Umsetzung des Konnektors wird in Abschnitt 6.2 betrachtet.

5.2.2. TinyDB

Bisher unterstützt das SIMPL-Rahmenwerk keinen Zugriff auf Sensornetze. Dies soll durch die Integration von TinyDB geändert werden. Mithilfe von TinyDB können Daten aus einem Sensornetz extrahiert werden (vgl. Abschnitt 2.5.2). Die gewünschten Daten werden durch TinySQL Anweisungen spezifiziert.

Um aus einem Java-Programm heraus eine Datenbankverbindung aufzubauen, wird eine Java-API (Application Programming Interface) angeboten. Das Herzstück dieser API ist die Klasse *TinyDBMain*. Durch sie ist es möglich Abfragen zu starten bzw. abzubrechen und Listener zu registrieren. Listing 5.6 zeigt exemplarisch ein kleines Java-Programm, das eine TinySQL Anweisung an TinyDB übergibt und die Ergebnisse auf der Konsole ausgibt. Als Erstes wird die Anweisung transformiert. Anschließend wird diese dem Netzwerk übergeben. Anweisungen werden in bestimmten Zeitintervallen evaluiert. Um auf Ereignisse, insbesondere das Generieren neuer Ergebnisse, reagieren zu können, wird ein Listener registriert. Werden neue Ergebnisse generiert, werden alle Listener benachrichtigt. Die einzelnen Listener können dann auf diese Ereignisse reagieren.

Zusätzlich bietet TinyDB noch die Möglichkeit die Ergebnisse einer TinySQL Anweisung im Flash-Speicher der einzelnen Motes abzulegen. Dazu muss ein Buffer erzeugt werden, welcher anschließend mit Daten gefüllt wird. Diese Buffer können über einen eindeutigen Namen, der bei der Erstellung der Buffer angegeben werden muss, referenziert werden. In Bezug auf das SIMPL-Rahmenwerk könnten die einzelnen Buffer als Container angesehen werden.

Da die Implementierungen der SIMPL Core Operationen grundlegend unterschiedlich zu den JDBC-basierten Konnektoren für relationale Datenbanken sind, wird in dieser Studienarbeit ein neuer Konnektor entwickelt. Auf die konkrete Umsetzung wird in Kapitel 6.3 eingegangen. Ergebnisse von TinySQL Anweisungen können als Tabelle dargestellt werden. Um diese Daten in ein XML-basiertes Format zu überführen, kann das RelationalDataFormat genutzt bzw. erweitert werden. Dazu ist es nötig einen neuen Konverter zu implementieren.

Dieser Konverter überführt Daten, die im Ausgabeformat des Konnektors vorliegen, in das XML-basierte RelationalDataFormat. Eine Transformation in die andere Richtung macht hingegen keinen Sinn, da Daten aus einem Sensornetz nur extrahiert werden können.

Ein weiteres Problem stellen die generischen Operationen des SIMPL Cores dar. Diese und die zugehörigen Workflow Aktivitäten werden synchron aufgerufen und erwarten sofort ein Ergebnis. TinySQL Anfragen hingegen werden in bestimmten Intervallen (vgl. Abschnitt 2.5.2) evaluiert. Idealerweise müsste ein „Send and Forget"-Ansatz implementiert werden. Jedesmal, wenn der Listener ein neues Ergebnis erhalten hat, wird eine Aktivität bzw. ein BPEL Event Handler benachrichtigt. Auf ein mögliches *Workaround* wird in Abschnitt 6.3 eingegangen.

Listing 5.6 Zugriff auf TinyDB [Sys]

```
public class DemoApp implements ResultListener {
	public DemoApp() {
		try {
			TinyDBMain.initMain(); // initialize
			// parse the query
			q = SensorQueryer.translateQuery("select light", (byte) 1);
			// inject the query, registering ourselves as a listener for result
			TinyDBMain.injectQuery(q, this);
		} catch (IOException e) {
			System.out.println("Network error.");
		} catch (ParseException e) {
			System.out.println("Invalid Query.");
		}
	}
	/* ResultListenr method called whenever a result arrives */
	public void addResult(QueryResult qr) {
		Vector v = qr.resultVector();
		for (int i = 0; i < v.size(); i++) {
			System.out.print("\t" + v.elementAt(i) + "\t|");
		}
	}
	public static void main(String argv[]) {
		new DemoApp();
	}
	TinyDBQuery q;
}
```

5.2.3. XML-Datenbanken

Des Weiteren soll das SIMPL-Rahmenwerk im Rahmen dieser Studienarbeit um den Zugriff auf XML-Datenbanken (vgl. Abschnitt 2.4.2) erweitert werden. **Exist** ist eine native XML-Datenbank. Als Abfragesprache wird die in Abschnitt 2.4.4 vorgestellte Sprache XQuery unterstützt. Für den Zugriff aus einem Java-Programm heraus wird die XML:DB API [XI] unterstützt. Diese API wurde entwickelt, um einheitlich auf native XML-Datenbanken zuzugreifen und ist mit JDBC für relationale Datenbanken vergleichbar. Bisher konnte

sich diese API jedoch noch nicht als de-facto Standard durchsetzen. Der letzte *Working Draft* stammt aus dem Jahr 2001. Auch **XTC** ist eine native XML-Datenbank und wurde von der Universität Kaiserslautern entwickelt. Als Abfragesprache wird ebenfalls XQuery unterstützt. Für den Zugriff aus Java-Programmen heraus steht eine proprietäre Java-API [Kai] zur Verfügung. **MonetDB/XQuery** dient ebenfalls zur Speicherung und Verwaltung von XML-Dokumenten. Die Entwicklung dieser Datenbank wurde im März 2011 eingestellt. Bei dieser Datenbank geschieht der Zugriff über JDBC. Jedoch werden der Datenbank keine SQL Anweisungen übermittelt, sondern XQuery Ausdrücke.

Werden XQuery Ausdrücke ausgewertet, ist das Ergebnis ein XML-Dokument bzw. XML-Fragmente. Bisher gibt es in SIMPL kein Workflow Datenformat, das beliebige XML-Fragmente speichern kann. Demzufolge muss ein neues Datenformat entwickelt und ein neuer Konverter implementiert werden. Da der Zugriff auf die verschiedenen Datenbanken über unterschiedliche APIs geschieht, müssen für diese drei Datenbanken drei neue Konnektoren implementiert werden.

Um auf ein XML-Dokument innerhalb einer XML-Datenbank zuzugreifen, muss dieses Dokument eindeutig referenziert werden. Bei den einzelnen XML-Datenbanken sehen die konkreten Identifikatoren bzw. lokalen Bezeichner folgendermaßen aus: Bei MonetDB/XQuery muss, wenn ein Dokument der Datenbank übergeben wird, ein Dokumentname festgelegt werden. Dieser muss eindeutig sein. Es darf kein weiteres Dokument mit dem gleichen Namen innerhalb der gesamten Datenbank geben. Mithilfe dieses Namens kann das Dokument z.B. innerhalb eines XQuery Ausdrucks referenziert werden. Bei Exist und XTC wird, wenn ein Dokument der Datenbank übergeben wird, standardmäßig der Dateiname als Dokumentname verwendet. Es darf in der zugehörigen *Collection* (vgl. Abschnitt 2.4.2) kein weiteres Dokument mit dem gleichen Namen geben. Demzufolge wird zur eindeutigen Referenzierung eines Dokuments der Dokumentname sowie der Name der zugehörigen Collection benötigt. Auf die konkrete Umsetzung wird in Abschnitt 6.4 eingegangen.

6. Umsetzung

Die konkrete Umsetzung der im letzten Kapitel genannten Ideen wird nun beschrieben. In den Abschnitten *Ein Basistyp für alle Workflow Datenformate* und *Ein Konnektor für relationale Datenbanken* wird auf konzeptionelle Änderungen sowie auf Änderungen an der bestehenden Implementierung eingegangen. Die Integration der neuen Datenquellen wird in den Kapiteln *TinyDB* und *XML-Datenbanken* beschrieben.

6.1. Ein Basistyp für alle Workflow Datenformate

Die bestehenden Workflow Datenformate RelationalDataFormat und RandomFileDataFormat werden weiter zusammengefasst. Dazu wird der neue Basistyp *tDataFormat* eingeführt. Listing 6.3 zeigt Ausschnitte aus dem resultierenden XML Schema. Die Gemeinsamkeiten zwischen den verschiedenen Datenformaten werden im Basistyp zusammengefasst. Sowohl beim RelationalDataFormat als auch dem RandomFileDataFormat werden Informationen über das verwendete Datenformat als Attribut des Wurzelelementes gespeichert. Der komplexe Typ *tDataFormat* beinhaltet nun dieses gemeinsame Attribut. Dieser Basistyp wird von den komplexen Typen *tRelationalDataFormat* und *tRandomFileDataFormat* erweitert. Diese komplexen Typen beinhalten die Elemente und Attribute, die alle Workflow Datenformate aufweisen, nicht mehr. Die gemeinsamen Eigenschaften des Basistyps werden diesen beiden Typen vererbt. Zusätzlich gibt es noch die beiden Elemente *relationalDataFormat* und *randomFileDataFormat*, um die komplexen Typen zu instanziieren. Die einzelnen Konverter werden in der Tabelle Dataconverters gespeichert (vgl. Abschnitt 4.2). Bisher wird in dieser Tabelle unter dem Attribut XML_Schema die zugehörige Definition des Workflow Datenformates gespeichert. Dies wird in Zunkunft geändert. Die Tabelle **Dataformat_Types** wird neu angelegt. In dieser Tabelle werden die Definitionen der einzelnen Workflow Datenformate sowie der Basistyp hinterlegt. In der Tabelle Dataconverters wird für jeden registrierten Konverter nur noch eine Referenz auf das zugehörige Workflow Datenformat in dieser neuen Tabelle gespeichert. Dazu muss sowohl das Resource Management als auch dessen Web Interface geändert werden. Wird in Zunkunft über das Web Interface ein neuer Konverter registriert, werden dem Nutzer die zur Verfügung stehenden Workflow Datenformate angezeigt. Dieser kann dann ein passendes Workflow Datenformat wählen. Des Weiteren wird eine Möglichkeit geschaffen neue Workflow Datenformate über das Web Interface zu registrieren.

Aus dem neu definierten Basistyp resultieren mehrere Vorteile: Die Gemeinsamkeiten zwischen den verschiedenen Datenformaten sind direkt sichtbar und Unterschiede können über Erweiterungen realisiert werden. Bei der Definition eines Workflowmodells muss der Nutzer,

Listing 6.1 Vereinheitlichung der verschiedenen Workflow Datenformate

```
<xsd:complexType name="tDataFormat">
  <xsd:attribute name="dataFormat" type="xsd:string"></xsd:attribute>
</xsd:complexType>

<xsd:complexType name="tRelationalDataFormat">
  <xsd:complexContent>
    <xsd:extension base="tDataFormat">
      <xsd:sequence>
        <xsd:element name="dataFormatMetaData" type="tDataFormatMetaData"
          maxOccurs="1" minOccurs="0">
        </xsd:element>
        <xsd:element name="table" type="tTable" maxOccurs="unbounded"
          minOccurs="0">
        </xsd:element>
       </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="tRandomFileDataFormat">
  <xsd:complexContent>
    <xsd:extension base="tDataFormat">
      <xsd:sequence>
        <xsd:element name="file" type="tFile" maxOccurs="unbounded"
          minOccurs="0">
        </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="folder" type="xsd:string"></xsd:attribute>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:element name="relationalDataFormat" type="tRelationalDataFormat"></xsd:element>

<xsd:element name="randomFileDataFormat" type="tRandomFileDataFormat"></xsd:element>
```

wenn er eine WriteDataBack oder QueryData Aktivität verwendet, eine geeignete *data set variable* (vgl. Abschnitt 3.3) auswählen. Um geeignete Variablen anzuzeigen muss bisher nach verschiedenen Typen gefiltert werden. In Zukunft kann einfach der Basistyp zur Filterung verwendet werden. Sollten in Zukunft weitere Datenformate hinzukommen, muss zumindest an dieser Stelle keine Änderung mehr durchgeführt werden.

6.2. Ein Konnektor für relationale Datenbanken

In Abschnitt 5.1.2 wird die Idee erläutert, dass es in Zunkunft nur noch einen Konnektor für relationale und JDBC-basierte Datenbanken geben soll. Es gibt zwei Probleme, die dabei gelöst werden müssen: Zum einen wird bei allen Datenbanken ein anderer Treiber

verwendet. Zum anderen müssen Unterschiede bzgl. des Erzeugens einer neuen Tabelle bei der QueryData Operation aufgelöst werden.

Um das erste Problem zu lösen, wird das Resource Management erweitert. In der Tabelle **Datasources** werden zusätzliche Informationen gespeichert (vgl. Abschnitt 4.2):

- **Datasources(**
  Id, Connector_Id, Logical_Name, Security_Username, Security_Password, Interface_Description, Properties_Description, Connector_Properties_Description, ***Driver_Name*****)**

Der Name des zugehörigen JDBC Treibers wird in der Spalte **Driver_Name** festgehalten. Der neue Konnektor kann nun mithilfe dieser Information den passenden Treiber laden. Diese Information wird bisher nur für relationale Datenbanken benötigt. Werden andere Datenquellen im Resource Management registriert, wird für dieses Attribut ein *NULL*-Wert gespeichert. Unter dem Attribut Connector_Properties_Description werden zusätzliche Eigenschaften, die ein geeigneter Konnektor aufweisen muss, gespeichert. Bisher wird der Typ der Datenquelle (z.B. Database), der Subtyp (z.B. MySQL), die Abfragesprache (z.B. SQL) sowie das Workflow Datenformat (z.B. RelationalDataFormat) gespeichert. Für jeden Konnektor werden entsprechend in der Tabelle **Connectors** die Eigenschaften, die ein Konnektor aufweist, in dem Attribut Properties_Description gespeichert. Mit einem Vergleich dieser Informationen kann dann für eine bestimmte Datenquelle ein passender Konnektor gewählt werden. Im Rahmen dieser Arbeit wird noch eine weitere Eigenschaft, der **API_Type**, gespeichert. Listing 6.2 zeigt exemplarisch eine Connector Properties Description. Bei relationalen und JDBC-basierten Datenbanken wird z.B. JDBC als **API_Type** gespeichert. Der neue Konnektor für relationale Datenbanken wird diese Eigenschaft teilen. Aufgrund dessen kann sofort der neue Konnektor als geeigneter Konnektor identifiziert werden. Generell wird nur noch der **API_Type** für das Matching zwischen Datenquellen und Konnektoren verwendet. Bei anderen Datenquellen wird entsprechend ein anderer **API_Type** gespeichert. Um diese Funktionalität zu integrieren, muss zum einen der neue Konnektor für relationale Datenbanken implementiert und zum anderen das Resource Management angepasst werden.

Listing 6.2 Beispiel einer Connector Properties Description

```
<connector_properties_description>
  <type>Database</type>
  <subType>MySQL</subType>
  <language>SQL</language>
  <dataFormat>RDBDataFormat</dataFormat>
  <api_type>JDBC</api_type>
</connector_properties_description>
```

Um die Unterschiede bei der QueryData Operation aufzulösen, wird vorläufig ein *Workaround* implementiert. Die benötigte Tabelle muss schon vor der Ausführung der Operation vorhanden sein, wofür im Zweifelsfall der Workflow-Modellierer selbst sorgen muss. Die QueryData Operation füllt diese Tabelle dann nur noch über einen INSERT-Ausdruck mit Daten. Dieser INSERT-Ausdruck ist für alle bisher integrierten relationalen Datenbanken

gleich. In einem weiteren Schritt könnte versucht werden, dieses Problem durch zusätzliche Metadaten im Resource Management zu lösen. So könnte in Zukunft ein entsprechendes Lebenszyklus-Management für Datencontainer entwickelt werden.

6.3. TinyDB

In Abschnitt 5.2.2 wird auf Probleme, die sich beim Zugriff auf TinyDB ergeben, eingegangen. Da Daten aus einem Sensornetz nur extrahiert werden können, wird die WriteDataBack Operation nicht unterstützt. Im Rahmen dieser Studienarbeit gelang es leider nicht, Buffer über die von TinyDB unterstützte Java API zu erzeugen. Die genaue Fehlerursache bzw. ob dies überhaupt möglich ist, konnte nicht festgestellt werden. Ansonsten hätte die IssueCommand Operation Buffer erstellen bzw. löschen und die QueryData Operation bzw. eine zweite IssueCommand Operation Daten in diesen Buffer bereitstellen können. Demzufolge werden auch die Operationen IssueCommand und QueryData noch nicht unterstützt. Die RetrieveData Operation übergibt die vom Nutzer spezifizierte TinySQL Anweisung und wartet, bis der zugehörige Listener erstmalig über das Eintreffen neuer Daten informiert wird. Anschließend wird die Abfrage gestoppt und der erhaltene Datensatz zurückgegeben.

Des Weiteren wird ein neuer Konverter implementiert. Dieser nutzt das im Rahmen dieser Studienarbeit geänderte RelationalDataFormat (vgl. Listing 6.3), um den extrahierten Datensatz in das XML-basierte Format zu überführen. Der Konverter erzeugt *row* Elemente für die einzelnen Zeilen. Sonstige Metadaten über die Tabelle werden nicht gespeichert. Da Daten aus einem Sensornetz nur extrahiert werden können, wird im neuen Konverter auch eine Transformation in die Rückrichtung nicht implementiert.

In weiteren Arbeiten könnte versucht werden, ein asynchrones Verfahren zu implementieren. Der Nutzer startet einmalig eine Abfrage und eine Workflow Aktivität bzw. ein BPEL Event Handler wird jedesmal vom SIMPL Core benachrichtigt, wenn neue Daten eingetroffen sind.

6.4. XML-Datenbanken

In Abschnitt 5.2.3 werden die zu integrierenden XML-Datenbanken analysiert. Alle Datenbanken unterstützen eine andere API. Aus diesem Grund müssen drei neue Konnektoren implementiert werden. Die generischen Operationen des SIMPL Cores werden folgendermaßen umgesetzt: Die IssueCommand Operation übergibt eine vom Nutzer spezifizierte Anweisung der jeweiligen Datenbank. Eine Benachrichtigung über den Erfolg bzw. Misserfolg wird zurückgegeben. Die RetrieveData Operationen erhält einen XQuery Ausdruck und führt diesen auf der Datenbank aus. Dieser Ausdruck muss Daten generieren. Als Resultat wird entweder ein ganzes XML-Dokument oder ein XML-Fragment zurückgegeben. Die WriteDataBack Operation implementiert folgenden Ablauf. Die übergebenen Daten werden in der Datenbank in einem neuen XML-Dokument gespeichert. Dazu wird die Variable *target* benötigt. Diese Variable beinhaltet den Namen einer Collection sowie den Namen

eines Dokuments. Sollte in der entsprechenden Collection bereits ein Dokument mit diesem Namen existieren, wird dieses Dokument gelöscht. Anschließend wird ein neues Dokoment aus den übergebenen Daten erzeugt und an dieser Stelle gespeichert. Sollte an der referenzierten Stelle kein Dokument vorhanden sein, wird direkt das neue Dokument erzeugt und gespeichert. Die QueryData Operation ist sehr ähnlich. Daten werden mithilfe eines XQuery Ausdrucks extrahiert und in einem neuem Dokument bereitgestellt. Die Variable *target* spezifiziert auch hier das eigentliche Ziel. Bei MonetDB/XQuery gibt es geringfügige Unterschiede. Bei dieser Datenbank reicht der Name eines Dokuments aus, um ein Dokument eindeutig zu identifizieren. Aus diesem Grund darf in der gesamten Datenbank kein Dokument mit dem gleichen Namen vorhanden sein. Ansonsten wird dieses Dokument gelöscht und durch das neue Dokument ersetzt. Eine Banachrichtigung über Erfolg bzw. Misserfolg wird bei beiden Operation zurückgegeben.

Probleme gibt es, wenn XML-Fragmente gespeichert werden sollen, die mehrere Wurzelelemente beinhalten. Bei den meisten XML-Datenbanken ist dies nicht möglich. Dieses Problem kann auf zwei Arten gelöst werden. Entweder wird ein neues Wurzelelement, welches die anderen Elemente umschließt, eingefügt oder die zugehörige Workflow Aktivität wird über den Misserfolg der Operation benachrichtigt. Im Rahmen dieser Studienarbeit wird der zweite Ansatz favorisiert, da der erste Ansatz das XML-Fragment verändern würde.

Da es bisher in SIMPL kein Workflow Datenformat gibt, welches beliebige XML-Fragmente speichern kann, muss ein neues Datenformat definiert werden. Listing 6.3 zeigt das neue Datenformat *XMLDataFormat*.

Listing 6.3 Workflow Datenformat um beliebige XML-Fragmente zu speichern

```
<xsd:complexType name="tXMLDataFormat">
  <xsd:complexContent>
    <xsd:extension base="tDataFormat">
      <xsd:sequence>
        <xsd:element name="dataFormatMetaData" type="tDataFormatMetaData"
          maxOccurs="1" minOccurs="0">
        <xsd:element name="data" type="xsd:anyType" maxOccurs="unbounded"
          minOccurs="0">
        </xsd:element>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:element name="xmlFileDataFormat" type="tXMLFileDataFormat"></xsd:element>
```

Der Basistyp *tDataFormat* wird auch hier erweitert. Das Element *dataFormatMetaData* dient, wie bei den anderen Formaten auch, als Container für Metadaten. Der Name der Datenquelle wird als Kindelement gespeichert. Das Element *data* ist vom XSD-Typ *anyType* und kann zur Speicherung beliebiger XML-Fragmente genutzt werden. Zusätzlich wird ein neuer Konverter implementiert, der XML-Fragmente in das XMLDataFormat überführen und die einzelnen XML-Fragmente wiederum aus dieser Struktur extrahieren kann.

6.5. Datencontainer Referenzen

In Zukunft soll es möglich sein einzelne Container über einen logischen Namen zu referenzieren. Um diese Funktionalität zu verwirklichen, muss das SIMPL-Rahmenwerk erweitert bzw. angepasst werden: Es wird eine Tabelle **Datacontainers** benötigt, die Metadaten über die einzelnen Container speichert. Des Weiteren muss der SIMPL Core in Zukunft logische Namen mithilfe dieser Tabelle auf datenquellenspezifische lokale Bezeichner abbilden und die GUI des erweiterten Eclipse BPEL Designers muss angepasst werden.

Listing 6.4 XML Schema für Container Referenzen [SIM]

```
<xsd:complexType name="DataContainerReferenceType">
</xsd:complexType>

<xsd:complexType name="RelationalDatabaseDataContainerReferenceType">
  <xsd:complexContent>
    <xsd:extension base="DataContainerReferenceType">
      <xsd:sequence>
        <xsd:element name="schema" type="xsd:string" maxOccurs="1"
          minOccurs="0">
        </xsd:element>
        <xsd:element name="table" type="xsd:string" maxOccurs="1"
          minOccurs="1">
        </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="stringPattern" type="xsd:string" use="required"
          fixed="schema.table">
      </xsd:attribute>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:element name="relationalDatabaseDataContainerReference"
  type="RelationalDatabaseDataContainerReferenceType">
</xsd:element>
```

An der Umsetzung dieser Idee wird momentan gearbeitet. In der Tabelle **DataContainer_Reference_Types** (vgl. Abschnitt 4.2) werden XML Schema Typen gespeichert. Diese Typen definieren die Struktur der lokalen Bezeichner. Allerdings unterscheiden sich die Bezeichner bei den verschiedenen Typen von Datenquellen. Bei relationalen Datenbanken wird der Schema- sowie der Tabellenname benötigt. Bei XML-Datenbanken teilweise Name der Collection sowie der Dokumentname, bei MonetDB/XQuery nur der Dokumentname. Um diese Heterogenität aufzulösen wurde mittlerweile ein gemeinsamer Basistyp eingeführt. Listing 6.4 zeigt den Basistyp *DataContainerReferenceType*. Für relationale Datenbanken erweitert der Typ *RelationalDatabaseDataContainerReferenceType* den Basistyp. Der Schemaname wird im Element *schema* gespeichert. Der eigentliche Tabellenname im Element *table*. Das Attribut *stringPattern* führt diese Informationen zusammen. Mithilfe des Elementes *relationalDatabaseDataContainerReference* kann der zugehörige Typ instanziiert werden.

Die lokalen Bezeichner bei XML-Datenbanken weisen eine andere Struktur auf. Aus diesem Grund muss ein neuer Typ definiert werden. Dieser erweitert wiederum den Basistyp *DataContainerReferenceType*. Listing 6.5 zeigt den komplexen Typ *XmlDatabaseDataContainerReferenceType*. Der Name der Collection wird mithilfe des Elementes *collection* gespeichert. Das Element *document* beinhaltet den Dokumentnamen. Das Attribut *stringPattern* führt diese Informationen erneut zusammen. Das Element *collection* muss nicht zwangsläufig verwendet werden, da z.B. bei MonetDB/XQuery der Dokumentname zur Referenzierung ausreicht (vgl. Abschnitt 5.2.3).

Listing 6.5 XML Schema für Referenzen auf Container in XML-Datenbanken

```
<xsd:complexType name="XmlDatabaseDataContainerReferenceType">
  <xsd:complexContent>
    <xsd:extension base="DataContainerReferenceType">
      <xsd:sequence>
        <xsd:element name="collection" type="xsd:string" maxOccurs="1"
          minOccurs="0">
        </xsd:element>
        <xsd:element name="document" type="xsd:string" maxOccurs="1"
          minOccurs="1">
        </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="stringPattern" type="xsd:string" use="required"
          fixed="collection/document">
      </xsd:attribute>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:element name="xmlDatabaseDataContainerReference"
  type="XmlDatabaseDataContainerReferenceType">
</xsd:element>
```

7. Zusammenfassung und Ausblick

SIMPL bietet die Möglichkeit, aus Simulationsworkflows heraus, auf beliebige Datenquellen über **vereinheitlichte** Schnittstellen zuzugreifen. Die Implementierung des SIMPL Cores bietet dazu die vier generischen Operationen: IssueCommand, RetrieveData, WriteDataBack und QueryData. Konnektoren implementieren diese generischen Operationen für konkrete Datenquellen und Konverter stellen benötigte Daten-Transformationen bereit. Mithilfe der Konverter können Daten, die im Ausgabeformat eines Konnektors vorliegen, in ein SDO bzw. Daten aus einem SDO in das Eingabeformat eines Konnektors transformiert werden. Vom SIMPL-Rahmenwerk benötigte Metadaten, wie z.B. Information über Konnektoren und Konverter, werden im Resource Management gespeichert. Bisher wurden vom SIMPL-Rahmenwerk bzw. dessen Implementierung relationale Datenbanken und das Windows Dateisystem unterstützt.

Die bestehende Implementierung wurde im Rahmen dieser Arbeit analysiert und es wurde versucht Unterschiede, die sich beim Zugriff auf die verschiedenen Datenquellen oder Typen von Datenquellen ergeben, soweit wie möglich bzw. sinnvoll aufzulösen. Für relationale und JDBC-basierte Datenbanken gibt es nur noch einen Konnektor. Gemeinsamkeiten zwischen den verschiedenen Workflow Datenformaten wurden in einem neuen Basistyp zusammengefasst. Des Weiteren wurden neue Datenquellen in das SIMPL-Rahmenwerk integriert. In Zukunft kann auf verschiedene XML-Datenbanken sowie auf ein Sensornetz mittels TinyDB zugegriffen werden.

Für weitere Arbeiten bieten sich die folgenden Probleme bzw Aufgaben an:

- Die Unterschiede bei der QueryData Operation für relationale und JDBC-basierte Datenbanken wurden über ein *Workaround* aufgelöst (vgl. Abschnitt 6.2). Die benötigte Tabelle muss bereits vor der Ausführung der QueryData Operation vorhanden sein. Eventuell kann dieses Problem aber auch über zusätzliche Metadaten, die im Resource Management gespeichert werden, gelöst werden. So könnte in Zukunft ein entsprechendes Lebenszyklus-Management für Datencontainer entwickelt werden.
- Um TinyDB in das SIMPL-Rahmenwerk zu integrieren, wurde ein neuer Konnektor implementiert (vgl. Abschnitt 6.3). Dieser wartet bis eine Anfrage erstmalig evaluiert wurde, stoppt die Anfrage und gibt dann das Ergebnis zurück. Wünschenswert wäre ein Konnektor, der eine Workflow Aktivität bzw. einen BPEL Event Handler benachrichtigt, wenn neue Ergebnisse generiert wurden. Dazu müssten die SIMPL Core Operationen und die zugehörigen Workflow Aktivitäten asynchron aufgerufen bzw. ausgeführt werden. Des Weiteren konnte im Rahmen dieser Arbeit das Problem bei der Erstellung und Füllung der Buffer mit den IssueCommand bzw. QueryData

Operationen nicht gelöst werden. Vielleicht kann der erläuterte Ansatz in weiteren Arbeiten doch noch umgesetzt und in das SIMPL-Rahmenwerk integriert werden.

- Die im Rahmen dieser Studienarbeit integrierten XML-Datenbanken unterstützen alle eine andere API (vgl. Abschnitt 6.4). Aus diesem Grund mussten drei neue Konnektoren implementiert werden. Vielleicht wird sich in Zukunft die XML:DB API oder eine andere Technik als de-facto Standard für den Zugriff auf XML-Datenbanken durchsetzen. Dann könnte eventuell analog zu den relationalen Datenbanken ein *einziger* Konnektor die generischen Operationen des SIMPL Cores für alle oder viele XML-Datenbanken implementieren.
- Bisher ist es nicht möglich einzelne Container über einen logischen Namen zu referenzieren (vgl. Abschnitt 4.2). Um dies zu ermöglichen, müsste die Tabelle **Datacontainers** erzeugt sowie der SIMPL Core umgestellt werden. Des Weiteren müsste eine Möglichkeit geschaffen werden, Container über das Web Interface zu registrieren.
- Bisher wird nur das Windows Dateisystem vom SIMPL-Rahmenwerk unterstützt. Wünschenswert wäre eine Integration anderer Dateisysteme. Außerdem muss bei Dateisystemen eine besondere Problematik behandelt werden: Das SIMPL-Rahmenwerk basiert darauf, dass eine Datenquelle eine bestimmte Befehlssprache für das Datenmanagement, wie SQL, TinySQL oder XQuery, unterstützt. Befehle in der jeweiligen Befehlssprache werden zu der Datenquelle geschickt und dort ausgeführt. Bei Dateisystemen werden bis jetzt Shell-Kommandos als solche Befehle benutzt. Allerdings sind die entsprechenden Shell-Sprachen i.d.R. nicht speziell auf das Datenmanagement in Dateien ausgelegt und unterstützen daher nur sehr grundlegende Operationen, wie z.B. Operationen um Dateien zu kopieren oder zu löschen. In Zukunft könnte noch ein Ansatz entwickelt werden, um Dateisysteme um bestimmte DM Operationen zu erweitern, etwa um typische relationale Operationen für CSV-Dateien wie Selektion, Projektion oder Join.
- Der CSVDataConverter basiert auf der Annahme, dass die erste Zeile einer CSV-Datei Metadaten über die einzelnen Spalten enthält (vgl. Abschnitt 5.1.1). Vielleicht wäre es interessant CSV-Dateien zu unterstützten, die keine Metadaten über die einzelnen Spalten beinhalten.
- Für die verschiedenen Workflow Datenformate wurde ein neuer Basistyp eingeführt (vgl. Abschnitt 6.1). Dieser fasst die Gemeinsamkeiten zwischen den verschiedenen Formaten zusammen. In weiteren Arbeiten könnte untersucht werden, ob es sinnvoll ist, eine Hierachie von Datenformaten zu bilden. In dieser Hierachie könnte es dann z.B. ein einheitliches RelationalDataFormat geben, in dem die Elemente *rdbTableMetaData* und *csvTableMetaData* fehlen würden. Weitere Untertypen würden das einheitliche RelationalDataFormat dann wiederum konkreter spezifizieren.

A. Struktur von WS-BPEL

Listing A.1: Grundlegende Struktur von WS-BPEL [JE07]

```
<process name="NCName" targetNamespace="anyURI" queryLanguage="anyURI" ?
       expressionLanguage="anyURI"?
       suppressJoinFailure="yes|no"?
       exitOnStandardFault="yes|no"?
       xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable">

       <extensions>?
              <extension namespace="anyURI" mustUnderstand="yes|no" />+
       </extensions>

       <import namespace="anyURI"?
        location="anyURI"?
        importType="anyURI" />*

       <partnerLinks>?
              <!-- Note: At least one role must be specified. -->
              <partnerLink name="NCName"
              partnerLinkType="QName"
              myRole="NCName"?
              partnerRole="NCName"?
              initializePartnerRole="yes|no"?>+
              </partnerLink>
       </partnerLinks>

       <messageExchanges>?
              <messageExchange name="NCName" />+
       </messageExchanges>

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

       <correlationSets>?
              <correlationSet name="NCName" properties="QName-list" />+
       </correlationSets>

       <faultHandlers>?
              <!-- Note: There must be at least one faultHandler -->
              <catch faultName="QName"?
                     faultVariable="BPELVariableName"?
```

```
                faultMessageType="QName" | faultElement="QName" )>*
                activity
            </catch>
            <catchAll>?
                activity
            </catchAll>
        </faultHandlers>

        <eventHandlers>?
            <!-- Note: There must be at least one onEvent or onAlarm. -->
            <onEvent partnerLink="NCName"
                portType="QName"?
                operation="NCName"
                (messageType="QName" | element="QName" )?
                variable="BPELVariableName"?
                messageExchange="NCName"?>*
                <correlations>?
                    <correlation set="NCName" initiate="yes|join|no" ? />+
                </correlations>
                <fromParts>?
                    <fromPart part="NCName" toVariable="BPELVariableName" />+
                </fromParts>
                <scope...>...</scope>
            </onEvent>
            <onAlarm>*
                <!-- Note: There must be at least one expression. -->
                (
                <for expressionLanguage="anyURI" ?>duration-expr</for>
                |
                <until expressionLanguage="anyURI" ?>deadline-expr</until>
                )?
                <repeatEvery expressionLanguage="anyURI" ?>
                    duration-expr
                </repeatEvery>?
                <scope...>...</scope>
            </onAlarm>
        </eventHandlers>
        activity
</process>
```

B. Datenformate (XML Schemas)

Listing B.1: RelationalDataFormat [SIM]

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xsd:schema
    targetNamespace="http://org.simpl.core/plugins/dataconverter/dataformat/RelationalDataFormat"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://org.simpl.core/plugins/dataconverter/dataformat/RelationalDataFormat"
  xmlns:sdo="commonj.sdo"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <xsd:annotation>
    <xsd:documentation>Defines the SDO structure of relational data such as from databases or
        CSV files.</xsd:documentation>
  </xsd:annotation>
  <xsd:element name="RelationalDataFormat" type="tRelationalDataFormat"></xsd:element>
  <xsd:complexType name="tRelationalDataFormat">
    <xsd:sequence>
      <xsd:element name="dataFormatMetaData" type="tDataFormatMetaData"
        maxOccurs="1" minOccurs="0">
      </xsd:element>
      <xsd:element name="table" type="tTable" maxOccurs="unbounded"
        minOccurs="0">
      </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="dataFormat" type="xsd:string"></xsd:attribute>
  </xsd:complexType>
  <xsd:complexType name="tTable">
    <xsd:sequence>
      <xsd:element name="rdbTableMetaData" type="tRDBTableMetaData"
        maxOccurs="1" minOccurs="0">
      </xsd:element>
      <xsd:element name="csvTableMetaData" type="tCSVTableMetaData"
        maxOccurs="1" minOccurs="0"></xsd:element>
      <xsd:element name="row" type="tRow" maxOccurs="unbounded"
        minOccurs="0">
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="tColumn">
    <xsd:simpleContent>
      <xsd:extension base="xsd:string">
        <xsd:attribute name="name" type="xsd:string"></xsd:attribute>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
  <xsd:complexType name="tRow">
    <xsd:sequence>
```

```
      <xsd:element name="column" type="tColumn" maxOccurs="unbounded"
        minOccurs="1"></xsd:element>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="tCSVTableMetaData">
    <xsd:attribute name="filename" type="xsd:string"></xsd:attribute>
    <xsd:attribute name="separator" type="xsd:string"></xsd:attribute>
    <xsd:attribute name="quoteChar" type="xsd:string"></xsd:attribute>
    <xsd:attribute name="escapeChar" type="xsd:string"></xsd:attribute>
    <xsd:attribute name="strictQuotes" type="xsd:string"></xsd:attribute>
  </xsd:complexType>
  <xsd:complexType name="tRDBTableMetaData">
    <xsd:sequence>
      <xsd:element name="columnType" type="tColumnType" maxOccurs="unbounded"
        minOccurs="0"></xsd:element>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string"></xsd:attribute>
    <xsd:attribute name="schema" type="xsd:string"></xsd:attribute>
    <xsd:attribute name="catalog" type="xsd:string"></xsd:attribute>
  </xsd:complexType>
  <xsd:complexType name="tColumnType">
    <xsd:simpleContent>
      <xsd:extension base="xsd:string">
        <xsd:attribute name="isPrimaryKey" type="xsd:boolean" use="optional">
        </xsd:attribute>
        <xsd:attribute name="columnName" type="xsd:string" use="required"></xsd:attribute>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
  <xsd:complexType name="tDataFormatMetaData">
    <xsd:sequence>
      <xsd:element name="dataSource" type="xsd:string" maxOccurs="1"
        minOccurs="0"></xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Listing B.2: RandomFileDataFormat [SIM]

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xsd:schema
    targetNamespace="http://org.simpl.core/plugins/dataconverter/dataformat/RandomFileDataFormat"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://org.simpl.core/plugins/dataconverter/dataformat/RandomFileDataFormat"
  xmlns:sdo="commonj.sdo"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <xsd:annotation>
    <xsd:documentation>Defines the SDO structure of data from files.</xsd:documentation>
  </xsd:annotation>
  <xsd:element name="RandomFileDataFormat" type="tRandomFileDataFormat"></xsd:element>
  <xsd:complexType name="tRandomFileDataFormat">
    <xsd:sequence>
      <xsd:element name="file" type="tFile" maxOccurs="unbounded"
        minOccurs="0">
      </xsd:element>
```

```
    </xsd:sequence>
    <xsd:attribute name="dataFormat" type="xsd:string"></xsd:attribute>
    <xsd:attribute name="folder" type="xsd:string"></xsd:attribute>
  </xsd:complexType>
  <xsd:complexType name="tFile">
    <xsd:attribute name="name" type="xsd:string"></xsd:attribute>
    <xsd:attribute name="content" type="xsd:string"></xsd:attribute>
  </xsd:complexType>
</xsd:schema>
```

Literaturverzeichnis

[AAB+06] M. Adams, C. Andrei, R. Barack, H. Blohm, C. Boutard, S. Brodsky, F. Budinsky, S. Bünnig, M. Carey, B. Doughan, A. Grove, O. Halaseh, L. Harris, U. von Mersewsky, S. Moe, M. Nally, R. Preotiuc-Pietro, M. Rowley, E. Samson, J. Taylor, A. Thiefaine. SDO for Java Specification V2.1, 2006. URL http://www.osoa.org/download/attachments/36/Java-SDO-Spec-v2.1.0-FINAL.pdf?version=1. (Zitiert auf Seite 48)

[AV10] I. F. Akyildiz, M. C. Vuran. *Wireless Sensor Networks*. John Wiley & Sons, 2010. (Zitiert auf den Seiten 28 und 29)

[BBC+07] S. Bajaj, D. Box, D. Chappel, F. Curbera, G. Danielss, P. Hallam-Baker, M. Hondo, C. Kaler, D. Langworthy, A. Nadalin, N. Nagaratnam, H. Prafullchandra, C. von Riegen, D. Roth, J. Schlimmer, C. Sharp, J. Shewchuk, A. Vedamuthu, U. Yal"cinalp, D. Orchard. Web Services Policy 1.5 - Framework, 2007. URL http://www.w3.org/TR/ws-policy/. (Zitiert auf Seite 47)

[BBC+11] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, J. Siméon. XML Path Language (XPath) 2.0 (Second Edition), 2011. URL http://www.w3.org/TR/xpath20/. (Zitiert auf Seite 12)

[BCF+10] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, J. Siméon. XQuery 1.0: An XML Query Language (Second Edition), 2010. URL http://www.w3.org/TR/xquery/. (Zitiert auf Seite 12)

[BHM+04] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, D. Orchard. Web Services Architecture, 2004. URL http://www.w3.org/TR/ws-arch/. (Zitiert auf Seite 16)

[BKNT10] C. Baun, M. Kunze, J. Nimis, S. Tai. *Cloud Computing: Web-basierte dynamische IT-Services*. Springer, 2010. (Zitiert auf Seite 16)

[BPSM+08] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau. Extensible Markup Language (XML) 1.0 (Fifth Edition), 2008. URL http://www.w3.org/TR/xml/. (Zitiert auf Seite 9)

[CMRW07] R. Chinnici, J.-J. Moreau, A. Ryman, S. Weerawarana. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language, 2007. URL http://www.w3.org/TR/wsdl20/. (Zitiert auf den Seiten 6 und 17)

[Foua] Apache Software Foundation. Apache ODE. URL http://ode.apache.org/. (Zitiert auf den Seiten 20 und 41)

[Foub] The Eclipse Foundation. BPEL Designer Project. URL http://www.eclipse.org/bpel/. (Zitiert auf Seite 41)

[GHM+07] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, H. F. Nielsen, A. Karmarkar, Y. Lafon. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition), 2007. URL http://www.w3.org/TR/soap12-part1/. (Zitiert auf den Seiten 6 und 16)

[Gro] PostgreSQL Global Development Group. PostgreSQL. URL http://www.postgresql.org/. (Zitiert auf Seite 43)

[GSK+11] K. Görlach, M. Sonntag, D. Karastoyanova, F. Leymann, M. Reiter. *Conventional Workflow Technology for Scientific Simulation*. in: Guide to E-Science, Springer, 2011. (Zitiert auf den Seiten 5, 20 und 32)

[JE07] D. Jordan, J. Evdemon. Web Services Business Process Execution Language Version 2.0, 2007. URL http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf. (Zitiert auf den Seiten 6, 20, 21, 41 und 75)

[Kai] Technische Universität Kaiserslautern. XML Transaction Coordinator (XTC). URL http://wwwlgis.informatik.uni-kl.de/cms/dbis/projects/xtc/. (Zitiert auf Seite 63)

[KE11] A. Kemper, A. Eikler. *Datenbanksysteme: Eine Einführung*. Oldenbourg Wissenschaftsverlag, 2011. (Zitiert auf den Seiten 9, 12, 14, 23, 24, 26 und 28)

[LR99] F. Leymann, D. Roller. *Production Workflow: Concepts and Techniques*. Prentice Hall International, 1999. (Zitiert auf den Seiten 5, 7, 18, 19 und 32)

[Mel10] I. Melzer. *Service-orientierte Architekturen mit Web Services: Konzepte - Standards - Praxis*. Spektrum Akademischer Verlag, 2010. (Zitiert auf den Seiten 5, 6, 14, 15, 16, 17, 19, 21 und 22)

[MHH03] S. Madden, J. Hellerstein, W. Hong. TinyDB: In-Network Query Processing in TinyOS, 2003. URL http://telegraph.cs.berkeley.edu/tinydb/tinydb.pdf. (Zitiert auf den Seiten 5, 29 und 30)

[Ora] Oracle. Java Database Connectivity (JDBC). URL http://download.oracle.com/javase/6/docs/technotes/guides/jdbc/. (Zitiert auf Seite 56)

[RRS+11] P. Reimann, M. Reiter, H. Schwarz, D. Karastoyanova, F. Leymann. SIMPL - A Framework for Accessing External Data in Simulation Workflows. *Gesellschaft für Informatik (ed.): Datenbanksysteme für Business, Technologie und Web*, pp. 534–553, 2011. (Zitiert auf den Seiten 5, 7, 31, 33, 36, 37, 38 und 39)

[Sch03] H. Schöning. *XML und Datenbanken: Konzepte und Systeme*. Carl Hanser Verlag, 2003. (Zitiert auf den Seiten 6, 9, 11, 25, 26 und 28)

[Seb10] T. J. Sebestyen. *XML: Einstieg für Anspruchsvolle*. Addison-Wesley, 2010. (Zitiert auf den Seiten 9 und 12)

[SIM] SIMPL. SIMPL Framework (Source Code). URL http://code.google.com/p/simpl09/. (Zitiert auf den Seiten 6, 41, 47, 48, 56, 57, 59, 70, 77 und 78)

[Sys] Berkeley WEBS: Wireless Embedded Systems. TinyOS. URL http://webs.cs.berkeley.edu/tos/dist-1.1.0/tinyos/windows/. (Zitiert auf den Seiten 6 und 62)

[XI] The XML:DB Initiative. Application Programming Interface for XML Databases. URL http://xmldb-org.sourceforge.net/xapi/. (Zitiert auf Seite 62)

[ZJ09] J. Zheng, A. Jamalipour. *Wireless Sensor Networks: A Networking Perspective*. John Wiley & Sons, 2009. (Zitiert auf Seite 28)

Alle URLs wurden zuletzt am 07.12.2011 geprüft.

Abkürzungsverzeichnis

BPEL	Business Process Execution Language
BPEL-DM	Business Process Execution Language extension for Data Management
CSV	Comma-Separated Values
DBS	Datenbanksystem
DBVS	Datenbankverwaltungssystem
DTD	Document Type Definition
HTML	Hypertext Markup Language
OASIS	Organization for the Advancement of Structured Information Standards
ODE	Orchestration Director Engine
SIMPL	SimTech - Information Management, Processes and Languages
SOA	Serviceorientierte Architektur
SQL	Structured Query Language
sWfMS	Scientific Workflow Management System
UDDI	Universal Description, Discovery and Integration
W3C	World Wide Web Consortium
WfMC	Workflow Management Coalition
WS	Web Service
WS-BPEL	Web Services Business Process Execution Language
WSDL	Web Services Description Language
XML	Extensible Markup Language
XPath	XML Path Language
XQuery	XML Query Language
XSD	XML Schema

Erklärung

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

(Henrik Andreas Pietranek)