ulm university universität
uulm

Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften, Informatik und Psychologie
Institut für Datenbanken und Informationssysteme

Konzeption und Realisierung eines Frameworks zur Abbildung von Geschäftsprozessen auf einer Graphendatenbank

Bachelorarbeit an der Universität Ulm

Vorgelegt von:
Kevin Bee
kevin.bee@uni-ulm.de

Gutachter:
Prof. Dr. Manfred Reichert

Betreuer:
Klaus Kammerer

2016

Fassung 30. Mai 2016

Satz: PDF-LaTeX 2ε

Kurzfassung

Wachsender Wettbewerb, Globalisierung und immer kürzere Produktentwicklungszyklen bringen Unternehmen an ihre Grenzen. Business Process Management ist ein systematischer Ansatz, um diesen gestiegenen Anforderungen gerecht zu werden. Hierzu werden Geschäftsprozesse erfasst und modelliert. Innerhalb eines Unternehmens können hierbei sehr viele Prozessmodelle entstehen, die verwaltet und stetig angepasst werden müssen. Eine Anpassung kann unter Umständen mehrere Prozessmodelle betreffen und sehr zeit- und arbeitsintensiv werden.

Ziel dieser Arbeit ist die Entwicklung des PQLParser Frameworks zur Unterstützung von Änderungsoperationen auf einer Menge von Prozessmodellen. Dazu können mehrere Prozessmodelle abgerufen und in eine gemeinsame Graphnedatenbank geladen werden. Mit Hilfe der Process Query Language ist es möglich Such- und Änderungsanfragen zu formulieren und diese auf der Menge der Prozessmodelle auszuführen. Suchanfragen können hierbei auf Basis von hochoptimierten Anfragen auf die Graphendatenbank effizient durchgeführt werden. Änderungsanfragen werden auf Korrektheit überprüft, damit bei der Durchführung von Änderungen keine syntaktischen Fehler auftreten. Das PQLParser Framework ermöglicht die Verwaltung von Suchanfragen und Änderungen auf einer Menge von Prozessmodellen.

Inhaltsverzeichnis

1 Einleitung

Business Process Management hilft Unternehmen Geschäftsprozesse zu erfassen, zu dokumentieren, auszuführen und zu verbessern. Sofern alle Geschäftsprozesse eines Unternehmens erfasst und verwaltet werden sollen, kann schnell eine eine Vielzahl an Prozessmodellen entstehen, die wiederum einen erhöhten Wartungs- und Pflegeaufwand mit sich bringen. Wenn ein Unternehmen leistungsfähig bleiben will, ist es notwendig, diese Geschäftsprozesse stetig zu verbessern und anzupassen. Infolgedessen müssen auch die zugehörigen Prozessmodelle verändert werden werden.

1.1 Problemstellung

Geschäftsprozesse sind einem stetigen Wandel unterlegen. Aus diesem Grund müssen regelmäßig Änderungen an bestehenden Prozessmodellen durchgeführt werden. Da

eine Änderung oftmals mehr als ein Prozessmodell betrifft und der Überblick bei der Vielzahl an bestehenden Prozessmodellen schnell verloren gehen kann, muss sichergestellt werden, dass eine Änderung bei allen betroffenen Prozessmodellen korrekt und konsistent eingepflegt wird. Dieser Vorgang kann sehr zeit- und kostenaufwendig sein, denn eine Änderung muss in jedem zugehörigen Prozessmodell durchgeführt werden. Dabei muss zusätzlich auf Korrektheit der Änderungen geachtet werden, damit die Struktur und Semantik eines Modells nicht verletzt wird und dessen Aussage erhalten bleibt.

1.2 Zielsetzung

Das in dieser Arbeit vorgestellte Framework soll Funktionen bereitstellen, mit deren Hilfe eine große Menge an Prozessmodellen durchsucht, abgerufen und verändert werden können. Basis des Frameworks soll die Process Query Language sein, mit der sich Anfragen und Änderungen auf einer Menge von Prozessmodellen textuell beschreiben lassen. Zur Umsetzung dieser Änderungen sollen vordefinierte High-Level-Operationen zur Verfügung gestellt werden, die bei Ausführung die Struktur eines Prozessmodells nicht verletzen. Als Speicherbasis soll das Framework eine Graphendatenbank bereitstellen, auf der sich Prozessmodelle strukturell korrekt speichern und verändern lassen. Durch die Trennung der Prozessmodelle von den restlichen in einem Prozessmanagementsystem soll sichergestellt werden, dass keine unerwünschten Änderungen auf nicht abgerufenen Prozessmodellen ausgeführt werden. Zusätzlich soll die Suche und Änderung von abgerufenen Modellen optimiert werden, sodass diese schnell durchgeführt werden. Das Framework soll somit die Verwaltung von Änderungen an mehreren Prozessmodellen oder mehreren Varianten von einem Prozessmodell ermöglichen.

1.3 Struktur der Arbeit

In Kapitel 2 werden die Grundlagen zu Business Process Management, der Extensible Markup Language und Domänen-spezifischen Abfragesprachen eingeführt. Kapitel 3

stellt das PQLParser Framework vor. Dabei wird zunächst die Architektur beschrieben und anschließend wichtige Implementierungsaspekte betrachtet. Verwandte Themen werden in Kapitel 4 aufgezählt und besprochen. Eine Zusammenfassung der Arbeit und ein Ausblick wird in Kapitel 5 gegeben.

2

Grundlagen

Dieses Kapitel führt grundlegende Konzepte ein, die zum weiteren Verständnis notwendig sind. Zunächst werden Grundlagen zu *Business Process Management (BPM)* eingeführt und wichtige Begriffe erläutert. Danach wird in Kapitel 2.1.2 die Modellierungssprache *Business Process Model and Notation 2.0 (BPMN)* vorgestellt. Kapitel 2.1.3 behandelt *Change Patterns* [1], Kapitel 2.2 erklärt zuerst Konzepte der *Extensible Markup Language (XML)* und stellt anschließend die Repräsentation eines Geschäftsprozess im XML-Format vor. Abschließend werden in Kapitel 2.3 *domänen-spezifische Abfragesprachen (DSLs)* vorgestellt.

2.1 Business Process Management

BPM ist nicht nur eine Technologie zur Verbesserung eines Unternehmens, sondern ein systematischer Ansatz zur Erfassung, Dokumentation, Erstellung, Verbesserung, Ausführung und Überwachung von Geschäftsprozessen [2, 3]. Mithilfe von BPM sind Unternehmen in der Lage, Geschäftsprozesse zu kontrollieren und zu überprüfen. Somit kann ein gleichbleibend gutes Resultat der Arbeit sichergestellt werden. Zusätzlich soll sichergestellt werden, dass Unternehmen ihre Ziele erreichen. Folglich trägt BPM zur Verbesserung von Geschäftsprozessen im Allgemeinen bei, um damit einen Mehrwert für das Unternehmen zu schaffen. Als übergeordnete Ziele können z.B. Kostenreduzierungen oder Verbesserungen der Durchlaufzeiten in einer Produktion angesehen werden [3, 4, 5].

Ein *Geschäftsprozess* besteht aus mehreren, zeitlich und kausal in Beziehung stehenden *Aktivitäten*, die zur Erreichung eines vordefinierten Ziels ausgeführt werden müssen. Diese Aktivitäten können sowohl manuell (durch eine Person), als auch automatisch (z.B. durch ein Informationssystem) ausgeführt werden.

Die elektronische Unterstützung von Geschäftsprozessen spielt eine immer größer werdende Rolle in Unternehmen. Diese kann durch sogenannte *Process-Aware Information Systems (PAIS)* umgesetzt werden [6, 7].

BPM fokussiert nicht auf die Änderung von einzelnen Aktivitäten sondern auf den kompletten Geschäftsprozess [4]. Die Geschäftsprozesse, die hierzu erfasst werden, sind ein Abbild der *Realweltprozesse* und stellen somit ein *Modell der realen Welt (Prozessmodell)* dar, bei dem auch triviale Schritte zu einer komplexen Aktivität zusammengefasst werden können. In Abbildung 2.1 ist beispielsweise ein Bestellprozess gezeigt, bei dem eine Bestellung aufgenommen, Produkte verschickt und die zugehörige Rechnung einem Kunden zugestellt wird. Nachdem die Zahlung eingegangen ist, kann die Bestellung archiviert werden [6].

Die in Abbildung 2.1 verwendete Modellierungssprache ist BPMN 2.0, die in Kapitel 2.1.2 näher erläutert wird. Die Schritte zur Unterstützung von BPM und der Entwicklung

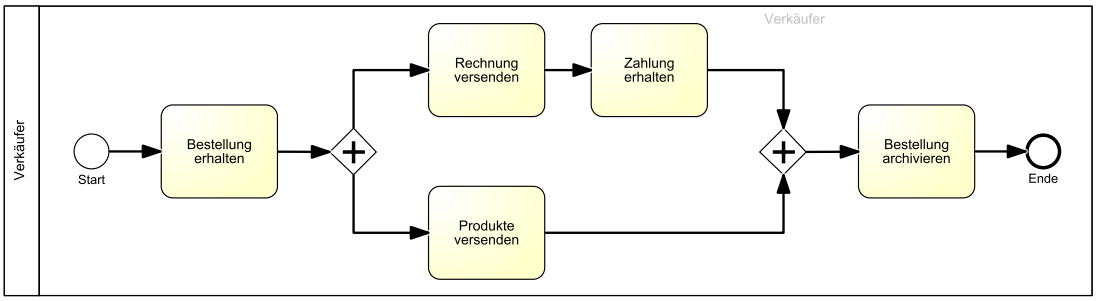

Abbildung 2.1: Bestellprozess (Quelle: [6])

eines Geschäftsprozess können anhand des *BPM Lebenszyklus* zusammengefasst und erklärt werden.

2.1.1 BPM Lebenszyklus

Geschäftsprozesse unterliegen stetigen Änderungen, da sich ein Unternehmen kontinuierlich an äußere Gegebenheiten, wie z.B. Veränderungen am Markt, anpassen muss. Der BPM Lebenszyklus beschreibt Phasen, die Geschäftsprozesse zur Entwicklung, Anpassung und Verbesserung durchlaufen sollten. Abbildung 2.2 zeigt das Schema des BPM Lebenszyklus mit sechs Teilschritten: *Prozessidentifikation*, *Prozesserhebung*, *Prozessanalyse*, *Prozessneugestaltung*, *Prozessimplementierung*, *Prozessüberwachung und -kontrolle* [4].

Bei der *Prozessidentifikation* werden Realweltprozesse identifiziert und zu einer *Prozessarchitektur* zusammengefasst, aus der Zusammenhänge zwischen allen erfassten Realweltprozessen entnommen werden können. Hierfür müssen zunächst die Unternehmensstrategie und die Ziele des jeweiligen Unternehmens analysiert werden. Sobald diese Realweltprozesse identifiziert wurden, muss entschieden werden, ob neue Prozessmodelle entwickelt oder bestehende überarbeitet werden sollen [4, 8].

In der Phase der *Prozesserhebung* werden Realweltprozesse dokumentiert und als Prozessmodelle interpretiert [4]. Hier werden gleichzeitig die einzusetzende Modellierungssprache und zu beachtenden Konventionen zur Modellierung festgelegt. Prozessmodelle,

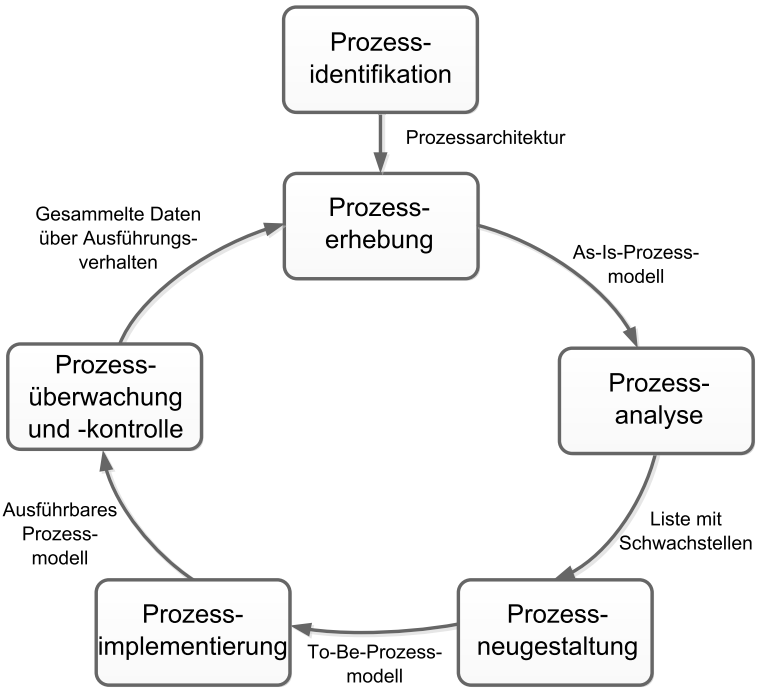

Abbildung 2.2: BPM Lebenszyklus (Quelle: nach [4])

die hieraus entstehen, reflektieren das Wissen der beteiligten Akteure und werden als *As-Is-Prozesse* bezeichnet [4].

In der *Prozessanalysephase* werden die As-Is-Prozesse aus der Prozesserfassung untersucht. Hierbei werden Schwachstellen gesucht, wie z.B. Aktivitäten, die die Durchlaufzeiten von Aufträgen unnötig verlängern oder die unnötig bzw. in einer falschen Reihenfolge ausgeführt werden [4].

Während der *Prozessneugestaltung* werden identifizierte Änderungen der Prozessanalysephase auf die jeweiligen Prozessmodelle übertragen. Ziel der Prozessneugestaltung ist ein so genanntes *To-Be-Prozessmodell*, welches als Grundlage für die folgende Phase dient [4, 8].

Bei der *Prozessimplementierung* werden To-Be-Prozesse der vorherigen Phase umgesetzt, indem beteiligte Akteure geschult oder diese Modelle mithilfe von Prozessauto-

mation durch das Entwickeln und Einsetzen von Softwaresystemen umgesetzt werden [4].

In der letzten Phase der *Prozessüberwachung und -kontrolle* werden Daten über die Ausführung des neuen oder überarbeiteten Prozesses gesammelt. Mithilfe dieser Daten kann die Effizienz und Effektivität eines Prozesses gemessen und dargestellt werden. Die somit erlangten Erkenntnisse können in einen weiteren Durchlauf des BPM Lebenszyklus als Basis für weitere Prozessverbesserungen dienen [8].

2.1.2 Business Process Model and Notation

Damit Geschäftsprozesse modelliert werden können, benötigt man eine einheitliche Modellierungssprache. Grafische Modelle sind für die Modellierung von Geschäftsprozessen gut geeignet. BPMN wurde speziell entwickelt, um Geschäftsprozesse als grafische Modelle darstellen zu können. Mit der Übernahme der Modellierungssprache durch die *Object Management Group (OMG)* im Jahre 2005 hat sich BPMN als Standardsprache in der Prozessmodellierung etabliert. Seit Version 2.0 können BPMN-Modelle auch direkt in einem PAIS ausgeführt werden [3, 7]. Abbildung 2.3 stellt die BPMN-Kernelemente dar.

Im Folgenden werden die Kernelemente von BPMN anhand eines einfachen Beispiels erklärt (siehe Abbildung 2.4). Dabei kann lediglich eine kleine Einführung in das Thema BPMN gegeben werden, um die Grundlagen zu betrachten, die für die später vorgestellte technische Implementierung benötigt werden. Eine weiterführende Beschreibung zu BPMN findet sich in [3]. Die Umwandlung eines BPMN-Modells in ein Graphenmodell eines Datenbanksystems wird in Kapitel 3.4.4 näher erläutert.

Abbildung 2.4 zeigt folgende Elemente: *Aktivitäten*, *Ereignisse*, *Gateways*, *Sequenzflüsse*, *Nachrichtenflüsse*, *Datenobjekte* und *Pools*. Aktivitäten, Ereignisse und Gateways können als *Fluss-Objekte* zusammengefasst werden. *Verbindende Objekte* umfassen Sequenzflüsse, Nachrichtenflüsse und *Assoziationen*. Unter Artefakte fallen Datenobjekte, *Gruppierungen* und *Anmerkungen*. Pools und *Lanes* zählen zur Gruppe der *Teilnehmer* [3].

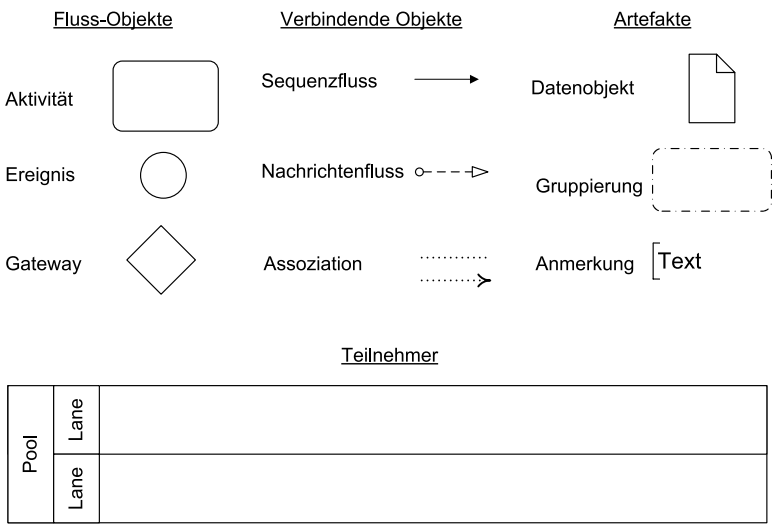

Abbildung 2.3: Kernelemente von BPMN (Quelle: [3])

Eine Aktivität kann von Menschen oder maschinell ausgeführt werden und beschreibt eine *atomare Aufgabe (Task)* oder einen *Subprozess*. Ein Task kann verschiedene Typen annehmen. Wichtige Typen von Tasks sind: *Task*, *Manual Task*, *User Task*, *Service Task* und *Script Task*, aber im Rahmen dieser Arbeit wird lediglich die normale Task betrachtet. Bei der normalen Task wird eine Aufgabe beschrieben, bei der nicht näher spezifiziert ist, wie und von wem sie ausgeführt wird [3].

Ereignisse beeinflussen Geschäftsprozesse dahingehend, dass sie Prozesse starten (*Startereignisse*), beenden (*Endereignisse*) oder eine Statusänderung während der Ausführung signalisieren oder behandeln können (*Zwischenereignisse*). Es gibt viele verschiedene Start-, Zwischen- und Endereigniss-Typen. Ereignisse können unter anderem *Nachrichtenereignisse* oder *Zeitereignisse* sein. Mithilfe von Nachrichtenereignissen kann das Senden oder Empfangen einer Nachricht zwischen verschiedenen Akteuren modelliert werden, beispielsweise das Erhalten von Login-Daten (siehe Abbildung 2.4). Durch Zeitereignisse können zeitliche Aspekte ausgedrückt werden, wie z.B. das Starten eines Prozesses an jedem ersten eines im Monat [3].

Gateways werden verwendet, um das Aufteilen und Zusammenführen des Kontrollflusses zu realisieren [9]. Dafür stehen verschiedene Typen zur Verfügung: *XOR-Gateway*, *AND-*

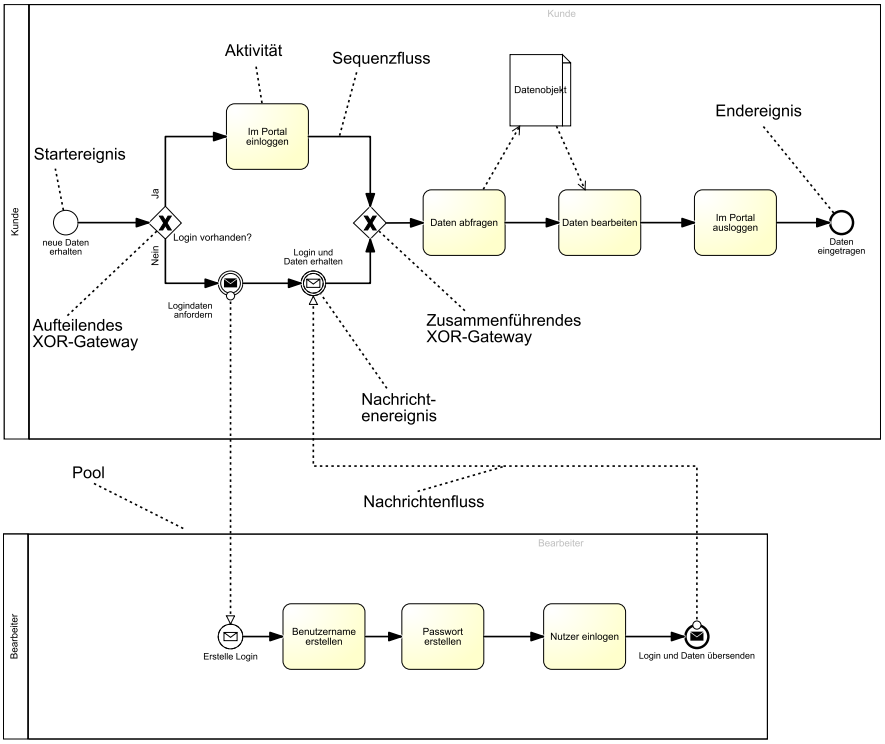

Abbildung 2.4: Beispielprozess: Bearbeitung von Daten mit vorgeschaltetem Login

Gateway, *OR-Gateway*, *Event-basiertes-Gateway*. Mithilfe des XOR-Gateway können Entscheidungen zwischen verschiedenen Pfaden simuliert werden. Das AND-Gateway wird dann benötigt, wenn verschiedene Aktivitäten parallel ablaufen sollen [3].

Der Sequenzfluss ist das Bindeelement, mit dem Aktivitäten, Ereignisse und Gateways miteinander verbunden werden. Damit kann der zeitliche Ablauf und somit die Reihenfolge der Elemente bestimmt werden [3]. Mit Nachrichtenflüssen wird die Kommunikation zwischen zwei verschiedenen Akteuren modelliert. Ein Nachrichtenfluss ist folglich ein Austausch von Informationen [3]. Mit Datenobjekten können Informationen oder Dokumente modelliert werden, die an einer Prozessausführung beteiligt sind [3].

Mithilfe von Pools können Aktivitäten von verschiedenen Akteuren zusammengefasst und voneinander getrennt werden. Verschiedene Pools können ausschließlich über Nachrichtenflüsse miteinander kommunizieren [3]. Mit einer Lane können Pools weiter

unterteilt werden. Eine Lane bewirkt somit eine noch genauere Zuordnung von Fluss-Objekten [3].

2.1.3 Change Patterns

In diesem Kapital wird auf die Durchführung von Änderungen auf Geschäftsprozessen eingegangen. Die sogenannten *Change Patterns* beschreiben, wie Kontrollflussänderungen in Geschäftsprozessen korrekt durchgeführt werden können. Die Change Patterns können in zwei Kategorien aufgeteilt werden: *Adaptation Patterns* und *Patterns für Änderungen in vordefinierten Regionen*. Die Patterns für Änderungen in vordefinierten Regionen lassen Änderungen an laufenden *Prozessinstanzen* zu. Eine Prozessinstanz ist dabei ein in der Ausführung befindendes Prozessmodell. Durch die Adaptation Patterns können strukturelle Änderungen von Prozessmodellen durchgeführt werden [1].

Die Anwendung der Adaptation Patterns kann als Transformation von einem Prozessmodell in ein anderes angesehen werden. Um dies zu realisieren, gibt es zwei Möglichkeiten. Eine Möglichkeit besteht darin, mehrere einzelne *elementare Änderungsoperationen* aneinanderzureihen und separat auszuführen, um eine Änderung, wie z.B. `Lösche Aktivität`, durchzuführen. Elementare Änderungsoperation sind primitive Operationen, wie `füge Knoten hinzu`, `lösche Knoten`, `füge Kante hinzu`, `lösche Kante`, `verschiebe Kante`. Bei der Umsetzung einer elementaren Änderungsoperation ist es schwierig Vor- und Nachbedingungen festzulegen, welche die strukturelle Korrektheit eines Prozessmodells sicherstellen (z.B. Vermeidung von Deadlocks) [10]. Aus diesem Grund können zur Umsetzung einer Änderungen sogenannte *High-Level-Operationen* definiert werden. Eine High-Level-Operation fasst die Ausführung mehrerer elementarer Änderungsoperationen zu einem komplexen Schritt zusammen. Durch die Abstraktion können Vor- und Nachbedingungen definiert werden, die gelten müssen, um so die strukturelle Korrektheit eines Prozessmodells zu gewährleisten. Ein Adaptation Pattern besteht aus genau einer High-Level-Operation [1]. Drei dieser Pattern (*Insert*, *Delete* und *Move*) werden im nachfolgend vorgestellten Framework beispielhaft implementiert. Die Implementierung wird in Kapitel 3.4.2 erläutert. Nachfolgend wird die Funktionsweise und der Aufbau dieser drei

Pattern kurz beschrieben. Eine vollständige Beschreibung der Funktionsweise und der Implementierung aller Patterns kann aus [1] entnommen werden.

Insertion Pattern

Mithilfe der High-Level-Operation *Insert* kann eine neue Aktivität in ein Prozessmodell eingefügt werden. Das Einfügen einer neuen Aktivität kann *seriell*, *parallel* oder *konditional* erfolgen. Seriell bezeichnet das Einfügen einer Aktivität zwischen zwei direkt aufeinander folgenden Aktivitäten. Bei parallelem bzw. konditionalem Einfügen von Aktivitäten werden diese als Zweig, mit einem Split- und Join-Gateway, dem Prozessmodell hinzugefügt. Die folgende Abfolge von elementaren Änderungsoperationen werden ausgeführt: `füge Knoten hinzu` → `verschiebe Kante` → `füge Kante hinzu` [1].

Deletion Pattern

Mit dem Deletion Pattern wird beschrieben, wie Aktivitäten aus einem Prozessmodell gelöscht werden können. Hierfür wird die folgende Abfolge von elementaren Änderungsoperationen ausgeführt: `lösche Kante` → `lösche Knoten` → `verschiebe Kante` [1].

Move Pattern

Das Move Pattern erlaubt die Verschiebung von bestehenden Aktivitäten. Beim Verschieben können wiederum drei Fälle unterschieden werden. Das Verschieben seriell, parallel und konditional kann auf die selbe Art wie beim Insert Pattern umgesetzt werden. Hierfür wird die folgende Abfolge von primitiven Operationen ausgeführt: `lösche Knoten` → `lösche Kante` → `füge Knoten hinzu` → `verschiebe Kante` → `füge Kante hinzu`. Eine weitere Möglichkeit das Move Pattern zu implementieren ist: `Wende Insert-Operation an` → `wende Delete-Operation an` [1].

2.2 Extensible Markup Language

Die *Extensible Markup Language (XML)* ist ein text-basiertes Format zum Austausch von Informationen zwischen Informationssystemen. Informationen können hierbei jegliche Art von Objekten sein; beispielsweise können Bücher durch die Angabe von Titel, Autor und ISBN dargestellt werden. Nachfolgend wird zunächst Generelles zu XML erläutert, danach wird auf das XML-Format von Geschäftsprozessen eingegangen.

XML-Dokumente sind aus *Entitäten* aufgebaut. Entitäten sind Speichereinheiten, die durch ihren Namen identifiziert werden können und einen Inhalt besitzen. Sie können entweder weitere Entitäten oder Text enthalten. Start eines XML-Dokuments ist gekennzeichnet mit der *XML-Deklaration*. In dieser wird die Version und weitere Eigenschaften bestimmt, wie z.B. die Kodierung des Textes des XML-Dokuments. Entitäten müssen jeweils mit einem *Start-Tag* geöffnet und mit einem *End-Tag* geschlossen werden. Eine Ausnahme hiervon bildet das leere Element, das mit einem Mix aus Start- und End-Tag geschrieben wird (z.B. <buch/> stellt eine leere Entität eines Buches dar). Des Weiteren können Entitäten mit Attributen versehen werden, die in den Start-Tag mit hineingeschrieben werden. Attribute sind Schlüssel-Wert-Paare, die als weitere Informationen zu Elementen angesehen werden können [11]. Listing 2.1 zeigt das Beispiel einer XML-Datei mit Informationen zu einem Buch.

```
<!-- Start des Dokuments mit der XML-Deklaration -->
<?xml version="1.0"?>
<!-- Start-Tag mit Attribut: ISBN-Nummer des Buches -->
<buch ISBN='3642304095'>
 <titel>
  <!-- Information als Text innerhalb einer Entitaet gespeichert -->
  Enabling Flexibility in Process-Aware Information Systems: Challenges, Methods,
      Technologies
 </titel>
 <autor>
  Manfred Reichert
 </autor>
<!-- End-Tag -->
</buch>
```

Listing 2.1: Beispiel einer XML-Datei

XML-Dokumente müssen *wohlgeformt* sein, d.h. sie müssen definierten syntaktischen Regeln folgen.

Wohlgeformtheit von XML-Dokumenten

- Das Dokument enthält ein oder mehrere Element(e)
- Es gibt genau ein Wurzelelement, das innerhalb von keinem anderen Element auftaucht
- Start- und End-Tag müssen blockstrukturiert sein
- Jede Entität für sich genommen ist wohlgeformt

Nachfolgend wird dieses Format vorgestellt und erläutert.

2.2.1 XML-Repräsentation von Geschäftsprozessen

In diesem Kapitel wird die Abbildung eines Geschäftsprozess auf ein XML-Format beschrieben.

Abbildung 2.5 zeigt ein einfaches Prozessmodell eines Shop-Benachrichtigungsprozesses, dessen Struktur in ein XML-Format umgewandelt werden soll. Dieses Prozessmodell beinhaltet zwei Pools: `Shop` und `Kunde`. Start des Prozesses ist die Ankunft neuer Waren im Shop. Anschließend wird der Kunde über den Erhalt der neuen Ware benachrichtigt. Mit der Nachricht über neue Ware im Shop wird die Kundenseite des Prozesses gestartet und der Kunde geht einkaufen. Anschließend ist der Prozess beendet. Um im Wesentlichen zu verstehen, wie Geschäftsprozesse in XML-Dateien abgebildet sind, werden in dem folgenden Beispiel die grafischen Aspekte der BPMN-XML-Repräsentation außer Acht gelassen.

Start des Dokuments bildet die Deklaration der XML-Datei mit Version und Kodierung. Anschließend wird ein Geschäftsprozess mit den `<definitions/>` begonnen. Hier werden Spezifikationen zu BPMN [12] und Informationen zum Prozesseditor gegeben (siehe Listing 2.2 Zeilen 1-4).

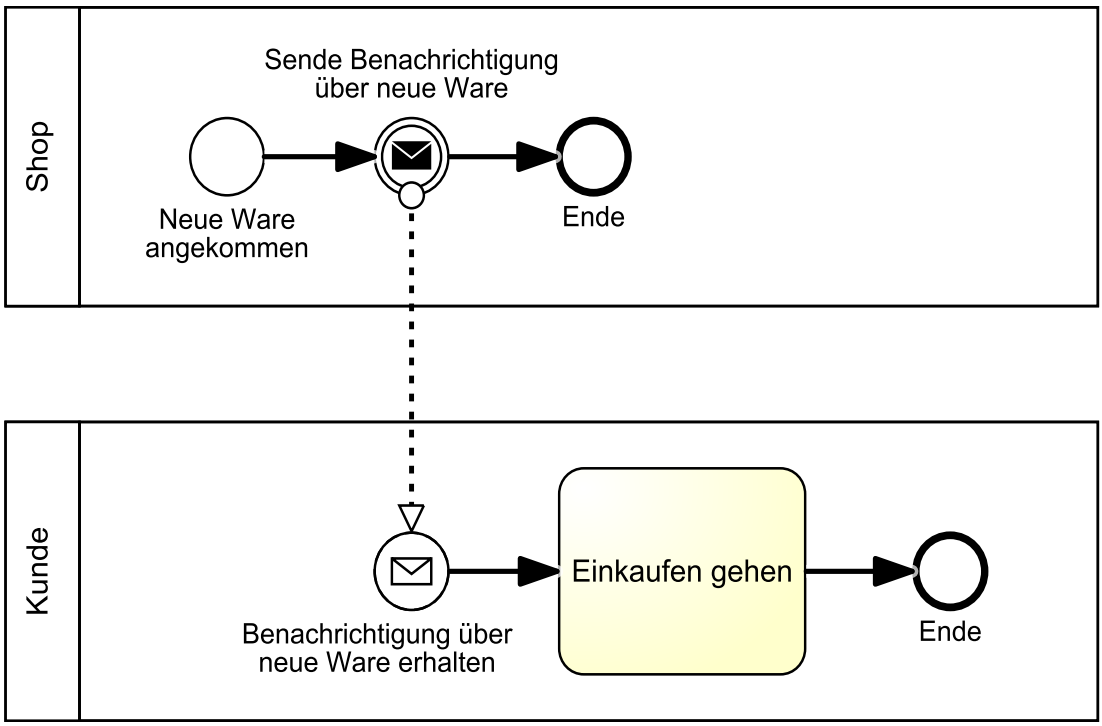

Abbildung 2.5: Shop-Benachrichtigungsprozess

Anschließend folgt die Definition der Elemente des Prozessmodells. Zuerst wird eine `<collaboration/>` definiert, in der alle Pools des Prozessmodells aufgeführt werden. Wenn es mehrere Pools in einem Prozessmodell gibt, werden diese mit weiteren `<participant/>` -Elementen beschrieben und mit angehängt. Am Ende des `<collaboration/>`-Elementes werden außerdem die Nachrichtenflüsse aufgelistet. Ein `<messageFlow/>`-Element besitzt eine eigene ID und die IDs der Quelle und des Ziels (siehe Listing 2.2 Zeilen 5-14).

Für jeden in der `<collaboration/>` vorkommenden Pool gibt es ein dazugehöriges `<process/>`-Element, in dem der Teil des Prozessmodells definiert ist, der in diesem Pool stattfindet (siehe Listing 2.2 Zeilen 16-46 für den Shop-Pool und Zeilen 48-73 für den Kunden-Pool). Dabei wird im `<participant/>`-Element mit dem Attribut `processRef` das `id`-Attribut des jeweiligen Pool referenziert (siehe Listing 2.2 Zeilen 7-11).

Ein Pool kann mehrere Lanes enthalten. Daher müssen alle enthaltenen Elemente jeder Lane definiert werden. Dafür existiert ein `<laneSet/>`-Element, in dem die ID jedes

Elements in der Lane dokumentiert ist. Die Zugehörigkeit eines Elements zu einer Lane ist in einem `<flowNodeRef/>`-Element gespeichert. In Listing 2.2, Zeile 21-24 wird beispielhaft eine Lane gezeigt, in der die Elemente des Shop-Pool gespeichert sind.

Die Darstellung von Fluss-Objekten und verbindenden Objekten wird nachfolgend erläutert. Der Aufbau von Aktivitäten, Gateways und Ereignissen ist sehr ähnlich, sie unterscheiden sich unter anderem in ihren Attributen. In Listing 2.2, Zeile 56-60 wird eine Aktivität dargestellt. Die Aktivität hat vier Attribute, darunter ist eine `ID` und den `Namen` der Aktivität. Des Weiteren sind in einem Element, die ankommenden und ausgehenden Sequenzflüsse gespeichert (siehe Listing 2.2, Zeile 58 und 59). Listing 2.2, Zeile 58 und 59 zeigt in dem Fall der Aktivität eine ankommende und zwei ausgehende Sequenzflüsse.

Die Sequenzflüsse werden nicht nur in den Elementen selbst gespeichert, sondern am Ende eines `<process/>`-Elements nochmals aufgelistet. Hier wird zunächst eine eindeutige ID vergeben und zusätzlich Quelle und Ziel des Sequenzflusses gespeichert (siehe Listing 2.2, Zeile 44 und 45).

```
<!-- Start des Dokuments -->
<?xml version="1.0" encoding="UTF-8"?>
<!-- Start des Geschäftsprozesses mit <definitions> als Wurzelelement -->
<definitions id="sid-a66dbab6-05e8-49ab-937d-d7d8ee2e54b3" targetNamespace="http://www.signavio.com"
    typeLanguage="http://www.w3.org/2001/XMLSchema" xsi:schemaLocation="http://www.omg.org/spec/BPMN
    /20100524/MODEL http://www.omg.org/spec/BPMN/2.0/20100501/BPMN20.xsd">
  <!-- <collaboration> startet die Auflistung der Pools eines Prozessmodells -->
  <collaboration id="sid-3d52c1c2-7cee-4472-a180-184c6f47fec8">
    <!-- mit <participant>-Elementen werden beteiligte Pools eines Prozessmodells
        beschrieben -->
    <participant id="sid-9D5D891E-BC60-417F-8E3E-0EE0075C6764" name="Shop" processRef="sid-
        FEED7AA8-F7B5-4347-8461-A278C859A25F">
    </participant>
    <participant id="sid-BD9FD514-A7EB-4C9A-A4A5-A0C4D7C85896" name="Kunde" processRef="sid-
        E80336E7-4377-4900-8428-A6B85CCCB4CC">
    </participant>
    <!-- Nachrichtenflüsse werden an das <collaboration/>-Element angehängt -->
    <messageFlow id="sid-C250393F-0C6B-43B9-9931-30AF64BD98E2" sourceRef="sid-4C62970E-494C
        -4767-8723-891CCFF94B6C" targetRef="sid-708650D0-AC75-4507-9F43-4326B6AF421B"/>
  </collaboration>
  <!-- <process/>-Element definiert die Elemente eines Pools -->
  <process id="sid-FEED7AA8-F7B5-4347-8461-A278C859A25F" isClosed="false" isExecutable="false"
      name="Shop" processType="None">
    <!-- das <laneSet/>-Element wird innerhalb jedes Pools definiert -->
```

```
18    <laneSet id="sid-8463fdf3-2623-42a7-990e-e95989d5949f">
        <!-- Hier werden zusätzlich die Lanes mit <lane/>-Elementen definiert -->
20      <lane id="sid-20C2727C-A26C-4F5E-88AA-5FF9A399601A">
          <!-- innerhalb der Lanes folgt eine Referenz auf die Elemente -->
22        <flowNodeRef>sid-405E211F-1AE7-4819-B2B1-5BF6ACA665E3</flowNodeRef>
          <flowNodeRef>sid-4C62970E-494C-4767-8723-891CCFF94B6C</flowNodeRef>
24        <flowNodeRef>sid-DB0DED88-A42B-4341-99FE-75608BA6F3C7</flowNodeRef>
        </lane>
26    </laneSet>
      <!-- Definition der eigentlich Elemente des Prozessmodells -->
28    <!-- Startereignis -->
      <startEvent id="sid-405E211F-1AE7-4819-B2B1-5BF6ACA665E3" name="Neue Ware&#10;angekommen"
          >
30      <outgoing>sid-215D9197-CBE2-4561-AE00-ECA69386BF75</outgoing>
      </startEvent>
32    <!-- auslösendes Nachrichten-Zwischenereignis -->
      <intermediateThrowEvent id="sid-4C62970E-494C-4767-8723-891CCFF94B6C" name="Sende
          Benachrichtigung &#10;über neue Ware">
34      <incoming>sid-215D9197-CBE2-4561-AE00-ECA69386BF75</incoming>
        <outgoing>sid-64E62CC1-98E0-4B60-B786-74E185A37416</outgoing>
36      <!-- Definition des Nachrichtenereignisses -->
        <messageEventDefinition id="sid-ddc8af33-35a6-4f1b-a811-2be612156a82"/>
38    </intermediateThrowEvent>
      <!-- Endereignis -->
40    <endEvent id="sid-DB0DED88-A42B-4341-99FE-75608BA6F3C7" name="Ende">
        <incoming>sid-64E62CC1-98E0-4B60-B786-74E185A37416</incoming>
42    </endEvent>
      <!-- Zum Schluss des <process/> werden die Sequenzflüsse definiert -->
44    <sequenceFlow id="sid-215D9197-CBE2-4561-AE00-ECA69386BF75" sourceRef="sid-405E211F-1
          AE7-4819-B2B1-5BF6ACA665E3" targetRef="sid-4C62970E-494C-4767-8723-891CCFF94B6C"/>
      <sequenceFlow id="sid-64E62CC1-98E0-4B60-B786-74E185A37416" sourceRef="sid-4C62970E-494C
          -4767-8723-891CCFF94B6C" targetRef="sid-DB0DED88-A42B-4341-99FE-75608BA6F3C7"/>
46  </process>
    <!-- zweiter Pool mit einem <process/>-Element an den anderen angehängt -->
48  <process id="sid-E80336E7-4377-4900-8428-A6B85CCCB4CC" isClosed="false" isExecutable="false"
        name="Kunde" processType="None">
      <laneSet id="sid-b7ef7686-ec74-4984-9585-7cd1f09a1d1f">
50      <lane id="sid-49E78962-805E-4888-8096-CCFA499FF9F0">
          <flowNodeRef>sid-621A28B6-88CF-43DA-97E5-F6EC870F2940</flowNodeRef>
52        <flowNodeRef>sid-463D70CA-BF6C-4D8F-AF96-5A68B71EE781</flowNodeRef>
          <flowNodeRef>sid-708650D0-AC75-4507-9F43-4326B6AF421B</flowNodeRef>
54      </lane>
      </laneSet>
56    <!-- Aktivität -->
```

```
    <task completionQuantity="1" id="sid-621A28B6-88CF-43DA-97E5-F6EC870F2940"
          isForCompensation="false" name="Einkaufen gehen" startQuantity="1">
58    <incoming>sid-8C42A511-A679-4993-99E6-9984539F69B7</incoming>
      <outgoing>sid-11378FC5-60AC-4EFE-A9EA-4CC10C38173A</outgoing>
60  </task>
    <!-- Endereignis -->
62  <endEvent id="sid-463D70CA-BF6C-4D8F-AF96-5A68B71EE781" name="Ende">
      <incoming>sid-11378FC5-60AC-4EFE-A9EA-4CC10C38173A</incoming>
64  </endEvent>
    <!-- Nachrichten-Startereignis -->
66  <startEvent id="sid-708650D0-AC75-4507-9F43-4326B6AF421B" isInterrupting="true" name="
        Benachrichtigung über&#10;neue Ware erhalten">
      <outgoing>sid-8C42A511-A679-4993-99E6-9984539F69B7</outgoing>
68    <!-- Definition des Nachrichtenereignisses -->
      <messageEventDefinition id="sid-00580eab-c5ec-4161-a0e5-365d55615c02"/>
70  </startEvent>
    <sequenceFlow id="sid-11378FC5-60AC-4EFE-A9EA-4CC10C38173A" sourceRef="sid-621A28B6-88
        CF-43DA-97E5-F6EC870F2940" targetRef="sid-463D70CA-BF6C-4D8F-AF96-5A68B71EE781"/>
72  <sequenceFlow id="sid-8C42A511-A679-4993-99E6-9984539F69B7" sourceRef="sid-708650D0-AC75
        -4507-9F43-4326B6AF421B" targetRef="sid-621A28B6-88CF-43DA-97E5-F6EC870F2940"/>
  </process>
74  <!-- Hier folgt die Definition der grafischen Elemente des Prozessmodells -->
</definitions>
```

Listing 2.2: XML-Repräsentation eines Geschäftsprozesses

Nachdem alle Elemente im XML-Dokument aufgeführt worden sind, wird am Ende des Dokuments eine grafische Darstellung der Elemente dargestellt. Im engen Rahmen dieser Arbeit kann diese jedoch nicht behandelt werden. Nähere Informationen hierzu können [12] entnommen werden.

2.3 Domänen-Spezifische Abfragesprachen

Domänen-spezifische Abfragesprachen (DSL) sind Programmiersprachen, die sich mittels angepasster Notation und Abstraktion auf eine spezifische Domäne beziehen [13]. Im Folgenden werden zwei DSLs vorgestellt, die im *PQLParser Framework* benutzt werden. Zum Einen wird die *Cypher Query Language (CQL)*, zum Anderen die *Process Query Language (PQL)* vorgestellt. Mithilfe von CQL können Anfragen auf die *Graphendatenbank Neo4j* durchgeführt werden, um dort Informationen aus dort gespeicherten

Graphen zu erhalten; die Process Query Language erlaubt Anfragen durchzuführen, mit denen Graphen selektiert und angepasst werden können. Zunächst wird jedoch in Kapitel 2.3.1 das Prinzip der *Grammatik* vorgestellt und in Kapitel 2.3.2 der Begriff *Parser* spezifiziert.

2.3.1 Grammatik

Eine Grammatik ist ein *Regelsystem*, das *Wörter* einer vorgegebenen *Sprache* erzeugen kann [14]. Ein Regelsystem besteht aus *Umformungsregeln*. Umformungsregeln wiederum bestehen aus *Regeln*, wie z.B. `linke Seite` → `rechte Seite`. Dies bedeutet, dass die linke Seite durch die rechte Seite ersetzt werden darf. Als erste Regel wird dabei meist ein *Startsymbol (S)* verwendet. Das Startsymbol beschreibt den Beginn einer Grammatik. Die Symbole, wie z.B. das Startsymbol S werden als *Variablen* bezeichnet. Variablen dienen als Platzhalter und können beliebige Werte annehmen. Bei der *Anwendung* einer Regel wird die linke Seite durch die rechte Seite ersetzt. Dabei kann die linke Seite auch aus mehreren Variablen bestehen. Es werden so lange Regeln zur Umformung angewendet, bis nur noch sogenannte *Terminalsymbole* im Wort stehen. Terminalsymbole stellen die eigentlichen Symbole eines Wortes dar. Jedes durch diese Schritte erzeugte Wort gehört zu der von der Grammatik erzeugten Sprache [15, 16].

Listing 2.3 zeigt eine Beispiel-Grammatik für arithmetische Ausdrücke. Die Variablen `S`, `T`, `F` und `E` werden hier in Großbuchstaben dargestellt. Die Terminalsymbole sind `(`, `)`, `+`, `*` und `a`. Aus dieser Grammatik lassen sich arithmetische Ausdrücke als Wörter ableiten, wie z.B. `a * a * (a + a)+ a`.

```
Regelsystem = {
      S -> T,
      S -> S + T,
      T -> F,
      T -> T * F,
      F -> a,
      F -> (S)
}
```

Listing 2.3: Arithmetik-Grammatik (Quelle: nach [16])

2.3.2 Parser

Parser sind Programme, die *formale* oder *natürliche Sprachen* erkennen. Formale oder natürlich Sprachen stellen dabei, die von einer Grammatik (siehe Kapitel 2.3.1) erzeugten, Wörter dar. Daher können mit einem Parser Eingaben auf Korrektheit überprüft werden. Die Korrektheit einer Eingabe ist dann gegeben, wenn diese den Regeln der zugrundeliegenden Grammatik entspricht [17, 18]. Im Folgenden wird eine Eingabe als eine Anfrage einer DSL gesehen, d.h. diese besteht aus mehreren Wörtern, die wiederum aus einzelnen Buchstaben aufgebaut sind.

Der eigentliche Vorgang des Parsens kann in zwei Teile zerlegt werden: Zunächst werden einzelne Buchstaben zu sogenannten *Tokens* gruppiert. Die Gruppierung wird mithilfe der Regeln einer Grammatik durchgeführt. Dieser Vorgang wird als *lexikalische Analyse* bezeichnet und ein Programm, das diese Funktion durchführt, wird *Lexer* genannt. Im nächsten Schritt wird mithilfe der Tokens die Struktur der Eingabe analysiert. Dieser Teil eines Programms wird als Parser bezeichnet. Dabei kann die Eingabe in eine *Zwischenrepräsentation* überführt werden, indem den Tokens Werte zugewiesen werden. Diese Zwischenrepräsentation kann anschließend weiter verarbeitet werden [18]. So können beispielsweise Anfragen von DSLs erkannt und dementsprechende Operationen ausgeführt werden. Die technische Umsetzung eines Parser wird in Kapitel 3.4.1 näher erläutert.

2.3.3 Cypher Query Language

Die Cypher Query Language erlaubt es Anfragen und Änderungen auf Graphendatenbanken zu definieren und auszuführen. Im Speziellen kann dies auf der Graphendatenbank Neo4j beschehen, die in Kapitel 3.3.2 näher erläutert wird. In CQL können Elemente eines Graphen selektiert, erstellt, abgeändert und wieder gelöscht werden. Ausgehend von einem Startknoten werden sogenannte *Muster* bestimmt und über die Kanten weitere verbundene Knoten gesucht. In der Anfrage können zusätzlich Attribute angegeben werden, um die Suche von Mustern einzuschränken. CQL ist eine deklarative Abfragesprache, d.h. in der Anfrage selber werden Informationen gegeben, um die

MATCH	Repräsentiert das gesuchte Pattern, bestehend aus Knoten und Kanten
WHERE	Filteroption für das ausgewählte Pattern im MATCH-Teil
RETURN	Gibt an, welche Attribute, Knoten und Kanten zurückgegeben werden
START	Bestimmt Startpunkte für die Suche nach Patterns (optional, wenn START nicht vorkommt übernimmt dies MATCH)
CREATE, MERGE	Erstellen von Knoten und Kanten
DELETE	Löschen von Knoten und Kanten
SET, REMOVE	Erstellen und Löschen von Attributen
FOREACH	Iteration über eine Liste von Werten und Operationsausführung
WITH	Ermöglicht die Verkettung von Abfragen
ORDER BY, SKIP, LIMIT	Sortieren von Paginierung

Tabelle 2.1: Schlüsselworte für CQL-Anfragen (Quelle: [19, 21])

gesuchten Elemente zu finden. Dazu wird aber kein Weg zur Findung der Informationen angegeben, dies geschieht ausschließlich über die sogenannte Mustersuche [19].

Bei einer Anfrage in CQL wird textuell definiert, welche Knoten gesucht werden sollen und was anschließend mit diesen gemacht wird. Der Aufbau von CQL ist angelehnt an den der *Structered Query Language (SQL)* [20]. CQL-Anfragen bestehen aus diversen Schlüsselworten. Beispielschlüsselwörter für eine einfache Anfrage sind *MATCH*, *WHERE* und *RETURN*.

Mithilfe des MATCH-Teils können Muster bestimmt werden, nach welchen gesucht werden sollen. Ein Beispiel hierfür wäre die Suche nach zwei aufeinanderfolgende Knoten a und b und die dazugehörige Anfrage würde `MATCH (a)-->(b)` lauten. a und b dienen nur als Platzhalter und entsprechen nicht den Namen der gesuchten Knoten. Im WHERE-Teil einer CQL-Anfrage können die im MATCH-Teil bestimmten Muster weiter gefiltert werden. Im RETURN-Teil der Anfrage werden diejenigen Elemente bestimmt, die dem Nutzer angezeigt werden sollen. Tabelle 2.1 bietet eine Übersicht der wichtigsten Schlüsselwörter und deren Erklärung.

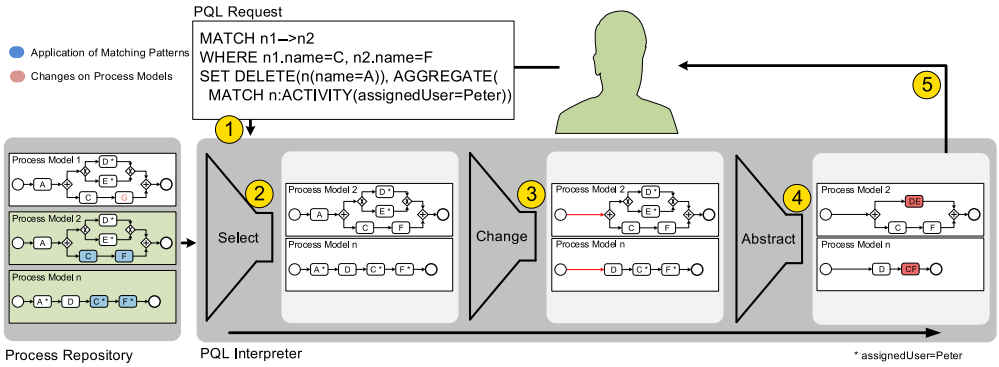

Abbildung 2.6: Ablauf einer PQL-Anfrage (Quelle: [25])

2.3.4 Process Query Language

Die *Process Query Language (PQL)* ist eine an der Cypher Query Language angelehnte DSL. Mithilfe von PQL können *Anfragen* auf graph-basierte Prozessmodelle gestellt werden [22]. Damit können beispielsweise Bereiche von Prozessmodellen selektiert und auf diesen Änderungsoperationen ausgeführt werden. Zusätzlich können durch *Abstraktion* zum einen Aktivitäten von Prozessmodellen verborgen werden (*Reduktion*) und zum anderen eine Menge von Aktivitäten zu einer abstrakten Aktivität zusammengefügt werden (*Aggregation*) [23, 24]. Im Rahmen dieser Arbeit wird der Teilbereich Abstraktion nicht weiter betrachtet, eine detaillierte Beschreibung kann [25] entnommen werden. PQL erlaubt nicht nur Anfragen auf einzelne Prozessmodelle, sondern lässt ebenfalls Anfragen auf mehrere Prozessmodelle gleichzeitig zu [25].

Abbildung 2.6 zeigt den Ablauf einer PQL-Anfrage auf eine Ansammlung von Prozessmodellen. Dieser beginnt mit der Formulierung einer Anfrage in PQL, in welcher zwei aufeinanderfolgende Knoten mit den Namen C und F ausgewählt werden sollen. Auf die gefundenen Prozessmodelle wird anschließend eine DELETE-Operation angewendet. Danach werden die Aktivitäten aggregiert, die von der Person `Peter` ausgeführt werden sollen. Nachdem die Operationen ausgeführt wurden, werden die resultierenden Prozessmodelle dem Benutzer angezeigt.

Eine PQL-Anfrage besteht aus zwei Teilen, der *Auswahlteil* und dem *Änderungsteil*. Im Auswahlteil werden die Prozessmodelle selektiert und im Änderungsteil werden Änderungen auf Prozessmodellen durchgeführt.

Der Auswahlteil einer PQL-Anfrage ist folgendermaßen aufgebaut: zuerst wird mit dem *MATCH-Teil* der Anfrage die Struktur bestimmt, nach der gesucht wird. Im *WHERE-Teil* der Anfrage kann zusätzlich nach Attributen gefiltert werden, so z.B. auch nach der ID oder dem Namen einer Prozessaktivität. Die Filterung der Elemente bezieht sich dabei, auf die zuvor ausgewählte, Struktur im MATCH-Teil. Eine Kombination von MATCH und WHERE ist ebenfalls möglich, so kann z.B. mit `MATCH a(42)-[ :sequenceFlow ]->b`, bei der der erste Knoten die ID 42 hat, nach zwei Knoten gesucht werden, die mit einem Sequenzfluss verbunden sind. Äquivalent dazu wäre die Anfrage: `MATCH a-[ :sequenceFlow ]->b WHERE a.id='42'`.

Im Änderungsteil (*SET-Teil*) der Anfrage können Änderungen auf selektierten Modellen angewandt werden. Dabei werden die in Kapitel 2.1.3 vorgestellten Adaptation Patterns unterstützt.

```
MATCH a-->b
WHERE a.Name=C, b.Name=F
SET INSERTNODE(a, b, ACTIVITY, 'New Node')
```

Listing 2.4: Beispiel einer PQL-Anfrage zum Einfügen einer Aktivität (Quelle: [25])

In Listing 2.4 ist eine Beispielanfrage aufgeführt, die eine neuen Aktivität zwischen die bestehenden Knoten a und b einfügt.

3

PQLParser Framework

In diesem Kapitel wird die prototypische Implementierung des PQLParser Frameworks vorgestellt. Mit diesem Framework können Prozessmodelle verwaltet und Änderungen darauf ausgeführt werden. Dazu werden mehrere Prozessmodelle (z.B. aus einem PAIS) in eine separate Datenbank geladen. Zur Ausführung von Änderungen werden vordefinierte High-Level-Operationen bereitgestellt, die die Durchführung von Änderungen auf mehreren Prozessmodellen ermöglichen. Durch die separate Speicherung der Prozessmodelle werden keine unerwünschten Änderungen auf Prozessmodellen durchgeführt, die nicht geändert werden sollen. Das Framework ermöglicht so auch die Verwaltung von mehreren Varianten eines Prozessmodells.

Im Folgenden werden zunächst Anwendungsfälle betrachtet, aus denen *funktionalen Anforderungen* und *nicht-funktionalen Anforderungen* für das Framework abgeleitet werden. Anschließend wird die Softwarearchitektur und technische Implementierung des PQLParser Frameworks vorgestellt.

Die einzelnen Module, die sich daraus ergeben, werden gesondert betrachtet und deren Funktionen näher erläutert. Dazu werden in Kapitel 3.3 die verwendeten externen Tools, die bei der Umsetzung des Frameworks helfen, vorgestellt. Danach werden in Kapitel 3.4 elementare Teile der Implementierung beschrieben. Kapitel 3.4.1 beschreibt, wie mithilfe eines Parsers eine PQL-Anfrage in eine Zwischenrepräsentation (CQL-Request) übergeführt wird. Kapitel 3.4.2 erläutert, wie die übersetzten CQL-Requests weiter verarbeitet werden. Darauf folgend wird in Kapitel 3.4.3 die Schnittstelle zur Graphendatenbank Neo4j vorgestellt. Abschließend werden in Kapitel 3.4.4 und 3.4.5 der Import und Export von Prozessmodellen in eine Graphendatenbank mit Unterstützung des PQLParser Framework beschrieben.

3.1 Anforderungsanalyse

Aufgrund einer Vielzahl vorhandener Geschäftsprozesse in Unternehmen verliert ein Nutzer schnell den Überblick über alle Prozessmodelle. Zusätzlich müssen Prozessmodelle angepasst werden, damit diese den Realweltprozessen entsprechen. Änderungen manuell auf Prozessmodelle zu übertragen ist zum einen zeitintensiv und zum anderen fehleranfällig [26]. Um dem entgegenzuwirken, wird ein *Repository* benötigt, in das Prozessmodelle geladen werden können. Aus diesem Repository können diese Prozessmodelle abgerufen und bearbeitet werden. Abbildung 3.1 zeigt eine schematische Darstellung der beschriebenen Vorgänge.

Weiterhin soll es möglich sein bestimmte Prozessmodell-Elemente, wie z.B. Aktivitäten oder Ereignisse, in einem Repository zu suchen und gefundene Prozessmodelle abzurufen. Dafür muss der Nutzer eine Suchanfrage an das Repository stellen können, das anschließend durchsucht wird und einem Nutzer zutreffende Modelle in einem in der Anfrage ausgewählten Format anzeigt oder zurückgibt (siehe Abbildung 3.2).

Außerdem soll es möglich sein, Änderungen an bestehenden Prozessmodellen durchzuführen (siehe Abbildung 3.3). Dabei werden zunächst die zu ändernden Kontrollflusselemente selektiert (siehe ① und ②) und von nicht betroffenen Prozessmodellen separiert

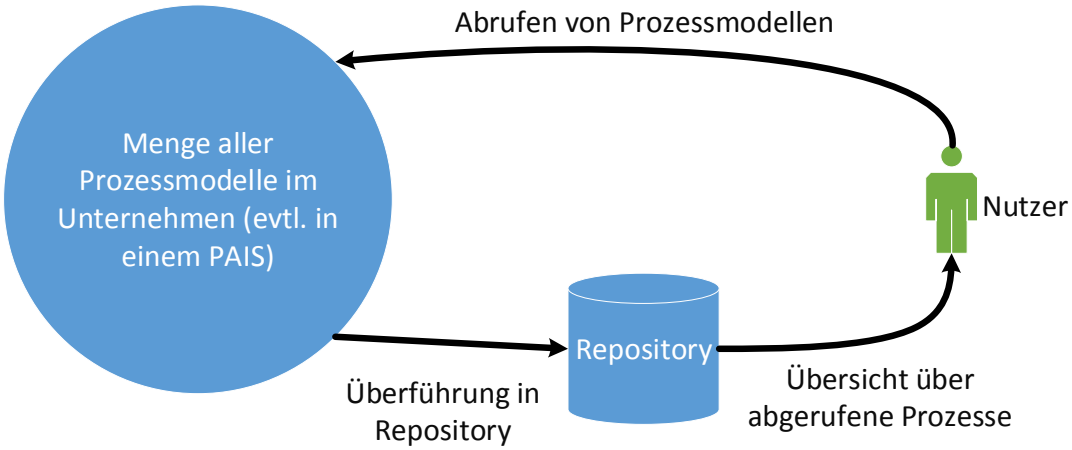

Abbildung 3.1: Schematische Darstellung des PQLParser Frameworks

(siehe ③). Anschließend werden die Änderungen ausgeführt (siehe ④ und ⑤). Danach soll das Prozessmodell wieder im Repository gespeichert werden (siehe ⑥).

Aus diesen Anwendungsfällen lassen sich die folgenden funktionalen und nicht-funktionalen Anforderungen ableiten.

3.1.1 Funktionale Anforderungen

In diesem Abschnitt werden die funktionalen Anforderungen (FA) an das PQLParser Framework vorgestellt und beschrieben. Anhand diesen Anforderungen wird der Funktionsumfang des Frameworks definiert. Die Implementierung der einzelnen Funktionen wird im weiteren Verlauf dieses Kapitels näher erläutert.

FA 1: Import von Prozessmodellen im BPMN-Format

Mit dem Framework soll es möglich sein, Prozessmodelle im BPMN-Format einzulesen und in ein internes Modell zu überführen. Dafür stehen die Prozessmodelle als XML-Datei zur Verfügung, wie sie beispielsweise aus einem Prozesseditor exportiert werden können.

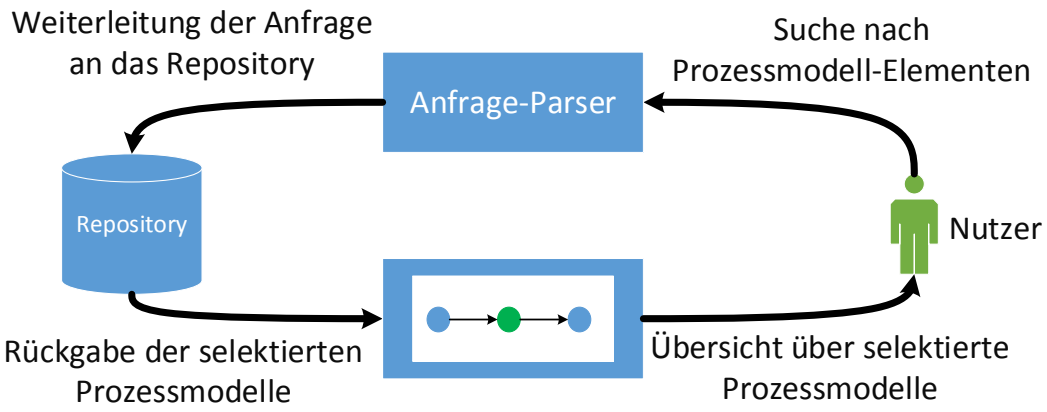

Abbildung 3.2: Suche nach Prozessmodell-Elementen in einem Repository

FA 2: Abbildung von Prozessmodellen in Neo4j-Datenbank

Das interne Modell, das aus einer XML-Datei erzeugt wird, soll auf ein Modell einer Neo4j-Graphendatenbank abgebildet werden.

FA 3: Eingabe und Interpretation von PQL-Anfragen

Dem PQLParser Framework sollen PQL-Anfragen übergeben werden können, die interpretiert und zur Ausführung gebracht werden.

FA 4: Übersetzung von PQL-Anfragen in abstrakte Abfragesprache

Um die Abbildung von Änderungsoperationen auf Prozessmodellen unabhängig von der zugehörigen Graphendatenbank zu implementieren, sollen die interpretierten PQL-Anfragen als Abstraktionsschritt in eine abstrakte, technologie-unabhängige Abfragesprache übersetzt werden.

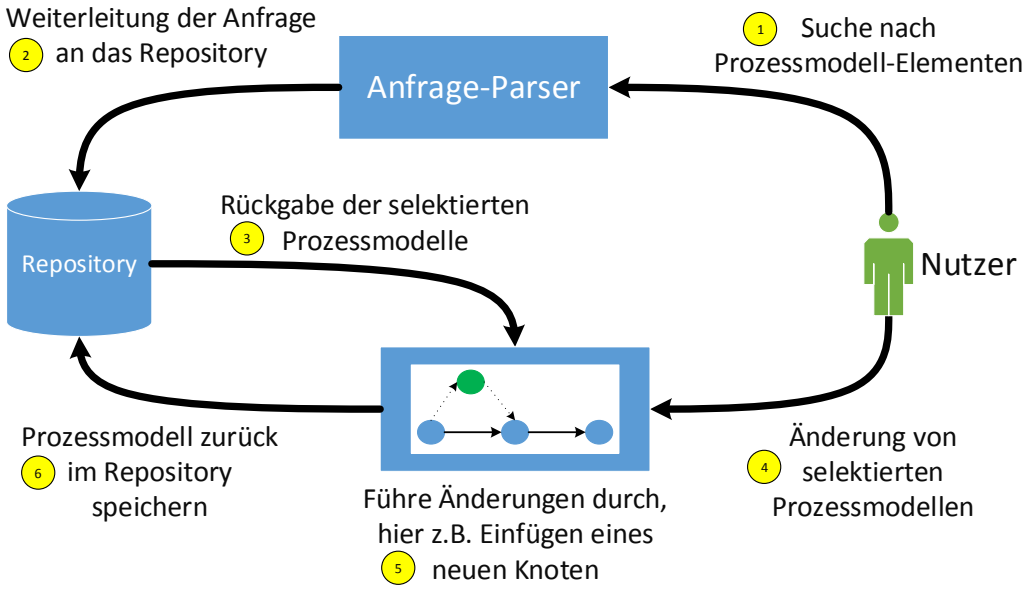

Abbildung 3.3: Änderung eines Prozessmodells

FA 5: Ausführung von abstrakter Abfragesprache auf einer Neo4j-Datenbank

Die interpretierten Anfragen der abstrakten Abfragesprache sollen auf einer Graphendatenbank, im Speziellen einer Neo4j-Datenbank, ausgeführt werden können.

FA 6: Export von Prozessmodellen aus einer Neo4j Datenbank

Das Framework soll einen Export von einem Prozessmodell in das definierte interne Modell ermöglichen.

FA 7: Darstellung von Prozessmodellen in BPMN-Format

Das zuvor definierte interne Modell soll in eine XML-Datei geschrieben werden können und einem konformen BPMN-Format entsprechen. Anschließend soll ein Prozessmodell im XML-Format in einen Prozesseditor importiert werden können.

3.1.2 Nicht-funktionale Anforderungen

In diesem Abschnitt werden die nicht-funktionalen Anforderungen (NFA) an das PQL-Parser Framework betrachtet. Dadurch soll sichergestellt werden, dass das Framework definierten Qualitätsmerkmalen entspricht.

NFA 1: Einfache Struktur der Abfragesprache

Die Struktur der Abfragesprache, mit der Prozessmodelle selektiert und Änderungen ausgeführt werden, soll einfach zu verstehen und anzuwenden sein. Zudem soll die Sprache menschenlesbar und leicht erlernbar sein.

NFA 2: Modularisierung und einfache Erweiterbarkeit

Es soll leicht möglich sein, das Framework zu erweitern und neue Module hinzuzufügen.

NFA 3: Datenbankschnittstelle soll gut gekapselt sein

Es soll möglich sein, die verwendete Datenbank ohne großen Aufwand austauschen zu können.

3.2 Architekturschema

In diesem Abschnitt wird die Architektur des PQLParser Framework vorgestellt.

Abbildung 3.4 zeigt den Ablauf einer PQL-Anfrage an das PQLParser Framework. Zunächst wird die Anfrage an das *PQLParser-Modul* geleitet, bei dem aus der PQL-Anfrage ein CQL-Request erstellt wird (siehe ①). Ein CQL-Request wird weiter an das *CQLTransformer-Modul* gegeben. Im sogenannten Transformer wird der CQL-Request interpretiert und eventuell in mehrere CQL-Anfragen aufgeteilt (siehe ②). Die generierten CQL-Anfragen werden wiederum an die *Datenbankschnittstelle* weitergegeben

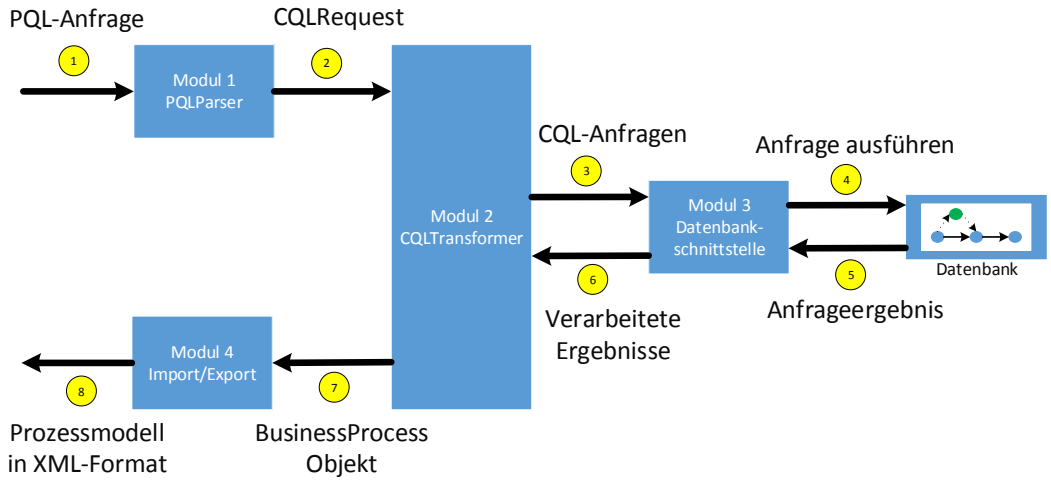

Abbildung 3.4: Ablauf einer Anfrage an das PQLParser Framework

(siehe ③). Die CQL-Anfragen werden schließlich auf der Datenbank ausgeführt und eventuelle Ergebnisse davon zurückgegeben (siehe ④ und ⑤). Die Ergebnisse der einzelnen CQL-Anfragen werden verarbeitet und an den Transformer zurückgegeben (siehe ⑥). Im CQLTransformer wird anschließend eine Zwischenrepräsentation eines Prozessmodells erstellt. Danach wird das Prozessmodell als XML-Datei ausgegeben (siehe ⑧). Hierfür wird das Prozessmodell an das *Import/Export-Modul* übergeben (siehe ⑦) Im Folgenden werden die genannten Module näher beschrieben. Abbildung 3.5 zeigt das daraus resultierende Klassendiagramm.

3.2.1 PQLParser

Das Modul PQLParser ist dafür zuständig, eine PQL-Anfrage in ein CQL-Request zu übersetzen. In diesem Modul gibt es zwei Komponenten, die bei der Übersetzung helfen: zum einen gibt es den PqlRequestController, der das Übersetzen verwaltet und die PQL-Anfrage an einen PQLVisitor weitergibt. Der PQLVisitor übersetzt nun die Anfrage mithilfe von dem mit ANTLR generierten Parser. Die Funktionsweise von ANTLR und des Parsers wird in Kapitel 3.3.1 näher beschrieben. Nach dem Übersetzen wird der

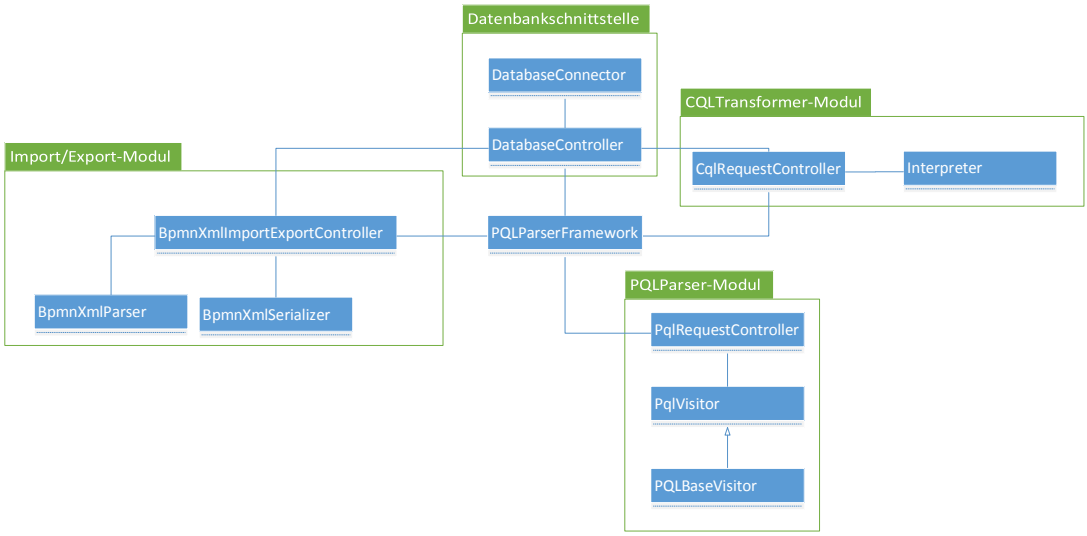

Abbildung 3.5: Klassendiagramm des PQLParser Framework

resultierende CQL-Request an den PqlRequestController zurückgegeben und kann nun an das Modul CQLTransformer übergeben und weiter verarbeitet werden.

3.2.2 CQLTransformer

In diesem Modul wird ein CqlRequest in mehrere CQL-Anfragen übersetzt. Die Anzahl der Anfragen hängt von der Änderungsoperation ab, die ausgeführt werden soll. Dabei muss die jeweilige Operation aus der PQL-Anfrage umgesetzt werden. Die erstellten CQL-Anfragen werden anschließend zum Ausführen an die Datenbankschnittstelle weitergegeben und die Ergebnisse zusammengefasst. Schließlich wird ein internes Modell eines Prozessmodells erstellt und an das Import/Export-Modul weitergegeben.

3.2.3 Datenbankschnittstelle

Dieses Modul verwaltet die Schnittstelle zur Datenbank und führt übergebene CQL-Anfragen darauf aus. Nach der Ausführung von Anfragen werden die Ergebnisse verarbeitet und an das CQLTransformer-Modul zurückgegeben. Als Datenbank wird in dieser

Arbeit beispielhaft die Neo4j-Datenbank verwendet, die in Kapitel 3.3.2 näher erläutert wird.

3.2.4 Import/Export

Das Import/Export-Modul ist dafür zuständig Prozessmodelle zu importieren, zu exportieren und als XML-Datei auszugeben. Es besteht aus drei Teilen: einem Controller, einem Import und einem Export. Der Controller nimmt Aufgaben entgegen und verteilt diese an die entsprechenden Teile weiter. Ein Import und Export von Prozessmodellen wird separat voneinander verwaltet. Der Import erstellt aus einem Prozessmodell im XML-Format eine interne Repräsentation. Der Export hingegen soll aus der internen Repräsentation eines Prozessmodells eine XML-Datei im BPMN-Format erstellen.

3.3 Verwendete Tools

Im folgenden Abschnitt werden zwei, im PQLParserFramework verwendete, Tools genannt und ihre Funktionsweise beschrieben. Zunächst wird der ANTLR - Parser Generator vorgestellt, Kapitel 3.3.2 beschreibt die Graphendatenbank Neo4j.

3.3.1 ANTLR – Parser Generator

ANTLR ist ein Framework zur automatischen Generierung eines Parsers. Anhand einer definierten Grammatik werden Lexer und Parser (siehe Kapitel 2.3.2) erstellt, mit denen unter anderem auch DSLs, wie z.B. PQL, eingelesen, verarbeitet und übersetzt werden können. Ein bekanntes Beispiel, bei dem ANTLR im Einsatz ist, ist die *Twitter Search Engine*, bei der eingehende Anfragen auf die Search Engine geparst und verarbeitet werden [18, 27].

Der Vorgang des Parsens ist in Abbildung 3.6 grafisch dargestellt. Ziel dieses Vorgangs ist ein sogenannter *Parsebaum*. In einem Parsebaum können Wörter (oder auch Anfragen einer DSL) aus der erzeugten Sprache einer Grammatik dargestellt werden (vgl.

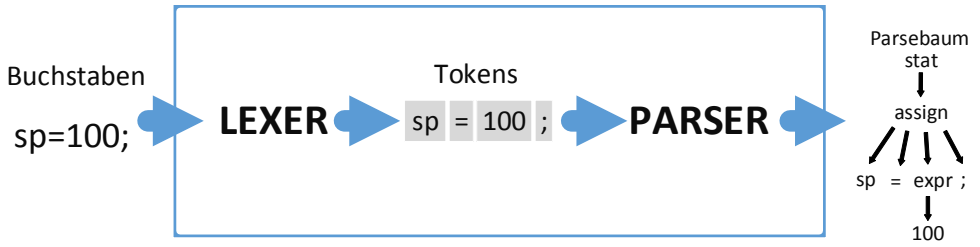

Abbildung 3.6: Grafisch dargestellter Vorgang eines durch ANTLR erstellten Parsers (Quelle: nach [18])

Kapitel 2.3.1). Dabei stellen die Blätter des Baumes die Terminalsymbole und die Knoten die Variablen der Grammatik dar. Um einen Parsebaum erstellen zu können, muss der Input zunächst zu Tokens gruppiert werden. Anschließend wird durch den Parser eine Repräsentation des Inputs als Baum erstellt. Abbildung 3.7 zeigt als Beispiel einen Parsebaum der PQL-Anfrage `MATCH a-[ :sequenceFlow ]->b WHERE a.id='test'.`

Das Analysieren der Grammatik und das Aufbauen des Parsebaumes basiert auf der Parsing-Technologie *Adaptive-LL(*)* oder *ALL(*)*, nähere Informationen hierzu können [28] entnommen werden. In dieser Arbeit wird ein kurzer Einblick in den Funktionsumfang von ANTLR gegeben.

Nachdem der Parsebaum gebildet wurde, stellt ANTLR zwei Möglichkeiten bereit diesen zu durchlaufen [18]: zum einen die *Listener-Methode* und zum anderen die *Visitor-Methode*. Bei beiden Methoden kann der Parsebaum analysiert werden.

Bei der Listener-Methode werden alle Elemente des Baumes sukzessive durchlaufen und bei jedem Element (sowohl Knoten, als auch Blätter des Baumes) werden Events gefeuert. Events werden dann gefeuert, wenn der Start und das Ende eines Elements durchlaufen werden. Dabei besteht bei jedem Element die Möglichkeit, eine `enter()` und eine `exit()`-Methode auszuführen, in denen ein eigens implementierter Code umgesetzt werden kann.

Die Visitor-Methode baut auf das *Visitor-Pattern* auf. Dafür werden sogenannte *Visitors* implementiert, mit denen das Durchlaufen des Parsebaumes kontrolliert werden kann. Mit der Methode `visit()` kann ein Durchlauf gestartet werden. Weitere Knoten und

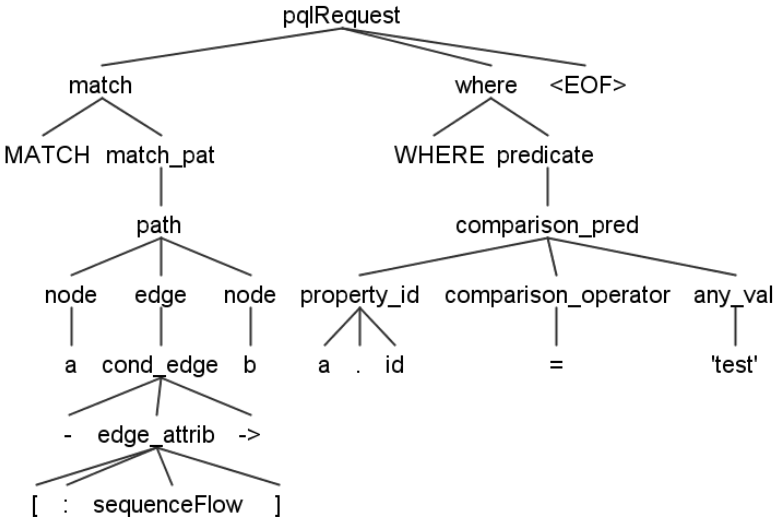

Abbildung 3.7: Beispiel eines Parsebaumes

Blätter können dann mit `visitKnoten()` besucht werden. Bei dem Besuch eines Elements kann schließlich die eigene Implementierung ausgeführt werden. Ein Vorteil der Visitor-Methode ist, dass der Parsebaum nicht in einer vordefinierten Reihenfolge durchlaufen werden muss, sondern einzelne Elemente separat und auch mehrmals angesteuert werden können. Zudem müssen nicht alle Knoten durchlaufen werden [18].

Für beide Methoden stellt ANTLR ein Interface bereit, mit dem die eben beschrieben Methoden implementiert werden können. Das PQLParser Framework verwendet die Visitor-Methode.

3.3.2 Graphendatenbank

Um ein Prozessmodell in einer Datenbank abbilden zu können, ist eine Datenstruktur nötig, die Elemente, wie z.B. Aktivitäten, Ereignisse und Gateways, darstellen kann. Gleichzeitig muss es möglich sein, Sequenz- und Nachrichtenflüsse zu modellieren. Graphendatenbanken sind für die Darstellung von vernetzten Daten eine gute Alternative, da sie zum einen die Graphenstruktur von Prozessmodellen unterstützen und zum anderen dafür ausgelegt sind, Graphen effizient abzufragen [19].

Allgemeines

Eine Graphendatenbank besteht aus *Graphen*. Ein Graph besteht aus *Knoten* und *Kanten*, wobei immer zwei Knoten mit einer Kante verbunden werden. In Graphendatenbanken werden Informationen in Knoten, Kanten und der Verbindung von Elementen dargestellt [19].

Ein großer Vorteil von Graphendatenbanken ist die Vernetzungen von Informationen. Da Informationen nicht in verschiedenen Tabellen gespeichert werden, müssen keine sogenannten *Join-Operationen* ausgeführt werden, wie sie beispielsweise in SQL formuliert werden. Mithilfe von Join-Operationen können in relationalen Datenbanken verschiedenen Tabellen zusammengeführt werden. Zu beachten ist, dass Join-Operationen rechenintensive Operationen sind und möglichst vermieden werden sollten. Der Aufwand für JOIN-Operationen steigen mit der Anzahl an Tabellen, die zusammengeführt werden sollen. In Graphendatenbanken werden Informationen durch das Besuchen benachbarter Knoten gesucht. Die Performance dieser Operation ist linear zu der Anzahl an gespeicherten Knoten [29, 30]. Da Graphen und Graphenalgorithmen in der Mathematik bereits ausführlich erforscht sind, können bereits bekannten Prinzipien auf die Graphendatenbank übertragen werden. Ein Beispiel hierfür ist der Dijkstra-Algorithmus, welcher die Möglichkeit bietet, den kürzesten Pfad durch einen Graphen zu finden [31, 30].

Neo4j

Neo4j ist eine in Java implementierte Open-Source-Graphendatenbank, die auf das *Property-Graph-Modell* aufbaut. Das Property-Graph-Modell besteht aus vernetzten Knoten. Knoten können mit *Labels* versehen werden. Ein Label weißt einem Knoten einen Text zu, welcher den *Status* oder die *Rolle* widerspiegeln. Dadurch können mehrere Knoten zu einer Gruppe zusammengefasst werden. Beispielsweise stellt die Rolle `Person` dar, dass es sich bei dem Knoten um eine Person handelt. Zudem können beliebig viele Attribute an einen Knoten angehängt werden [21]. Zusätzlich existieren Kanten, die ebenfalls mit Labels und wiederum beliebig vielen Schlüssel-Wert-Paaren versehen werden können, die die Knoten untereinander verbinden. Kanten sind gerichtete Ver-

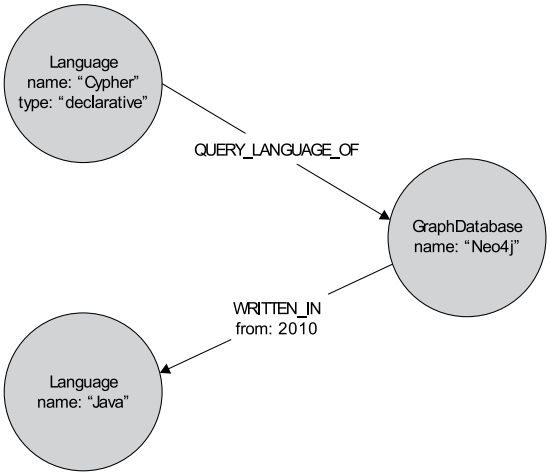

Abbildung 3.8: Beispiel eines Neo4j-Graphen (Quelle: [33])

bindungen zwischen zwei Knoten und haben stets einen Start- und einen Endknoten [32].

In Abbildung 3.8 ist ein Beispielgraph abgebildet, in dem drei Knoten dargestellt sind. Diese Knoten haben jeweils ein Label (hier: `Language` und `GraphDatabase`) und dazu noch unterschiedliche Attribute (hier: z.B. `name:"Cypher"`). Die Verbindungen von Knoten haben ebenfalls ein Label (hier: z.B. `QUERY_LANGUAGE_OF`) und können wiederum mit Attributen ausgestattet werden (hier: z.B.`from:2010`).

3.4 Prototypische Implementierung

Im Folgenden wird die Implementierung der in Kapitel 3.2 beschriebenen Module vorgestellt. Zuerst wird das Modul PQLParser beschrieben, anschließend wird die Transformation von CQLRequests erläutert. Danach wird erklärt, wie die Datenbankschnittstelle implementiert ist. Abschließend wird der Import und Export von Prozessmodellen in das Framework beschrieben.

3.4.1 PQLParser

In diesem Abschnitt wird das PQL-Parser-Modul beschrieben und die Umsetzung erläutert. In diesem Modul wird das in Kapitel 3.3.1 vorgestellte Framework ANTLR angewendet, um eingehende PQL-Anfragen zu parsen und in eine Zwischenrepräsentation zu überführen. Zur Erstellung des Parsers benötigt ANTLR eine Grammatik, welche die Sprachkonstrukte von PQL abbildet. Daher wird zunächst der Aufbau dieser Grammatik beschrieben und anschließend dargelegt, wie eine PQL-Anfrage in eine Zwischenrepräsentation überführt wird.

ANTLR-Grammatik von PQL

In Kapitel 2.3.1 wurden Grammatiken bereits vorgestellt. Der allgemeine Aufbau der Grammatik, die ANTLR zum Erstellen eines Parsers benötigt, weist allerdings einige Unterschiede auf. Nachfolgend wird der Aufbau einer Grammatik für den ANTLR-Parser anhand der PQL-Grammatik dargestellt. Zum besseren Verständnis wird die Grammatik in fünf Teile aufgeteilt. Zunächst wird der Start der Grammatik gezeigt und anschließend werden die drei Komponenten MATCH, WHERE und SET (vgl. Kapitel 2.3.4) separat betrachtet. Zum Schluss wird der Teil der Grammatik angegeben, der die Regeln angibt, die auf Terminalsymbole verweisen.

Listing 3.1 zeigt den ersten Teil der PQL-Grammatik. Start der Grammatik bildet die Definition mit dem Schlüsselwort `grammar`, damit wird der Name der Grammatik angegeben (hier: `PQL`). Anschließend wird das Package deklariert, in dem ANTLR die Java-Dateien erstellen soll. Danach folgt mit dem Startsymbol der Beginn der eigentlichen Grammatik (hier: `pqlRequest`).

Das Startsymbol ist gleichzeitig das *Wurzelelement* des Parsebaumes (vgl. Abbildung 3.7). An der ersten Regel kann der Unterschied der Syntax zu der in Kapitel 2.3.1 vorgestellten Grammatik erkannt werden. Die Schreibweise der Regeln folgt nach dem Muster: `linke Seite : rechte Seite`, die Funktionsweise der Regeln ist weiterhin dieselbe. Die Variablen werden so lange ersetzt, bis ausschließlich Terminalsymbole im Wort stehen (vgl. Kapitel 2.3.1).

Regeln können aus den folgenden vier Patterns aufgebaut werden: *Sequenz*, *Entscheidung*, *Abhängigkeit von Tokens* und *verschachtelte Ausdrücke*. Eine Sequenz ist eine Abfolge von Elementen, beispielsweise besteht das Wort `MATCH` aus einer Abfolge von fünf Buchstaben. Mit Entscheidung kann die Auswahl aus mehreren alternativen Möglichkeiten beschrieben werden. Dies wird mit dem `|`-Symbol dargestellt. Mithilfe der Abhängigkeit von Tokens kann modelliert werden, dass gewisse Tokens ein Vorkommen eines weiteren Tokens voraussetzen. Ein Beispiel hierfür ist das Setzen der schließenden Klammer `)`, nachdem die öffnende gesetzt wurde. Verschachtelte Ausdrücke erlauben die Referenzierung von Regeln auf sich selbst.

Eine PQL-Anfrage besteht aus einem verpflichteten MATCH-Teil, danach kommen optional die Teile WHERE und SET, gekennzeichnet mit einem Fragezeichen nach dem Variablennamen. Mit einem `*`-Symbol kann das beliebige Vorkommen einer Variable modelliert werden. Das `EOF`-Symbol am Ende der Zeile gibt das explizite Ende einer Anfrage an, d.h. nachdem dieses Symbol erkannt wurde, folgen keine weiteren Symbole. Das Ende einer Regel wird durch das Semikolon angezeigt.

```
//Definition der Grammatik
grammar PQL;
//Angaben des Packages für Java
@header{
package pqlParserANTLR;
}
//Startsymbol oder Wurzelelement der Grammatik
pqlRequest : match where? set? EOF;
```

Listing 3.1: Start der PQL-Grammatik

In Listing 3.2 wird der MATCH-Teil der PQL-Grammatik dargestellt. Da die `match`-Regel verpflichtend ist, wird jede PQL-Anfrage mit einem `MATCH`-Schlüsselwort gestartet. In diesem Teil werden Pfade zur Selektion von Prozessmodellen bestimmt. Diese Pfade können dabei aus Knotennamen und Kanten bestehen. Beispielsweise selektiert `MATCH a -[:sequenceFlow]->b` einen Pfad, der aus den zwei Knoten a und b besteht, die mit einem Sequenzfluss miteinander verbunden sind. Das Suchen nach mehreren Pfaden wird ebenfalls unterstützt. Dabei können weitere Pfade, mit einem Komma (,) getrennt, an den ersten angehängt werden. Bei der Angabe von mehreren Pfaden werden Ände-

rungsoperationen, die im SET-Teil bestimmt werden, für jeden Pfad separat ausgeführt. Weitere Optionen sind die Angabe einer `match_function`, die z.B. das Anwenden des Dijkstra-Algorithmus ermöglicht.

```
//Auswahlteil eine PQL-Anfrage mit der Bestimmung der Struktur des Pattern, nachdem
    gesucht wird
match : 'MATCH' match_pat (separater match_pat)*;
match_pat : (ID EQUAL)? (match_function '(' path ')' | path);
//Ein Pfad besteht aus einem Knoten und beliebig viele folgenden Knoten mit Kanten
    dazwischen
path : node (edge node)*;
//Auf Knoten können zusätzlich Attribute abgebildet werden
node :  ID ( node_type)? ( node_id)?;
//entspricht dem Typ des Elements, z.B. exclusiveGateway
node_type : ':' ID;
//Angabe der ID eines Knoten mit umschließender Klammerung
node_id : '(' ID ')';
//Bei Kanten besteht die Möglichkeit gerichtet oder ungerichtete Kanten abzubilden
edge : cond_edge | uncond_edge;
uncond_edge : (('--')|('-->'));
cond_edge : (('-' edge_attrib '-') | ('-' edge_attrib '->'));
//Den Type der Kante kann hiermit bestimmt werden, z.B. sequenceFlow
edge_attrib : '[' ID? (':'(( ID ('|:' ID )*)? edge_quant?)| edge_quant)? ']';
//Möglichkeit zur Bestimmung der Anzahl an dazwischenliegenden Kanten zwischen zwei
    Knoten
edge_quant : '*' (ID | ( ID '..' ID))?;
```

Listing 3.2: MATCH-Teil der PQL-Grammatik

Im WHERE-Teil der Anfrage (siehe Listing 3.3) können Pfade durch Filterung nach Attributen näher bestimmt werden. Der WHERE-Teil ist lediglich optional, muss also nicht in jeder Anfrage vorkommen. Das vorherige Beispiel der zwei Knoten a und b kann hierbei erweitert werden. So bestimmt die Anfrage `MATCH a-[:sequenceFlow]->b WHERE a .id='test'` zunächst zwei Knoten a und b, die durch einen Sequenzfluss miteinander verbunden sind. Zusätzlich soll der Knoten a die ID `test` haben. Es können aber auch *Reguläre Ausdrücke* angegeben werden, um nach Attributen zu filtern.

```
//Auswahlteil einer PQL-Anfrage mit der Filterung nach Attributen
where : 'WHERE' predicate_bool_operator? predicate ((bool_operator
    predicate_bool_operator? predicate)+)?;
predicate : comparison_pred | regex_pred;
comparison_pred : property_id (comparison_operator any_val)?;
```

```
regex_pred : property_id regex_operator regex_expression;
regex_operator : '=~';
```

Listing 3.3: WHERE-Teil der PQL-Grammatik

Listing 3.4 zeit den Änderungsteil einer PQL-Anfrage. Hier können Änderungsoperationen angegeben werden, die auf die zuvor selektierten Knoten angewandt werden. Das eben genannte Beispiel kann z.B. durch eine Insert-Operation erweitert werden. `MATCH a-[:sequenceFlow]->b WHERE a.id='test'SET INSERTNODE(a, b, task, 'New Node', serial)` fügt einen Knoten seriell zwischen den zuvor selektierten Knoten a und b ein. Der neue Knoten ist vom Typ `task` und wird mit `New Node` bezeichnet.

```
//Änderungsteil der PQL-Grammatik
set : 'SET' operation (',' operation)*;
//Operationen die auf selektierte Prozessmodelle ausgeführt werden können
operation : change_operation;
//Umsetzung der Adaptation Patterns: Insert, Delete und Move
change_operation : insert_node | delete_node | move_node;
//Insert Pattern
insert_node : INSERTNODE'(' ID ',' ID ',' ID ',' SINGLEQUOTESTRING ',' (SERIAL |
    PARALLEL | CONDITIONAL) ')';
//Delete Pattern
delete_node : DELETENODE'(' ID ')';
//Move Pattern
move_node : MOVENODE'(' ID ',' ID ',' ID ',' (SERIAL | PARALLEL | CONDITIONAL) ')';
```

Listing 3.4: SET-Teil der PQL-Grammatik

Listing 3.5 zeigt die Regeln der Grammatik, die auf Terminalsymbole verweisen. Dabei werden Regeln angeben, wie z.B. das =-Symbol in Zeile 14. Zusätzlich sind im unteren Teil der Grammatik Regeln mit dem Schlüsselwort `fragment` gekennzeichnet. Diese Regeln werden von ANTLR zur internen Verarbeitung benötigt [18].

```
//Regeln die auf eine Folge von Buchstaben verweisen
INSERTNODE : 'INSERTNODE';
DELETENODE : 'DELETENODE';
MOVENODE : 'MOVENODE';
SERIAL : 'serial';
PARALLEL : 'parallel';
CONDITIONAL : 'conditional';
AND : 'AND';
OR : 'OR';
```

```
NOT : 'NOT';
ID : (INTEGER|LETTER)+;
TYPE : LETTER+;
ORDERING_FUNCTION : (EQUAL | NOTEQUAL | LESS_THAN_OR_EQUAL | LESS_THAN |
    MORE_THAN_OR_EQUAL | MORE_THAN);
EQUAL : '=';
STRING: '"' (ESC|.)*? '"' ;
SINGLEQUOTESTRING : '\''(ESC|.)*? '\'';
WS        :   [ \t\r\n]+ -> skip ;
fragment INTEGER : ('0'..'9');
fragment LETTER : ('a'..'z'|'A'..'Z');
fragment NOTEQUAL : '!=';
fragment LESS_THAN_OR_EQUAL : '<=';
fragment LESS_THAN : '<';
fragment MORE_THAN_OR_EQUAL : '>=';
fragment MORE_THAN : '>';
fragment ESC : '\\"' | '\\\\' ;
```

Listing 3.5: Terminalsymbole der Grammatik

Übersetzung einer PQL-Anfrage zu einem CqlRequest

Aus der in Kapitel 3.4.1 beschriebenen Grammatik kann mithilfe von ANTLR ein Parser mit Visitor-Methode automatisch generiert werden. Nach der Erstellung des Parsers wird ein Interface bereitgestellt, das die Visitor-Methode implementiert. Die *PqlVisitor*-Klasse erbt dieses Interface und übersetzt eingehende PQL-Anfragen. Für die weitere Verarbeitung der PQL-Anfragen ist es hilfreich, diese in eine Zwischenrepräsentation zu überführen. Diese Zwischenrepräsentation wird in der *CqlRequest*-Klasse dargestellt.

Ein CqlRequest besteht aus der originalen PQL-Anfrage und einer Liste mit *CqlQuery*-Objekten. Damit die Umsetzung von mehreren Pfaden in einem MATCH-Teil umgesetzt werden kann, muss jeder im MATCH-Teil vorkommende Pfad separat betrachtet werden. Dazu wird für jeden Pfad ein CqlQuery-Objekt erstellt und der Liste des CqlRequests hinzugefügt. Ein CqlQuery-Objekt enthält eine Cql-Anfrage, bei der der entsprechende Pfad des MATCH- und der WHERE-Teil der PQL-Anfrage verarbeitet wird. Zusätzlich wird die Änderungsoperation gespeichert, die ausgeführt werden soll.

Grundsätzlich können zwei Typen von PQL-Anfragen unterschieden werden. Zum einen gibt es Anfragen ohne einen SET-Teil und somit ohne Änderungsoperation. Bei dieser Art von Anfrage soll das Repository nach Prozessmodellen durchsucht und dem Nutzer zurückgegeben werden. Zum anderen gibt es Anfragen mit einem SET-Teil und somit mit Änderungsoperation, die ausgeführt werden sollen. Hier muss zusätzlich zur Suche und Rückgabe der Prozessmodelle eine Änderung an den betroffenen Stellen ausgeführt werden. Bei diesen PQL-Anfragen müssen die drei Änderungsoperationen Insert, Delete und Move jeweils separat betrachtet werden. Nachfolgend wird die Erstellung des CqlQuery-Objekts zu den unterschiedlichen Typen von PQL-Anfragen näher erläutert.

Erstellung der CqlQuery-Objekte zu PQL-Anfragen ohne Änderungsoperation

Bei PQL-Anfragen ohne Änderungsoperationen müssen Prozessmodelle lediglich im Repository gesucht und zurückgegeben werden. Dabei wird die Änderungsoperation auf `none` gesetzt. Mit der CQL-Anfrage muss ohne Änderungsoperation nur die ID des Prozessmodells gefunden werden, in der der Pfad aus dem MATCH-Teil enthalten ist. Folglich sollte die CQL-Anfrage die ID des Prozessmodells als Resultat zurückliefern. Als Beispiel wird nach zwei aufeinanderfolgende Knoten gesucht, die mit einem Sequenzfluss miteinander verbunden sind. Zusätzlich soll Knoten `a` die ID `test` besitzen (siehe Listing 3.6).

```
MATCH a-[:sequenceFlow]->b
WHERE a.id='test'
```

Listing 3.6: PQL-Anfrage ohne Änderungsoperation

Listing 3.7 zeigt die resultierende CQL-Anfrage. Dabei werden beide Teile MATCH und WHERE übernommen und zusätzlich vor den Pfad im MATCH-Teil ein Knoten in die Suche mit aufgenommen, der vom Typ `BusinessProcess` ist. Am Ende der CQL-Anfrage wird mit `RETURN` bestimmt, welche Knoten zurückgegeben werden sollen. In diesem Fall werden alle Knoten vom Typ `BusinessProcess` zurückgegeben.

```
MATCH (bpNodes:BusinessProcess)-[*]->(a)-[:sequenceFlow]->(b)
WHERE a.id='test'
```

```
RETURN DISTINCT bpNodes
```

Listing 3.7: CQL-Anfrage zu PQL-Anfrage ohne Änderungsoperation

Erstellung der CqlQuery-Objekte zu PQL-Anfragen mit Änderungsoperation

Bei PQL-Anfragen mit Änderungsoperationen müssen zusätzlich noch IDs von betroffenen Knoten, an denen Änderungen durchgeführt werden, in Erfahrung gebracht werden. Zusätzlich wird bei einer Insert-Operation der Knoten bereits in der CQL-Anfrage erstellt und die ID ebenfalls zurückgegeben. Als Änderungsoperation wird der jeweilige Typ der Operation gesetzt, z.B. ist der Typ einer seriellen Insert-Operation `insert_node_serial`. Listing 3.8 zeigt die von oben erweiterte PQL-Anfrage, um einen neuen Knoten zwischen Knoten a und b einzufügen.

```
MATCH a-[:sequenceFlow]->b
WHERE a.id='test'
SET INSERTNODE(a, b, task, 'New Node', serial)
```

Listing 3.8: PQL-Anfrage mit Insert-Operation

In Listing 3.9 kann die resultierende CQL-Anfrage entnommen werden. Zur Umsetzung einer Insert-Operation werden die IDs des Vorgängers und des Nachfolgers des einzufügenden Knotens benötigt. Zusätzlich wird die generierte ID des einzufügenden Knotens ebenfalls zurückgegeben.

```
MATCH (a)-[:sequenceFlow]->(b)
WHERE a.id='test'
CREATE (newNode:task{id:'newGeneratedId', name:'New Node'})
RETURN a.id AS predId, b.id AS succId, newNode.id AS newNodeId
```

Listing 3.9: PQL-Anfrage mit Insert-Operation

Ähnlich zur Insert-Operation wird die Move-Operation ausgewertet. Es wird ebenfalls die Änderungsoperation entsprechend der Move-Operation gesetzt. Die CQL-Anfrage gibt die IDs von dem zu bewegenden Knoten und den beiden Knoten an der einzufügenden Stelle zurück. An dieser Stelle wird auf ein Beispiel der Move-Operation verzichtet, denn die erzeugte CQL-Anfrage ist ähnlich zu der von der Insert-Operation. Bei dieser Anfrage

wird lediglich auf das Erstellen eines neuen Knoten verzichtet und dafür die ID des zu bewegenden Knotens zurückgegeben.

Die Delete-Operation unterscheidet sich von den zwei anderen Operationen, da hier nur die ID des zu löschenden Knoten gebraucht wird. Die Änderungsoperation wird wie gewohnt auf die Operation gesetzt.

3.4.2 CQLTransformer und Implementierung der Änderungsoperationen

In diesem Abschnitt wird die Transformation eines CqlRequests beschrieben. Das Ziel dieser Transformation ist die Erstellung von CQL-Anfragen, die die Operationen einer PQL-Anfrage ausführen. Operationen sind im speziellen: die Suche nach Prozessmodellen in der Graphendatenbank und die Ausführung der Adaptation Patterns Insert, Move und Delete, die in Kapitel 2.1.3 beschrieben wurden. Dafür wird die in einem CqlRequest-Objekt gespeicherte CqlQuery-Liste ausgewertet. Die Auswertung des CqlRequests erfolgt in der *CqlRequestController*-Klasse, bei der durch einen Aufruf der Methode `executeCqlQueryBySetOperation( CqlRequest cqlRequest )` je nach Änderungsoperation die weitere Ausführung bestimmt wird. Bevor eine Änderungsoperation jedoch ausgeführt werden kann, muss geprüft werden, ob die syntaktische Korrektheit eines Prozessmodells nach dessen Ausführung bestehen bleibt. Dazu wird nachfolgend für jede Änderungsoperation gezeigt, wie diese durchgeführt werden und wann diese ausgeführt werden dürfen (vgl. [1]). Zudem wird beispielhaft die Umsetzung der CQL-Anfragen einer seriellen Insert-Operation vorgestellt.

Implementierung der Insert-Operation

Die Insert-Operation kann in drei verschiedenen Varianten durchgeführt werden: das Einfügen eines Knoten seriell, parallel und konditional. Nachfolgend wird zunächst das Einfügen seriell erklärt und anschließend die Umsetzung der parallelen und konditionalen Insert-Operation näher erläutert.

Beim seriellen Einfügen von Knoten wird ein neuer Knoten zwischen zwei direkt aufeinanderfolgenden Knoten eingefügt. Dabei muss vorab geprüft werden, ob das Einfügen

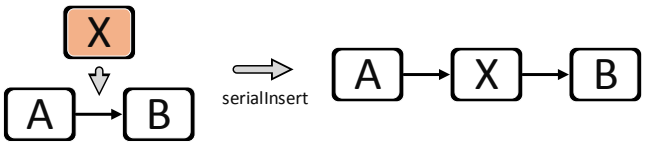

Abbildung 3.9: Serielle Insert-Operation (Quelle: nach [1])

korrekt durchgeführt werden kann. Dies geschieht, indem geprüft wird, ob die zwei Knoten tatsächlich direkt aufeinander folgen. Somit können hier zwei Möglichkeiten auftreten. Bei der ersten Möglichkeit wird ein Knoten zwischen zwei bestehende Knoten eingefügt, ohne dass weitere Knoten dazwischen liegen. Wenn dies der Fall ist, kann ohne weitere Prüfung das Einfügen durchgeführt werden (siehe Abbildung 3.9). Bei der zweiten Möglichkeit liegen zwischen den beiden bestehenden Knoten noch weitere Knoten. In diesem Fall kann der Knoten ohne zusätzliche Informationen nicht seriell eingefügt und die Änderungsoperation somit nicht durchgeführt werden.

Listing 3.10 zeigt die resultierenden CQL-Anfragen einer seriellen Insert-Operation. Zunächst wird die Kante, die vom Vorgänger auf den Nachfolger zeigt, auf den neuen Knoten umgesetzt. Dazu muss die Kante, mit den Informationen der alten, neu erstellt werden, da Neo4j die Änderung der Start und Endpunkte einer Kante nicht erlaubt. Anschließend wird eine neue Kante vom neuen Knoten zum Nachfolger erstellt. Danach wird die alte Kante zwischen Vorgänger und Nachfolger gelöscht. Zum Schluss muss der neue Knoten dem entsprechenden Pool hinzugefügt werden.

```
//Erstelle bestehende Kante zwischen Vorgänger und einzufügenden Knoten (Aufruf:
    createExistingRelationship(predId, newNodeId, oldRel))
MATCH (s), (r)
WHERE s.id='sourceNodeId' AND r.id='refNodeId'
CREATE (s)-[:relationshipType{id:'relationshipId'}]->(r)
//Erstelle neue Kante zwischen neuem Knoten und Nachfolger (Aufruf:
    createNewRelationship(newNodeId, succId, "sequenceFlow"))
MATCH (n1),(n2)
WHERE n1.id='sourceNodeId' AND n2.id='refNodeId'
CREATE (n1)-[r:relType{id:'newId'}]->(n2)
//Lösche alte Kante zwischen Vorgänger und Nachfolger (Aufruf:
    deleteRelationshipByPredIdAndSuccId(predId, succId))
MATCH (n1)-[r]->(n2)
WHERE n1.id='sourceNodeId' AND n2.id='refNodeId'
```

```
12 DELETE r
   //Erstelle flowNodeRef zur Zuordnung im Pool für neuen Knoten (Aufruf:
       createFlowNodeRefToNode(predId, newNodeId))
14 MATCH (l:lane)-[:flowNodeRef]->(pred)-->(newNode)
   WHERE newNode.id='newNodeId' AND pred.id='predNodeId'
16 CREATE (l)-[:flowNodeRef]->(newNode)
```

Listing 3.10: CQl-Anfragen zur Ausführung einer seriellen Insert-Operation

Die beiden Operationen paralleles und konditionales Einfügen können zusammengefasst werden, denn hierbei spielt jeweils nur der Typ des Gateways (AND oder XOR) eine Rolle. Bei diesen beiden Typen von Insert-Operationen wird zusätzlich zum Knoten ein Split- und ein Join-Gateway eingefügt, wobei der neue Knoten als Zweig des Gateways eingefügt wird. Dabei bedeutet parallele Insert-Operation eine Parallelisierung und somit das gleichzeitige Ausführen von Elementen und eine konditionale Insert-Operation das bedingte Ausführen von Elementen und somit eine Entscheidung zwischen mehreren Zweigen eines Gateways.

Es kann wiederum zwischen zwei Fällen unterschieden werden, wie die Insert-Operation ausgeführt werden soll. Beim ersten Fall handelt es sich um das Einfügen zwischen zwei Knoten, wobei eine beliebige Anzahl von Knoten zwischen den einzufügenden Knoten liegen kann. Bei diesem Fall müssen alle Knoten, die zwischen den beiden Knoten liegen, zusammengefasst werden und als ein Zweig in dem jeweils einzufügenden Gatewaytyp dargestellt werden. Anschließend kann der neue Knoten als ein weiterer Zweig des Gateways eingefügt werden. Es muss weiter geprüft werden, ob einer der dazwischenliegenden Knoten ein Gateway ist. Hat in diesem Fall dieses Gateway kein passendes Join- oder Split-Gateway innerhalb der zwei Knoten, zwischen denen eingefügt werden soll, kann die Insert-Operation nicht weiter ausgeführt werden, denn dadurch würde die Struktur des Prozessmodells verletzt werden. Falls kein Gateway-Knoten zwischen den bestehenden Knoten liegt, kann auch hier der neue Knoten mit den entsprechenden Gateways ohne weitere Probleme eingefügt werden (siehe Abbildung 3.10, Variante 1). Beim zweiten Fall wird versucht, ein Knoten entweder parallel oder konditional in ein bestehendes Gateway einzufügen. Dabei wird ein Gateway vom einzufügenden Typ als Zweig des bestehenden Gateways eingefügt. Das neu eingefügte Gateway hat dabei zwei Zweige, zum einen den Knoten, der eingefügt werden soll, und zum anderen

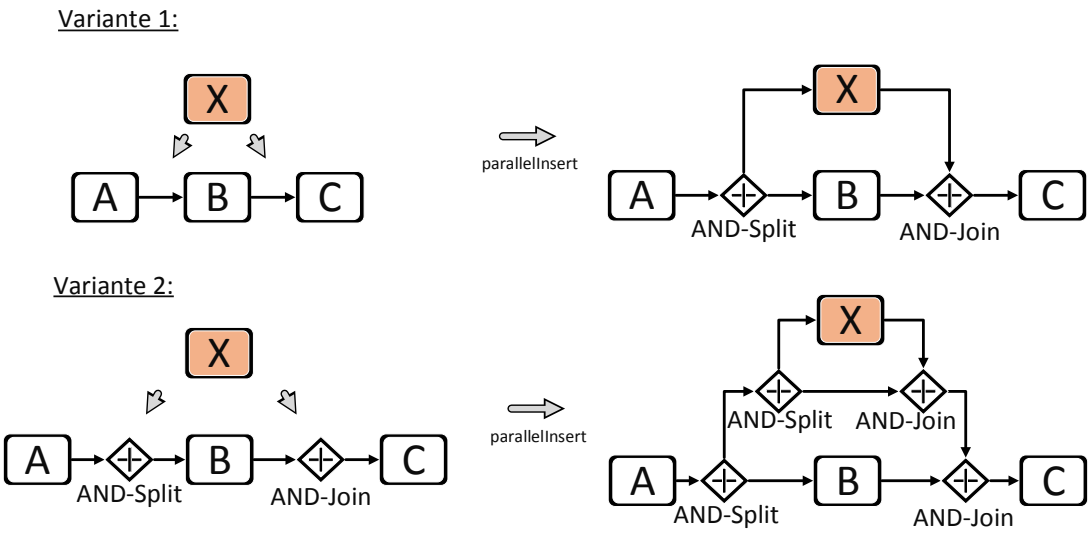

Abbildung 3.10: Parallele Insert-Operation (Quelle: nach [1])

einen leeren Zweig (siehe Abbildung 3.10, Variante 2). Nachfolgend wird beispielhaft die parallele Insert-Operation vorgestellt. Die konditionalen Insert-Operation ist dazu analog. Es muss anstatt einem AND-Gateway ein XOR-Gateway eingefügt werden.

Die Umsetzung einer parallelen Insert-Operation kann durch folgende Operationen durchgeführt werden. Zuerst wird die ausgehende Kante des Vorgängers und die eingehende Kante des Nachfolgers geladen und gespeichert. Danach wird die ID des Knotens nach dem Vorgänger abgefragt und die Kante zu diesem gelöscht. Anschließend wird ein Split- und ein Join-Gateway erstellt. Der Vorgänger wird dann mit dem Split-Gateway verbunden, ebenso das Join-Gateway mit dem Nachfolger. Darauf folgend wird der neue Knoten als Zweig im Gateway eingefügt. Dazu wird eine neue Kante zwischen Split-Gateway zum neuen Knoten und vom neuen Knoten zum Join-Gateway erstellt. Danach müssen noch die Knoten, die zwischen Vorgänger und Nachfolger lagen, als ein weiterer Zweig im Gateway eingefügt werden. Falls sich keine Elemente dazwischen befanden, wird ein leerer Zweig eingefügt. Zum Schluss muss die Zuordnung für alle neu erzeugten Elemente (Gateways und neuer Knoten) zum Pool erfolgen. Somit ist die parallele Insert-Operation abgeschlossen.

Abbildung 3.11: Delete-Operation (Quelle: nach [1])

Implementierung der Delete-Operation

Mit der Delete-Operation können Knoten von Prozessmodellen gelöscht werden. Im Vergleich zur Insert-Operation muss bei der Delete-Operation überprüft werden, um welches Element es sich bei dem zu löschenden Knoten handelt. Dabei muss bei den folgenden Knotentypen eine weitere Prüfung vorgenommen werden: `startEvent`, `endEvent`, `gateway`-Knoten und Knoten mit ausgehenden und eingehenden Nachrichtenflüssen. Wenn die Delete-Operation auf einen Knoten ausgeführt werden soll, der nicht unter den eben genannten ist, kann die Operation ohne weitere Prüfung durchgeführt werden (siehe Abbildung 3.11).

Bei der Umsetzung der Delete-Operation wird zunächst die Kante vom Vorgänger zum Knoten, der gelöscht werden soll, geladen. Danach wird der Knoten gelöscht und damit auch sämtliche eingehende und ausgehende Kanten des Knotens. Nach der Löschung wird eine neue Kante vom Vorgänger zum Nachfolger mit den Informationen der zuvor geladenen Kante erstellt.

Implementierung der Move-Operation

Die Move-Operation wird als Kombination der beiden Operationen Insert und Delete implementiert. Hierbei wird eine Insert-Operation mit den Informationen des zu bewegenden Knoten an der Stelle ausgeführt, an die der Knoten bewegt werden soll. Der zu bewegende Knoten wird zwischen den einzufügenden Knoten als Abbild des alten Knoten neu erstellt. Danach wird der bestehende Knoten mit einer Delete-Operation gelöscht. Da diese Operation als Kombination der beiden anderen ausgeführt wird, muss für beide Operationen festgestellt werden, ob eine Move-Operation durchgeführt werden kann.

3.4.3 Datenbankschnittstelle

In diesem Abschnitt wird die Schnittstelle zur Neo4j-Datenbank betrachtet. Zunächst wird erläutert, wie eine Verbindung zur Datenbank aufgebaut und verwaltet wird. Nach dem Aufbau einer Verbindung können CQL-Anfragen auf der Datenbank ausgeführt werden.

Aufbau der Verbindung

Die Datenbankverbindung wird mithilfe des *Singleton-Pattern* geregelt, sodass zur einer beliebigen Zeit maximal eine Instanz der Datenbankverbindung existiert. Damit ist es möglich, die Anzahl der Verbindungen zur Datenbank zu regeln. Auf die Neo4j-Datenbank kann maximal eine Verbindung schreibend zur gleichen Zeit zugreifen.

Um eine Verbindung zur Datenbank herzustellen, wird die Methode `getInstance()` in der *DatabaseConnector*-Klasse aufgerufen. Nach dem Aufruf wird geprüft, ob bereits ein Objekt instanziert wurde. Wenn keine Instanz existiert, wird eine neue erstellt und diese zurückgegeben. Ansonsten wird die bereits existierende zurückgegeben. Mit diesem Objekt können nun Operationen auf der Datenbank ausgeführt werden.

Die Verbindung zur Datenbank wird durch die *DatabaseController*-Klasse geregelt. Bei jeder Aktion, die auf der Datenbank ausgeführt werden soll, wird geprüft, ob eine offene Verbindung zur Datenbank besteht. Dabei handelt es sich um eine zweistufige Kontrolle. Bei jeder Stufe wird entweder eine neue Verbindung erstellt oder der Timer zum Schließen der Datenbankverbindung erneuert. Zuerst wird überprüft, ob bereits ein Verbindungsobjekt existiert. Danach wird mit `graphDb.isAvailable()` getestet, ob eine Verbindung über dieses Objekt verfügbar ist. Falls bereits eine Verbindung offen ist, wird lediglich der Timer erneuert. Andernfalls wird eine neue Verbindung geöffnet und bereitgestellt. Beim eigentlichen Öffnen der Verbindung zur Datenbank wird mit `new GraphDatabaseFactory().newEmbeddedDatabase()` eine eingebettete Datenbank von Neo4j gestartet. Zusätzlich wird mit `registerShutdownHook()` ein ShutdownHook hinzugefügt, damit die Verbindung in jedem Fall sauber getrennt wird. Nach dem Öffnen wird zusätzlich eine *TimerTask* gestartet, die die Verbindung nach 30 Sekunden ohne Benutzung

wieder schließt. Eine TimerTask führt nach Ablaufen einer vorgegebenen Zeitperiode eine Aktion aus. In diesem Fall wird die Verbindung zu einer Datenbank geschlossen.

Ausführung von CQL-Anfragen

In diesem Abschnitt wird die Ausführung von CQL-Anfragen beschrieben. Es können grundsätzlich zwei unterschiedliche Arten von Anfragen betrachtet werden. Zum einen gibt es Anfragen, bei denen auf der Datenbank etwas ausgeführt wird und kein Rückgabewert erwartet wird. Zum anderen gibt es Anfragen, bei denen durch die Anfrage eine Rückgabe von Werten erwartet wird.

In Neo4j müssen alle Aktionen, die auf der Datenbank ausgeführt werden, in eine Transaktion verpackt werden. Um eine Anfrage auf der Datenbank auszuführen, muss folgende Reihenfolge von Aktionen beachtet werden.

- Starte Transaktion
- Führe Anfrage auf der Datenbank aus
- Markiere Transaktion als erfolgreich bzw. fehlgeschlagen
- Beende Transaktion

Bei Anfragen ohne Rückgabewert kann eine generische Methode zur Ausführung geschrieben werden, bei der die Anfrage selber in einem String übergeben wird. Zuerst wird die Verbindung zur Datenbank geprüft und anschließend – wie oben beschrieben – eine Transaktion gestartet. Danach wird die Anfrage ausgeführt. Da kein Rückgabewert erwartet wird, kann das Resultat geschlossen und die Transaktion auf erfolgreich gesetzt werden. Listing 3.11 zeigt die generische Methode zur Ausführung von Anfragen ohne Rückgabewert.

```
/**
* Generische Methode zum Ausführen von CQL-Anfragen.
*
*/
public void executeCqlQueryWithoutResult(String query){
        //Test Verbindung zur Datenbank
        checkDatabaseConnection();
```

```
8         //Beginne eine Transaktion und versuche die Anfrage auf der Datenbank
              auszuführen
          try (Transaction tx = graphDb.beginTx();
10                Result result = graphDb.execute(query)) {
                  //Schlie ß e Resultat, da kein Rückgabewert erwartet wird
12                result.close();
          //Setze Transaktion auf erfolgreich
14        tx.success();
          }
16 }
```

Listing 3.11: Generische Methode zur Ausführung von Anfragen ohne Rückgabewert

Bei Anfragen mit Rückgabewert kann keine generische Methode geschrieben werden, da man je nach Anfrage ein anderes Resultat erhält. Um diese Art von Anfrage abzudecken, werden vordefinierte Methoden bereitgestellt. Hierunter fallen vor allem die Anwendung der Änderungsoperationen, die in Kapitel 3.4.2 vorgestellt wurden.

Der Ablauf einer solchen Anfrage entspricht dem der generischen Methode, jedoch muss, nachdem die Anfrage ausgeführt wurde, das Resultat der Anfrage verarbeitet werden. Es wird zuerst geprüft, ob eine Datenbankverbindung offen ist. Danach wird eine vordefinierte Anfrage ausgeführt. Beispielsweise können alle Knoten geladen werden, die das Label `BusinessProcess` besitzen und anschließend sollen die Namen der Knoten als Liste zurückgegeben werden. Das Resultat wird sukzessive durchgegangen und die Namen werden in einer Liste gespeichert, welche dann am Ende an die aufrufende Methode zurückgegeben wird. Zum Schluss muss das Resultat geschlossen und die Transaktion als erfolgreich markiert werden.

3.4.4 Import von Prozessmodellen

In diesem Abschnitt wird der Import von Prozessmodellen in das PQLParser Framework beschrieben. Es wird im Folgenden davon ausgegangen, dass das Prozessmodell im XML-Format zur Verfügung steht. Die Übertragung von Prozessmodellen mittels XML-Dokumenten bietet einige Vorteile. Durch die Wohlgeformtheit kann davon ausgegangen werden, dass übergebene XML-Dokumente einer gewissen Struktur entsprechen. Zudem sind sie als text-basiertes Format plattformunabhängig [11].

Der erste Schritt beim Import ist das Parsen der XML-Datei, um das Prozessmodell in eine interne Repräsentation überzuführen. Als nächstes wird die interne Repräsentation in CQL-Anfragen übersetzt und auf der Datenbank ausgeführt. Die CQL-Anfragen erstellen ein Abbild des Prozessmodells auf der Datenbank.

Interne Repräsentation von Prozessmodellen

Um ein Prozessmodell in eine Datenbank importieren zu können, muss das entsprechende Prozessmodell in einer internen Repräsentation dargestellt werden. Das erleichtert sowohl den Import und die Überführung in die Datenbank als auch den Export und das Herauslesen aus der Datenbank. Die folgende Struktur soll ein Prozessmodell intern beschreiben. Die *BusinessProcess*-Klasse bildet dabei das Prozessmodell ab. Diese besteht aus einer eindeutigen ID, einer Liste von Pools, einer Liste von Verbindungen und einer Liste von Attributen. Mit der *BusinessProcessPool*-Klasse können Pools dargestellt werden. Diese bestehen aus einer ID, einer Liste von Knoten und einer Liste von Attributen. Die Klassen *BusinessProcessNode* und *BusinessProcessRelationship* stellen Lanes, Aktivitäten, Ereignisse, Sequenzflüsse und Nachrichtenflüsse dar. Dabei haben BusinessProcessNodes einen Typ (z.B. `exclusiveGateway`), eine ID und eine Liste von Attributen. BusinessProcessRelationships haben ebenfalls einen Typ, eine ID, eine Liste von Attributen und zusätzlich jeweils eine Referenz zum Start- und Endknoten. Mit der *BusinessProcessProperty*-Klasse werden alle Attribute der Elemente beschrieben. BusinessProcessPropertys bestehen dabei aus Schlüssel-Wert-Paaren.

Ablauf des Imports im PQLParser Framework

Für den Import von Prozessmodellen sind die Klassen *BpmnXmlImportExportController* und *BpmnXmlParser* zuständig. Der Importprozess wird mit dem Aufruf der `bpmnXmlImport ( String fileName )`-Methode und der Übergabe des Dateinamen des Prozessmodells gestartet. Nach der Übergabe wird die XML-Datei geparst und in die interne Repräsentation übergeführt.

Das Parsen der XML-Datei übernimmt ein StAX-Parser [34]. Dabei wird das Dokument sukzessive durchlaufen und für jedes Element ein, der internen Repräsentation entsprechendes, Java-Objekt erzeugt. Nachfolgend wird beispielhaft gezeigt, wie eine Aktivität in die interne Repräsentation übergeführt wird. Nachdem der StAX-Parser den Start eines Elementes erkennt, wird ein BusinessProcessNode-Objekt erstellt. Der `type` dieses Objekts wird jeweils auf den Namen und die `id` auf die ID des Elements aus der XML-Datei gesetzt. Anschließend wird die BusinessProcessNode in die Liste des dazugehörigen Pools hinzugefügt. Der BusinessProcess wird somit Schritt für Schritt erstellt und anschließend zurückgegeben. Nach dem Parsen ist das Prozessmodell in einem BusinessProcess-Objekt dargestellt und kann weiter verarbeitet werden.

Anschließend werden aus der internen Repräsentation eine Reihe von CQL-Anfragen erstellt, indem der CqlRequestController-Klasse das geparste Prozessmodell übergeben wird. Die resultierenden CQL-Anfragen können dann auf der Datenbank ausgeführt werden. Im nächsten Abschnitt wird gezeigt, wie die interne Repräsentation in CQL-Anfragen übersetzt wird.

Übersetzung der internen Repräsentation in CQL-Anfragen

Die folgenden Operationen werden benötigt, um die Abbildung der internen Repräsentation auf die Datenbank zu ermöglichen.

- Erstelle Knoten für BusinessProcess-Objekt
- Erstelle Pool- und Lane-Knoten
- Erstelle Knoten für Elemente
- Erstelle Kanten, die die Knoten verbinden

Der BusinessProcess-Knoten referenziert dabei auf die Pool-Knoten mit einer Kante vom Typ `participant`. Lane-Knoten werden wiederum von den Pool-Knoten mit einer `laneSet`-Kante referenziert. Die Zuteilung der Element-Knoten erfolgt mit einer `flowNodeRef`-Kante. Listing 3.12 zeigt beispielhafte CQL-Anfragen, die zur Abbildung eines Prozessmodells auf einer Datenbank benötigt werden. Zeile 2 zeigt die Erstellung eines BusinessProcess-

Knoten, Zeile 4 zeigt die Erstellung von BusinessProcessNodes und Zeile 6-8 zeigt die Erstellung der Kanten der Knoten.

```
//Erstellung von einem BusinessProcess-Knoten
CREATE (n:BusinessProcess{id:'businessProcessId', propertyName:'value', ...})
//Erstellung einer BusinessProcessNode
CREATE (n:nodeType{id:'nodeId', propertyName:'value',...})
//Erstellung einer Verbindung von zwei Knoten
MATCH (n1),(n2)
WHERE n1.id ='sourceNodeId' AND n2.id = 'refNodeId'
CREATE (n1)-[r:relationshipType{id:'relationshipId'}]->(n2)
```

Listing 3.12: CQL-Anfragen zur Erstellung eines Prozessmodells

3.4.5 Export von Prozessmodellen

In diesem Abschnitt wird der Export aus dem PQLParser Framework vorgestellt. Das Prozessmodell wird zunächst aus der Datenbank geladen und in die interne Repräsentation überführt. Anschließend wird die interne Repräsentation in eine XML-Datei geschrieben.

Ablauf des Export im PQLParser Framework

Zuständig für den Export von Prozessmodellen sind die Klassen BpmnXmlImportExportController und *BpmnXmlSerializer*. Der Exportprozess beginnt mit dem Aufruf der `exportBusinessProcessByName( String processName )`-Methode, indem der Name des Prozesses übergeben wird, der exportiert werden soll. Die Operationen die ausgeführt werden müssen, um das Prozessmodell in die internen Repräsentation zu überführen werden nachfolgend dargestellt.

- Suche ID des Prozessmodells anhand des Prozessnamens
- Erstelle BusinessProcess-Objekt mit der ID
- Füge Attribute des Prozessmodells hinzu
- Füge Pools und Lanes dem Prozessmodell hinzu

- Füge Knoten dem Prozessmodell hinzu
- Füge Kanten dem Prozessmodell hinzu

In einem nächsten Schritt wird das Prozessmodell in eine XML-Datei geschrieben. Diese Datei muss dem XML-Format von BPMN-Prozessmodellen entsprechen, welches in Kapitel 2.2.1 beschrieben wurde. Zum Schreiben in eine XML-Datei wird wiederum der StAX-Parser verwendet. Dabei wird zunächst eine XML-Datei mit dem Namen des Prozesses erzeugt. Anschließend wird diese mit der Angabe der XML-Version gestartet und das <definitions/>-Feld erstellt, in welches die Attribute, die im Prozess gespeichert sind, hineingeschrieben werden. Daraufhin werden die Pools sowie die dazugehörigen Lanes und Nodes in das Dokument geschrieben. In einem letzten Schritt werden die nötigen grafischen Elemente der XML-Datei geschrieben und das Dokument kann abgeschlossen und dem Benutzer zur Verfügung gestellt werden.

4

Verwandte Arbeiten

In diesem Kapitel werden verwandte Arbeiten vorgestellt und die Unterschiede zu dem in dieser Arbeit vorgestellten Framework herausgearbeitet.

Eine Alternative zur Speicherung von Daten in einem Graphen bietet die *HyperGraphDB* [35]. Bei dieser Graphendatenbank kommt ein erweitertes *Hyphergraph-Modell* zum Einsatz. Dieses Modell erlaubt nicht nur Verbindungen von Knoten zu Knoten, sondern auch Verbindungen von Kanten zu anderen Kanten bzw. von Kanten zu Knoten. Zudem sind Verbindungen möglich, die mehr als zwei Knoten verbinden. Dadurch können solche Netzwerke dargestellt werden, wie sie in Forschungsarbeiten der Künstlichen Intelligenz oder der Bioinformatik verwendet werden [36].

Ein weiterer Vertreter der Graphendatenbanken ist AllegroGraph [37]. AllegroGraph speichert die Daten im Format von *Resource Description Framework (RDF)*. Mit RDF können Informationen in der Form von Subjekt, Prädikat und Objekt gespeichert wer-

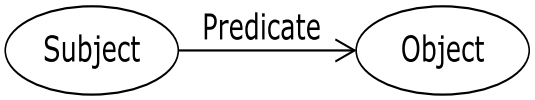

Abbildung 4.1: RDF-Graph mit zwei Knoten und Verbindung (Quelle: [38])

den (siehe Abbildung 4.1). RDF ist eine von W3C standardisierte Repräsentation zur Speicherung von Informationen im Web [38].

Neo4j basiert im Vergleich zu den beiden oben vorgestellten Graphendatenbanken auf dem Property-Graph-Model. Dieses Model eignet sich insbesondere für die Abbildung von Prozessmodellen. Elemente wie Aktivitäten, Ereignisse und Gateways können durch Knoten abgebildet und mit einem Label versehen werden. Verbindende Objekte wie Sequenzflüsse und Nachrichtenflüsse werden durch Kanten dargestellt, die ebenfalls mit einem Label versehen werden können. Zusätzlich bieten Attribute die Möglichkeit weitere Optionen von BPMN-Elementen in der Datenbank abzubilden.

Das Hyphergraph-Modell bietet dafür eine zu große Palette an Möglichkeiten, die bei der Abbildung nicht benötigt werden. Bei RDF dagegen fehlt die Möglichkeit Elemente mit Attributen zu versehen, um z.B. die Art eines Gateways darzustellen (z.B. Split- oder Join-Gateway). Zudem können Graphen von Neo4j mit CQL angesprochen werden. PQL ist angelehnt an CQL, wodurch die Möglichkeit besteht, PQL leicht auf CQL abzubilden.

Eine alternative Methode, um Abfragen auf Prozessmodellen zu definieren, bietet BPMN-Q [40]. Hier werden die Anfragen zunächst visuell in einer ähnlichen Notation wie BPMN definiert und anschließend übersetzt. Mit den übersetzten Anfragen werden dann die Prozessmodelle selektiert [39]. Mithilfe von BPMN-Q ist bereits ein Framework zur Umsetzung von Änderungen an Prozessmodellen entwickelt worden [39]. Aus Abbildung 4.2 kann die Architektur dieses Frameworks entnommen werden. Dabei wird zur Datenhaltung eine relationale Datenbank genutzt. Die grafischen Elemente eines Prozessmodells müssen dafür in Tabellen transformiert werden. Mit BPMN-Q können dann grafisch Anfragen auf das Repository gestellt und die passenden Prozessmodelle selektiert werden.

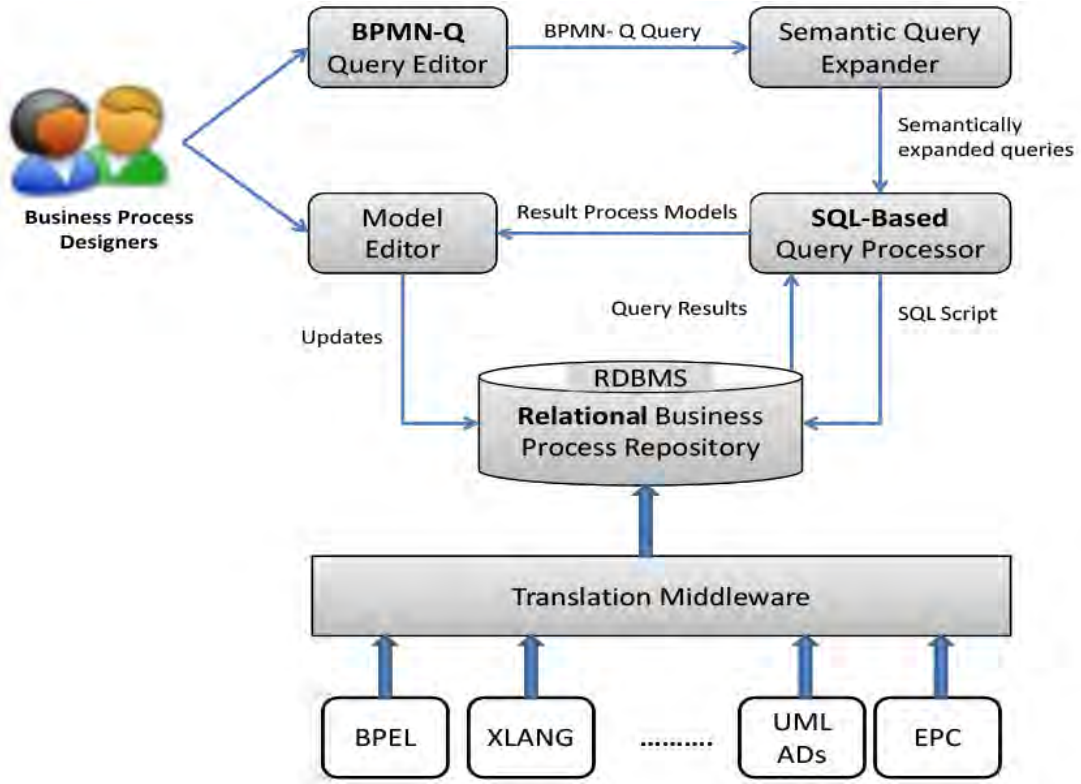

Abbildung 4.2: BPMN-Q Framework-Architektur (Quelle: [39])

Diese werden danach dem Nutzer zur Bearbeitung zur Verfügung gestellt. Anschließend können sie in einem Editor bearbeitet und wieder in das Repository gespeichert werden. Um die Prozessmodelle zu selektieren, muss BPMN-Q in SQL [20] übersetzt werden, damit die relationale Datenbank angesprochen werden kann. Eine einfache Übersetzung der Anfragen ist jedoch nicht möglich, da dies sehr komplexe und große SQL-Anfragen zur Folge hätte. Deshalb wird anhand von Statistiken über die Prozessmodelle im Repository und einem Schwellenwert die Größe der Anfrageergebnisse bestimmt.

Eine weitere Methode, um Anfragen auf Prozessmodellen zu definieren, ist BP-QL [41]. Bei BP-QL werden grafisch mithilfe von Patterns Passagen eines Prozessmodells beschrieben. Die Patterns werden auf die Elemente und Verbindungen des Prozessmodells abgebildet und anhand dessen die Ergebnisse einer Anfrage in BP-QL ausgewertet.

Keine der beiden genannten Anfragesprachen, weder BPMN-Q noch BP-QL, sehen Änderungen und Abstraktionen auf selektierten Prozessmodellen vor [25]. Da das PQLParser Framework Änderungen an Prozessmodellen durchführen soll, ist PQL als Anfragesprache besser geeignet. Zusätzlich ist PQL menschenlesbar und kann leicht erweitert werden, indem entsprechende Passagen der PQL-Grammatik angepasst werden.

Dadurch dass im PQLParser Framework eine Graphendatenbank als Repository verwendet wird und sowohl PQL als auch CQL auf Graphen optimierte Anfragesprachen sind, sind die Probleme der Umwandlung von Prozessmodellen auf Tabellen nicht gegeben. PQL ist darauf ausgerichtet, Änderungen an Modellen auszuführen. Daher ist ein separater Editor, um Änderung auf selektierten Prozessmodellen durchzuführen, nicht nötig.

5

Zusammenfassung und Ausblick

Durch die Erfassung aller im Unternehmenskontext ausgeführten Prozesse entsteht eine Vielzahl an Prozessmodellen, die verwaltet und an denen stetig Änderungen durchgeführt werden müssen. Allerdings ist die händische Durchführung von Änderungen an Prozessmodellen eine komplexe Aufgabe. Aus diesem Grund wurde das PQLParser Framework entwickelt, das als Prototyp im Rahmen dieser Arbeit vorgestellt wurde.

Mit diesem Framework wird ermöglicht, einzelne Prozessmodelle aus einer Menge von Prozessmodellen in einem PAIS zu separieren und in eine gemeinsame Datenbank zu laden. Die Process Query Language erlaubt es, textuell Prozessmodelle zu selektieren und Änderungsoperationen darauf auszuführen. Die Änderungsoperationen werden mithilfe der Change Patterns als High-Level-Operationen definiert und garantieren somit die strukturelle Korrektheit von Prozessmodellen bei deren Ausführung. Mit dem PQL-Parser Framework können somit gleichzeitig Änderungen an mehreren Prozessmodellen automatisch durchgeführt werden.

Das PQLParser Framework ist in vier Module aufgeteilt: *PQLParser*, *CQLTransformer*, *Datenbankschnittstelle* und *Import/Export-Modul*. Der PQLParser ermöglicht das Einlesen und Interpretieren von PQL-Anfragen. Mit dem CQLTransformer werden PQL-Anfragen umgesetzt und auf der Datenbank zur Ausführung gebracht. Die Datenbankschnittstelle verwaltet die Verbindung zur Neo4j-Datenbank und führt übergebene CQL-Anfragen darauf aus. Das Import/Export-Modul ist dafür zuständig, Prozessmodelle im BPMN-XML-Format einzulesen und dem Benutzer wieder auszugeben.

Das in dieser Arbeit dargelegte PQLPaser Framework ist als prototypische Implementierung zu verstehen. Zukünftige Arbeiten könnten die Funktionen des Frameworks erweitern.

Das PQLParser Framework zeigt, dass gleichzeitig Änderungen an mehreren Prozessmodellen automatisch durchgeführt werden können. Das Framework kann durch dessen Modularität einfach ergänzt werden, indem z.B. der vollständige Funktionsumfang von PQL unterstützt wird. Zusätzlich zu Änderungsoperationen könnten ebenfalls die Reduktion und Aggregation von Prozessmodellen umgesetzt werden. Dies kann durch die Anpassung der PQL-Grammatik und des CQLTransfomer-Modul erfolgen.

Des Weiteren könnte eine Optimierung der CQL-Anfragen vorgenommen werden, um diese noch effizienter auszuführen und Änderungen noch schneller umzusetzen.

Eine weitere Möglichkeit, Benutzer zukünftig bei der Verwendung des PQLParser Frameworks zu unterstützen, bieten vordefinierten Operationen. Diese können beispielsweise in einer Operationsdatenbank gespeichert und einfach wiederverwendet werden.

Literaturverzeichnis

[1] Weber, B., Reichert, M., Rinderle-Ma, S.: Change Patterns and Change Support Features - Enhancing Flexibility in Process-Aware Information Systems. Data and Knowledge Engineering **66** (2008) 438–466

[2] Kargl, U.: Change Management im Business Process Management: BPM initiierte Veränderungsprozesse. Diplomica Verlag (2013)

[3] Freund, J., Rücker, B.: Praxishandbuch BPMN 2.0. Hanser (2010)

[4] Dumas, M., La Rosa, M., Mendling, J., Reijers, H.A.: Fundamentals of Business Process Management. Springer (2013)

[5] Jeston, J., Nelis, J.: Business Process Management. Routledge (2014)

[6] Weske, M.: Business Process Management: Concepts, Languages, Architectures. Springer Science & Business Media (2012)

[7] Reichert, M., Weber, B.: Enabling Flexibility in Process-Aware Information Systems: Challenges, Methods, Technologies. Springer (2012)

[8] Gericke, A., Bayer, F., Kühn, H., Rausch, T., Strobl, R.: Der Lebenszyklus des Prozessmanagements. In: Prozessmanagement für Experten: Impulse für aktuelle und wiederkehrende Themen. Springer, Berlin, Heidelberg (2013)

[9] White, S.A.: Introduction to BPMN. IBM Cooperation **2** (2004)

[10] Rinderle, S., Reichert, M., Dadam, P.: Correctness Criteria for Dynamic Changes in Workflow Systems–a Survey. Data & Knowledge Engineering **50** (2004) 9–34

[11] Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., Yergeau, F.: Extensible Markup Language (XML). World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210 **16** (1998)

[12] Object Management Group: Business Process Model And Notation™ (BPMN™) Version 2.0. `http://www.omg.org/spec/BPMN/2.0/` (2011) [Online; Stand: 17.05.2016].

[13] Van Deursen, A., Klint, P., Visser, J.: Domain-Specific Languages. Centrum voor Wiskunde en Informatika **5** (2000) 12

[14] Wegener, I.: Theoretische Informatik. BG Teubner, Stuttgart **5** (1993) 1

[15] Erk, K., Priese, L.: Theoretische Informatik: Eine umfassende Einführung. Springer (2008)

[16] Schöning, U.: Theoretische Informatik–kurzgefasst. Spektrum (2001)

[17] Hohagen, F.: Parsing: eine Einführung in die maschinelle Analyse natürlicher Sprache. Springer (2013)

[18] Parr, T.: The Definitive ANTLR 4 Reference. 2nd edn. Pragmatic Bookshelf (2013)

[19] Hunger, M.: Neo4j 2.0: Eine Graphdatenbank für alle. Entwickler.press, Frankfurt am Main (2014)

[20] Chamberlin, D.D., Boyce, R.F.: SEQUEL: A Structured English Query Language. In: Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control, ACM (1974) 249–264

[21] Robinson, I., Webber, J., Eifrem, E.: Graph Databases: New Opportunities for Connected Data. O'Reilly (2015)

[22] Kammerer, K.: Enabling Personalized Business Process Modeling: The Clavii BPM Platform. Master's thesis, Ulm University (2014)

[23] Kammerer, K., Kolb, J., Andrews, K., Bueringer, S., Meyer, B., Reichert, M.: User-centric Process Modeling and Enactment: The Clavii BPM Platform. In: Proceedings of the BPM Demo Session 2015 (BPMD 2015), co-located with the 13th International Conference on Business Process Management (BPM 2015). Number 1418 in CEUR Workshop Proceedings, CEUR-WS.org (2015) 135–139

[24] Kolb, J., Kammerer, K., Reichert, M.: Updatable process views for user-centered adaption of large process models. In: 10th Int'l Conference on Service Oriented Computing (ICSOC'12). Number 7636 in LNCS, Springer (2012) 484–498

[25] Kammerer, K., Kolb, J., Reichert, M.: PQL - A Descriptive Language for Querying, Abstracting and Changing Process Models. In: 17 Int'l Working Conference on Business Process Modeling, Development, and Support (BPMDS'15). Number 214 in LNBIP, Springer (2015) 135–150

[26] Weber, B., Reichert, M., Mendling, J., Reijers, H.A.: Refactoring Large Process Model Repositories. Computers in Industry **62** (2011) 467–486

[27] Twitter Inc.: Twitter Search API. `https://dev.twitter.com/rest/public/search` (2016) [Online; Stand: 19.05.2016].

[28] Parr, T., Harwell, S., Fisher, K.: Adaptive LL(*) Parsing: The Power of Dynamic Analysis. In: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications. OOPSLA '14, New York, NY, USA, ACM (2014) 579–598

[29] Miller, J.J.: Graph Database Applications and Concepts with Neo4j. In: Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA. Volume 2324. (2013)

[30] Neo Technology: The Neo4j Developer Manual v3.0. `http://neo4j.com/docs/developer-manual/current/` (2016) [Online; Stand: 19.05.2016].

[31] Schöning, U.: Algorithmik. Spektrum (2001)

[32] Junghanns, M.: Untersuchung zur Eignung von Graphdatenbanksystemen für die Analyse von Informationsnetzwerken. Master's thesis, Leipzig University (2014)

[33] Panzarino, O.: Learning Cypher. Packt Publishing (2014)

[34] Lam, T., Ding, J.J., Liu, J.C., et al.: XML Document Parsing: Operational and Performance Characteristics. Computer (2008) 30–37

[35] Iordanov, B.: HyperGraphDB: A Generalized Graph Database. In: Web-Age Information Management. Springer (2010) 25–36

[36] Angles, R.: A Comparison of Current Graph Database Models. In: Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on, IEEE (2012) 171–177

[37] Franz Inc.: AllegroGraph. `http://franz.com/agraph/allegrograph/` (2016) [Online; Stand: 14.05.2016].

[38] Klyne, G., Carroll, J.J., McBride, B.: RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation **25** (2014)

[39] Sakr, S., Awad, A.: A Framework for Querying Graph-Based Business Process Models. In: Proceedings of the 19th international conference on World wide web, ACM (2010) 1297–1300

[40] Awad, A.: BPMN-Q: A Language to Query Business Processes. In: EMISA. Volume 119. (2007) 115–128

[41] Beeri, C., Eyal, A., Kamenkovich, S., Milo, T.: Querying Business Processes with BP-QL. Information Systems **33** (2008) 477–507

Abbildungsverzeichnis

Tabellenverzeichnis

Name: Kevin Bee Matrikelnummer: 756516

Erklärung

Ich erkläre, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den .....

Kevin Bee