Erweiterung eines nativen XML-Datenbanksystems um die Validierung anhand einer Ontologie

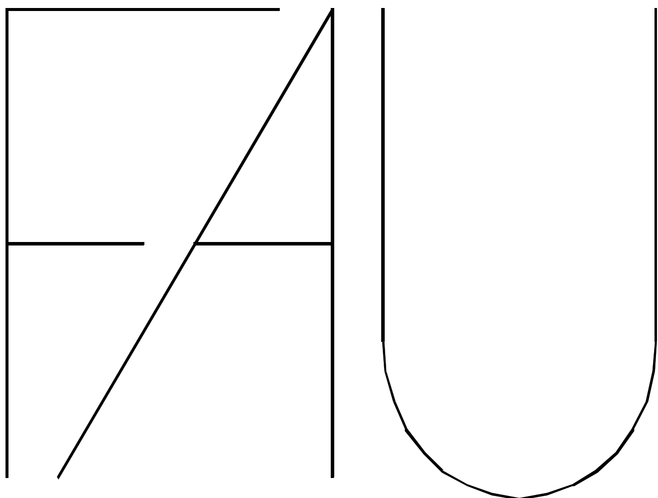
Studienarbeit

Stefan Hanisch

Lehrstuhl für Informatik 6
(Datenmanagement)

Department Informatik
Technische Fakultät

Friedrich-Alexander-
Universität
Erlangen-Nürnberg



Erweiterung eines nativen XML-Datenbanksystems um die Validierung anhand einer Ontologie

Studienarbeit im Fach Informatik

vorgelegt von

Stefan Hanisch

geb. 25.11.1983 in Neustadt a. d. Aisch

angefertigt am

Department Informatik
Lehrstuhl für Informatik 6
Datenmanagement
Friedrich-Alexander-Universität Erlangen–Nürnberg

Betreuer: Prof. Dr. Richard Lenz
Dipl.-Inf. Christoph Neumann

Beginn der Arbeit: 01.04.2009
Abgabe der Arbeit: 23.12.2009

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Universität Erlangen-Nürnberg, vertreten durch die Informatik 6 (Datenmanagement), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Studienarbeit einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 21.12.2009

Stefan Hanisch

Abstract

Upgrading of a native XML database system in order to support consistency checking of ontologies

The importance of ontologies has grown ever since the specification of OWL. One of the main reasons for this development is the fact that OWL ontologies can be serialized using XML syntax. OWL ontologies are used in medicine and biology to represent knowlegde, but nevertheless there are no native XML database systems that provide consistency checking for OWL documents upon insert or update.

The goal of this thesis is the extension of a native XML database system. The upgraded database system supports consistency checking of OWL documents upon insert and update. The first step is to find an appropriate native XML database system. For that reason four open source implemetations of native XML database systems are tested. The tests are intended to find out if the database system supports schema validation of XML documents. After those tests the architecture of the database systems is analysed in order to find out if the validation unit is encapsulated. The encapsulation is interesting because the extensions which allow consistency checking are supposed to be integrated there. The choice falls on the database system that offers the best encapsulation of the validation unit.

Next to the database system an OWL reasoner is needed. Three OWL reasoners are tested regarding their ability to check the consistency of OWL documents. The final choice falls on the OWL reasoner that fits the requirements best.

The functionality of the chosen database system is extended to support XML documents as well as OWL documents and XML schema validation as well as OWL consistency checking. In order to check the consistency of a document the database system has to use the OWL reasoner.

Kurzzusammenfassung

Erweiterung eines nativen XML-Datenbanksystems um die Validierung anhand einer Ontologie

Seit der Spezifikation von OWL sind Ontologien zur Wissensrepräsentation in immer neue Bereiche eingezogen. Das ist darauf zurückzuführen, dass Ontologien mit OWL einheitlich in XML-Syntax dargestellt werden können. Trotz der immer weiteren Verbreitung von OWL, existiert noch kein natives XML-Datenbanksystem, das die Funktion besitzt, solche Dokumente beim Einfügen oder Ändern auf ihre Konsistenz zu prüfen.

Im Rahmen der vorliegenden Arbeit wird ein natives Datenbanksystem so erweitert, dass OWL-Dokumente beim Einfügen und bei Änderungen automatisch auf ihre Konsistenz überprüft werden. Dazu wird zuerst ein geeignetes natives XML-Datenbanksystem gesucht. Zunächst werden vier bestehende Open-Source Implementierungen von nativen XML-Datenbanksystemen untersucht, um festzustellen, ob sie eine Validierung von XML-Dokumenten gegenüber einer, mit einer Schemasprache definierten Grammatik unterstützten. Anschließend wird eine Analyse der Architektur der Datenbanksysteme durchgeführt, die untersucht, inwiefern die Validierungseinheit des Datenbanksystems gekapselt ist, da an dieser Stelle angesetzt werden soll um die Konsistenzprüfung möglichst einfach zu integrieren. Im weiteren Verlauf der Arbeit wird das native XML-Datenbanksystem verwendet, die über die beste Kapselung der Validierungseinheit verfügt.

Nach den XML-Datenbanksystemen werden drei OWL-Reasoner untersucht. Dabei wird ermittelt, ob mit ihnen die Konsistenz von OWL-Dokumenten geprüft werden kann. Verwendet wird schließlich der OWL-Reasoner, der die aufgestellten Anforderungen am besten erfüllt.

Nachdem die Auswahl getroffen ist wird die Funktionalität des nativen XML-Datenbanksystems so erweitert, dass neben XML-Dokumenten auch OWL-Dokumente in dem Datenbanksystem verwaltet und auf ihre Konsistenz überprüft werden können. Zur Prüfung der Konsistenz von OWL-Dokumenten greift das Datenbanksystem dabei auf den OWL-Reasoner zurück.

Inhaltsverzeichnis

Abbildungsverzeichnis ix

Tabellenverzeichnis xi

Listings xiii

1 Einleitung 1

1.1 Motivation 1
1.2 Ziel 2

2 Methodik 3

3 Grundlagen 7

3.1 Extensible Markup Language (XML) 7
3.2 XML Query Language (XQuery) 10
3.3 XML-Datenbanksysteme 12
3.4 Netzwerkschnittstellen 13
3.4.1 Representational State Transfer (REST) 13
3.4.2 XML-RPC 13
3.5 Resource Description Framework (RDF) 14
3.6 OWL-DL-Ontologien 16
3.7 OWL-Reasoner 19
3.8 DIG-Schnittstelle 20
3.9 nRQL-Schnittstelle 21
3.10 Zusammenfassung 22

4 Anforderungsanalyse 23

4.1 Endnutzer-Szenario 23
4.1.1 Ist-Zustand 23
4.1.2 Soll-Zustand 24
4.2 Fachliche Anforderungsanalyse 25
4.2.1 0 x T-Box und 1 x A-Box (0:1) 26

4.2.2 1 x T-Box und 0 x A-Box (1:0) ... 26
4.2.3 1 x T-Box und 1 x A-Box (1:1) ... 27
4.2.4 1 x T-Box und N x A-Box (1:N) ... 28
4.2.5 M x T-Box und N x A-Box ... 28
4.2.6 Überblick ... 29
4.3 Technische Anforderungsanalyse ... 31
4.4 Zusammenfassung ... 32

5 Evaluation verfügbarer Softwarebausteine 33

5.1 OWL-Reasoner ... 33
5.1.1 FaCT++ ... 33
5.1.2 Pellet ... 34
5.1.3 RacerPro ... 34
5.1.4 Übersicht ... 35
5.2 Native XML-Datenbanksysteme ... 36
5.2.1 Apache Xindice ... 36
5.2.2 XML Transaction Coordinator (XTC) ... 37
5.2.3 BaseX ... 37
5.2.4 eXist ... 38
5.2.5 Übersicht ... 39
5.3 Zusammenfassung ... 40

6 Grobanalyse 41

6.1 Schnittstellen ... 41
6.2 Zugriffspfade ... 42
6.2.1 Implizite Validierung ... 42
6.2.2 Explizite Validierung ... 43
6.3 Trigger ... 43
6.4 Zusammenfassung ... 44

7 Grobentwurf 47

7.1 Unterscheidung XML/OWL-Dokumente ... 47
7.2 Konsistenzprüfung ... 48
7.2.1 Implizite Validierung ... 48
7.2.2 Explizite Validierung ... 49
7.3 Verschiedene Übergabemethoden an den OWL-Reasoner ... 49
7.3.1 Dateiübergabe ... 50
7.3.2 Dokumentübergabe über eine Netzwerkverbindung ... 51

7.3.3 Speicherübergabe ... 52
7.3.4 Übersicht ... 53
7.4 OWL-Modul ... 53
7.4.1 Basisentwurf ... 54
7.4.2 Erweiterter Entwurf ... 55
7.4.3 Übersicht ... 57
7.5 Zusammenfassung ... 58

8 Systemanalyse 59
8.1 Implizite Validierung ... 59
8.2 Explizite Validierung ... 64
8.3 Konfiguration ... 72
8.3.1 Konfiguration des Datenbanksystems ... 72
8.3.2 Konfiguration von Collections ... 73
8.4 Lesen eines Dokumentes aus dem Datenbanksystem ... 73
8.5 Trigger ... 75
8.5.1 Konfigurationsphase ... 76
8.5.2 Vorbereitungsphase ... 77
8.5.3 Endphase ... 77
8.6 Zusammenfassung ... 77

9 Systementwurf 79
9.1 OWL-Modul ... 79
9.2 Änderungen an dem Datenbanksystem ... 81
9.3 Erweiterung der Konfiguration für OWL-Validierung ... 82
9.4 ValidationTrigger für XQuery und XUpdate ... 83
9.4.1 ValidationTrigger auf Basis des HistoryTriggers ... 83
9.4.2 Konfiguration des ValidationTriggers ... 86
9.4.3 Funktionsumfang des Triggers ... 86
9.5 Zusammenfassung ... 88

10 Technische Umsetzung 89
10.1 Erweiterung der Konfigurierbarkeit ... 89
10.1.1 Konfigurationsdatei conf.xml ... 89
10.1.2 XML-Schema der Konfigurationsdatei ... 90
10.1.3 Erweiterung der Klasse XMLReaderObjectFactory ... 90
10.1.4 Erweiterung der Klasse Configuration ... 91
10.2 Implementierung des OWL-Moduls ... 92

10.2.1 Die Klasse OWLChecker . . . 93
10.2.2 Die Klasse CustomizeDocument . . . 94
10.2.3 Die Klasse OWLValidation . . . 95
10.2.4 Die Klasse OWLException . . . 98
10.2.5 Klassen für den erweiterten Entwurf . . . 98
10.2.6 Probleme . . . 100
10.3 Integration des OWL-Moduls in eXist . . . 101
10.3.1 Collection . . . 101
10.3.2 Validator . . . 103
10.3.3 Aufgetretene Probleme . . . 104
10.4 ValidationTrigger . . . 104
10.4.1 Vorbereitungsphase des ValidationTriggers . . . 104
10.4.2 Endphase des ValidationTriggers . . . 105
10.4.3 Aufgetretene Probleme . . . 105
10.5 Gegenüberstellung der Validierungsmechanismen . . . 106
10.6 Zusammenfassung . . . 107

11 Evaluation der Umsetzung 109

12 Zusammenfassung der Arbeit 113

13 Ausblick 115

A Dokumentation 117

A.1 eXist Validierung und Anfrageausführung . . . 117
A.2 Testbeschreibung . . . 118
A.2.1 Implizite Validierung . . . 118
A.2.2 ValidationTrigger . . . 119
A.2.3 Explizite Validierung . . . 120
A.2.4 Ergebnis . . . 121

Literaturverzeichnis 123

Abbildungsverzeichnis

3.1 Clara hat die Kinder Eve und Bob 14
3.2 Eine Person ist vom Typ Klasse 15
3.3 Beziehung der Teilsprachen von OWL zu RDF 17

4.1 Ist-Zustand 24
4.2 Soll-Zustand 25
4.3 T-/A-Box Abbildung auf Dokumente 25

6.1 Netzwerkschnittstellen des Datenbanksystems 41
6.2 Zugriffspfade zur Klasse Collection 42
6.3 Zugriffspfade zur Validatorklasse 43

7.1 Entscheidungsbaum zur impliziten Konsistenzprüfung 48
7.2 Übersicht über Übergabemethoden 50
7.3 Dateiübergabe von Dokumenten an den OWL-Reasoner 51
7.4 Dateiübergabe an RacerPro mit jracer 51
7.5 Netzwerkübergabe von Dokumenten 52
7.6 Speicherübergabe von Dokumenten 53
7.7 Einfache Darstellung der Konsistenzprüfung 54
7.8 Erweitertes OWL-Modul 56

8.1 Einfügen über die REST-Schnittstelle 60
8.2 Einfügen über die XML-RPC-Schnittstelle 61
8.3 Einfügen über die XML:DB API 62
8.4 Implizite Validierung in der Klasse Collection 63
8.5 Sequenzdiagramm Validierung über die XML-RPC-Schnittstelle . . . 65
8.6 Sequenzdiagramm Validierung im Embedded-Modus 66
8.7 Sequenzdiagramm Validierung bei XQuery 67
8.8 Sequenzdiagramm der Klasse Validator 69
8.9 Sequenzdiagramme Validator, Grammatik Fall 1 70
8.10 Sequenzdiagramme Validator, Grammatik Fall 2 70
8.11 Sequenzdiagramme Validator, Grammatik Fall 3 71
8.12 Sequenzdiagramme Validator, Grammatik Fall 4 71
8.13 Datei aus dem Datenbanksystem lesen 74
8.14 Klassendiagramm von Serializer 75

8.15 Sequenzdiegramm eines DokumentTriggers 76

9.1 Einfache Darstellung der Konsistenzprüfung 80
9.2 Validator mit OWL-Modul 81
9.3 Collection mit OWL-Modul 82
9.4 Phasenübersicht ValidationTrigger 84
9.5 Sequenzdiagramm: Vorbereitungsphase des ValidationTriggers 84
9.6 Sequenzdiagramm: Endphase des ValidationTriggers 85

10.1 Klassendiagramm der Klasse OWLChecker 93
10.2 Klassendiagramm der Klasse CustomizeDocument 94
10.3 Klassendiagramm der Klasse OWLValidation 95
10.4 Ablauf der Konsistenzprüfung 96
10.5 Klassendiagramm der Klasse OWLException 98
10.6 Zusätzliche Klassen für erweiterten Entwurf 99

Tabellenverzeichnis

5.1 Übersicht über die erfüllten Anforderungen (OWL-Reasoner) 36
5.2 Übersicht über die erfüllten Anforderungen (XML-Datenbanksysteme) 40

9.1 Funktionsumfang des ValidationTriggers 87

10.1 Vergleich der Validierungsmechanismen 106

Listings

3.1 XML-Beispieldatei 8
3.2 Beispiel für ein Element mit Attribut 8
3.3 Beispiel eines XQuery FLWOR-Ausdrucks 11
3.4 Beispiel einer Änderung durch einen XQueryUpdateFacility-Ausdrucks 11
3.5 Beispiel in Tripelstruktur 14
3.6 RDF-Beispiel 15
3.7 RDF-Beispiel 15

6.1 Collection Konfigurationsdatei für einen Trigger 44

8.1 Ausschnitt Konfigurationsdatei 72
8.2 Configuration Konfigurationsdatei 73

9.1 Ausschnitt aus der Konfigurationsdatei 83
9.2 Konfiguration für den ValidationTrigger 86

10.1 Ausschnitt aus der Konfigurationsdatei 89
10.2 Ausschnitt aus dem XML-Schema der Konfigurationsdatei 90
10.3 Ausschnitt aus XMLReaderObjectFactory.java 91
10.4 Ausschnitt aus XMLReaderObjectFactory.java 91
10.5 Veränderungen der Klasse Configuration 92
10.6 Ausführung der OWL-Konsistenzprüfung (Klasse Collection) 102
10.7 Beispiel aus Validator.java 103
10.8 Ausführung der OWL-Konsistenzprüfung (Klasse Validator) 103

A.1 Explizite Validierung ohne Grammatik 117
A.2 Explizite Validierung mit Grammatik 117
A.3 Ausführung einer XQuery Anfrage über die Kommandozeile 117
A.4 Java Client im Textmodus für XQuery Anfragen starten 118
A.5 Ausführung eines XUpdates von der Kommandozeile aus 118

1 Einleitung

1.1 Motivation

In den letzten Jahren haben sich XML-basierte Ontologien für die Wissensrepräsentation immer mehr verbreitet. Dies ist nicht zuletzt auf die Spezifikation von OWL[1] und die Verbreitung des Semantic Web zurückzuführen. Mit Ontologien können einfache Systeme, wie beispielsweise die Länder der Erde klassifiziert und beschrieben werden. Es ist außerdem möglich komplexe Systeme aus der Biologie oder Kunstwerke in Museen zu klassifizieren und zu beschreiben. Im wesentlichen bilden OWL-Ontologien das Wissen auf eine hierarchische Tripelstruktur ab, welche wiederum auf die XML-Syntax abgebildet wird.

Für die Verwaltung solcher hierarchischer Tripelstrukturen stehen bisher so genannte Triplestores zur Verfügung. Mit diesen Triplestores können Anfragen an die enthaltenen Daten gestellt werden, wozu beispielsweise SPARQL genutzt wird. Die Verwendung von Triplestores für Ontologien mit sehr vielen Tripeln ($>10^{12}$)[2] bringt den Nachteil mit sich, dass einerseits das Laden sehr lange dauert und andererseits die Performanz des Systems schlecht mit der Menge der Daten skaliert.

Neben den Triplestores existieren XML-Datenbanksysteme, in denen XML-Dokumente verwaltet werden können. Da OWL nichts anderes ist als eine spezialisierte Form von XML, können diese Datenbanksysteme auch OWL-Dokumente verwalten. Diese Datenbanksysteme bieten, wie Triplestores, die Möglichkeit Daten hineinzuladen und Anfragen an diese Daten zu stellen. Im Gegensatz zu Triplestores sind XML-Datenbanksysteme auch in der Lage XML-Dokumente gegen eine entsprechende Grammatik, auf ihre Validität zu prüfen. Außerdem bieten viele XML-Datenbanksysteme die Möglichkeit die enthaltenen Daten durch Anfragen zu verändern. Ein vorhandenes Datenbanksystem soll so erweitert werden, dass es neben XML-Dokumenten auch OWL-Dokumente auf ihre Konsistenz prüfen kann.

Bisher werden OWL-Dokumente in der Regel mit einem Editor bearbeitet und dann im lokalen Dateisystem gespeichert. Die Prüfung der Konsistenz muss bei fast allen Editoren manuell durchgeführt werden, eine Ausnahme hierfür ist beispielsweise

[1] http://www.w3.org/TR/owl-ref/
[2] Nachzulesen unter: http://esw.w3.org/topic/LargeTripleStores

der OWL-Editor Protégé[3]. Für den Benutzer ist es ein Nachteil, dass selbst wenn die Konsistenzprüfung durch den Editor erfolgt, er anhand des Dokumentes im Dateisystem nicht erkennen kann, ob das Dokument konsistent ist oder nicht. Das hat zur Folge, dass die Konsistenz des Dokumentes vor einer Bearbeitung erneut geprüft werden muss, da ansonsten die Gefahr besteht ein inkonsistentes Dokument zu ändern.

1.2 Ziel

Ziel der Arbeit ist die Erweiterung eines XML-Datenbanksystems um die Konsistenzprüfung von OWL-Dokumenten. Nach dieser Erweiterung soll das Datenbanksystem folgende Funktionen besitzen:

- Das Datenbanksystem führt beim Einfügen von OWL-Dokumenten automatisch eine Konsistenzprüfung durch und fügt die Dokumente nur dann in das Datenbanksystem ein, wenn sie konsistent sind.
- Änderungen an OWL-Dokumenten in dem Datenbanksystem, die durch eine Anfrage (XQuery/XUpdate) ausgelöst werden, werden nur übernommen, wenn das Dokument weiterhin in einem konsistenten Zustand ist.
- Das Datenbanksystem kann zur zentralen Datenhaltung in einem Netzwerk verwendet werden und besitzt eine REST- oder XML-RPC-Schnittstelle.
- Das Datenbanksystem kann über die XML:DB API in andere Anwendungen integriert werden.
- Die ursprüngliche XML-Validierung mit einer Grammatik muss zugänglich bleiben.

Die durchzuführende Erweiterung des Datenbanksystems bringt für den Benutzer eine Reihe von Vorteilen. Einer dieser Vorteile ist, dass die Dokumente, während des Einfügens in das modifizierte XML-Datenbanksystem auf ihre Konsistenz überprüft werden. Die Dokumente werden von dem Datenbanksystem nur dann eingefügt, wenn sie konsistent sind. Dadurch kann der Nutzer sicher sein, dass ein Dokument, welches er aus der Datenbank bekommt, konsistent ist. Außerdem kann die Datenbank von mehreren Nutzern, die auf dieselben Daten zugreifen müssen, zur zentralen Datenspeicherung verwendet werden.

[3]Zu finden unter: http://protege.stanford.edu/

2 Methodik

Vor Beginn der eigentlichen Arbeit war es notwendig, sich in zwei Themengebiete, die verbunden werden sollten, einzuarbeiten. Das erste Themengebiet sind XML-Datenbanksysteme, die dem Bereich der Datenbanksysteme entstammen. Das zweite Themengebiet befasst sich mit OWL-Ontologien und OWL-Reasonern, die aus dem Bereich der künstlichen Intelligenz stammen. Die Ergebnisse werden in Kapitel 3 beschrieben.

Zunächst ist es notwendig eine Anforderungsanalyse durchzuführen. Diese erfolgt in Kapitel 4 und bestimmt die Anforderungen, die aus Endnutzersicht an das neue Datenbanksystem gestellt werden. Außerdem wird untersucht, welche Kombinationen aus T-Box und A-Box möglich sind und wie die einzelnen Boxen auf die in dem Datenbanksystem verwendeten Dokumente abgebildet werden. Abschließend wird eine technische Anforderungsanalyse durchgeführt, die bestimmt, welche Anforderungen das XML-Datenbanksystem und der OWL-Reasoner erfüllen müssen, damit sie für die Verwendung im Rahmen dieser Arbeit in Frage kommen.

Nach der Anforderungsanalyse werden die zur Verfügung stehenden OWL-Reasoner im Abschnitt 5.1 auf ihre Verwendbarkeit im Rahmen der gestellten Anforderungen untersucht. Bei den OWL-Reasonern wird RacerPro ausgewählt, da er eine vollständige Überschneidung mit den Anforderungen besitzt und unsere Universität gute Kontakte zu den Entwicklern hat, wodurch ein besonders guter Support erwartet werden kann. Auf den OWL-Reasoner wird über eine Netzwerkschnittstelle zugegriffen, wodurch der OWL-Reasoner jederzeit gegen einen anderen OWL-Reasoner mit gleicher Netzwerkschnittstelle ersetzt werden kann.

Im nächsten Schritt werden in Kapitel 5.2 einige native XML-Datenbanksysteme dahingehend untersucht, ob sie die in der Anforderungsanalyse aufgestellten Anforderungen erfüllen und somit für die Verwendung in Frage kommen. Zur Detaillierung der Systemanalyse wird in Kapitel 8 eine Untersuchung der Systemarchitektur dieser Datenbanksysteme durchgeführt. Die Analyse konzentriert sich speziell darauf, ob eine Validierungseinheit im jeweiligen Datenbanksystem vorhanden ist. Zusätzlich wird untersucht, wie einfach eine solche Validierungseinheit ausgetauscht werden könnte. Als Ergebnis der Untersuchung wird eXist[1] als Auswahl hervorgehen, da sie am besten

[1] Weitere Informationen unter: http://www.exist-db.org/

mit den Anforderungen korreliert, die an die nativen XML-Datenbanksysteme gestellt werden.

Nach der Auswahl der Systembausteine wird in Kapitel 6 untersucht, welche Anwendungsmöglichkeiten das Datenbanksystem bietet und wie in den einzelnen Fällen auf das Datenbanksystem zugegriffen werden kann. Anschließend wird anhand der Zugriffspfade gezeigt, an welcher Stelle der Datenbankarchitektur die Validierung von XML-Dokumenten ausgelöst wird. Als letztes wird in diesem Abschnitt ein Einblick in die Funktionsweise der Datenbanktrigger von eXist gegeben.

In Abschnitt 7 wird erläutert, warum es notwendig ist bei der Validierung zwischen XML- und OWL-Dokumenten zu unterscheiden und wie im Datenbanksystem festgelegt werden kann, welcher Dokumenttyp geprüft werden soll. Anschließend wird skizziert, wie die Konsistenzprüfung beim Einfügen eines neuen Dokumentes und beim Überprüfen eines vorhandenen Dokumentes aus Sicht des Datenbanksystems abläuft. In Abschnitt 7.3 werden drei Übergabemethoden an den OWL-Reasoner betrachtet und anschließend auf ihre Vor- und Nachteile hin untersucht. Diese werden miteinander verglichen um die beste Schnittstelle auszuwählen. Als Ergebnis geht die nRQL-Schnittstelle hervor. Zum Schluss werden in diesem Kapitel zwei Entwürfe für das OWL-Modul, welches dem Datenbanksystem die Konsistenzprüfung von OWL-Dokumenten ermöglicht, vorgestellt.

Als Detaillierung der groben Analyse der Zugriffspfade in Kapitel 6, wird in Kapitel 8 untersucht, was im Inneren des Datenbanksystems passiert, wenn ein Dokument validiert werden soll. Dazu wurde mit Hilfe von Sequenzdiagrammen der Zugriff auf das Datenbanksystem einschließlich der Auslösung der Validierung dargestellt. Die Sequenzdiagramme werden für die REST- und XML-RPC-Schnittstelle und für die XML:DB API beschrieben, da für diese Schnittstellen sichergestellt wird, dass die Konsistenzprüfung funktioniert. Als nächstes wird untersucht, welche Konfigurationsmöglichkeiten das Datenbanksystem in Bezug auf die Validierung bietet. Dabei stellt sich heraus, dass die Validierung über die Konfiguration ein- oder ausgeschaltet werden kann. Dies wird für das gesamte Datenbanksystem über die Konfigurationsdatei eingestellt. Collections in dem Datenbanksystem können diese globale Einstellung für sich und ihre Kinder durch eine eigene Konfigurationsdatei verändern. Anschließend wird beschrieben, was passiert, wenn ein Dokument aus dem Datenbanksystem ausgelesen wird, wie ein Java-Trigger aufgebaut ist und wie er funktioniert. Diese Ergebnisse werden später für die Entwicklung des `ValidationTriggers` benötigt.

In Kapitel 9 wird der verfeinerte Entwurf, des in Kapitel 7 vorgestellten Grobentwurfes für das OWL-Modul erläutert. Zusätzlich wird der Entwurf für die Integration des OWL-Modules in das Datenbanksystem beschrieben. Danach wird die geplante Erweiterung der Konfiguration des Datenbanksystems, damit zwischen der Validierung/Konsistenzprüfung von XML- und OWL-Dokumenten umgeschaltet werden

kann, vorgestellt. Abschließend wird der Entwurf des `ValidationTriggers` präsentiert, der dafür sorgt, dass die Dokumente auch bei einer Veränderung durch ein XQuery oder XUpdate auf ihre Konsistenz überprüft werden.

Nachdem der Gesamtarchitekturentwurf steht, wird in Kapitel 10 beschrieben, wie die geplanten Änderungen an dem XML-Datenbanksystem umgesetzt werden. Daraufhin wird erläutert wie der Entwurf für das Modul, welches die Verbindung zwischen dem Datenbanksystem und dem OWL-Reasoner herstellt, umgesetzt wird. Abschließend wird die Umsetzung des `ValidationTriggers` dargestellt.

Nach der technischen Umsetzung wird diese in Kapitel 11 evaluiert. Für die Evaluierung wird überprüft, ob die in Kapitel 1.2 gestellten Anforderungen an die Umsetzung erfüllt werden. Daraufhin folgt in Kapitel 12 eine Zusammenfassung der Arbeit. Abschließend wird in Kapitel 13 untersucht, in welchen Bereichen Verbesserungspotential besteht. Zu diesen Punkten werden Vorschläge gemacht, wie diese Verbesserungen konkret aussehen könnten.

3 Grundlagen

In diesem Kapitel wird ein kurzer Einblick in die Technologien gegeben, die direkt oder indirekt im weiteren Verlauf der Arbeit benötigt werden. Zunächst wird eine Übersicht über XML gegeben, wobei auf den Aufbau von XML-Dokumenten und auf den Zusammenhang zwischen einer Grammatik und einem XML-Dokument eingegangen wird. Daraufhin soll erklärt werden, wie XQuery verwendet wird um Anfragen an XML-Dokumente stellen zu können. Nach den Anfragesprachen folgt eine Einführung in die verschiedenen Typen von XML-Datenbanksystemen und eine Vorstellung von zwei möglichen Netzwerkschnittstellen. Im Weiteren folgen die Einblicke in die Funktionsweise und den Aufbau von RDF und OWL-DL-Ontologien. Abschließend wird ein Einblick in das Aufgabengebiet von OWL-Reasonern, gefolgt von zwei möglichen Netzwerkschnittstellen für OWL-Reasoner gegeben.

3.1 Extensible Markup Language (XML)

Die „Extensible Markup Language“ [BPSM+08] (XML) wurde vom World Wide Web Consortium[1] (W3C) im Jahr 1998 spezifiziert. Die XML-Spezifikation definiert XML wie folgt:

„Die Extensible Markup Language, abgekürzt XML, beschreibt eine Klasse von Datenobjekten, die XML-Dokumente genannt werden und beschreibt teilweise das Verhalten von Programmen, die XML-Dokumente verarbeiten. XML ist ein Anwendungsprofil oder eine eingeschränkte Form der Standard Generalized Markup Language (SGML) [ISO86]. Durch ihre Konstruktion sind XML-Dokumente konforme SGML-Dokumente.“

Wie SGML ist auch XML eine Dokumentenrepräsentationssprache, welche die Syntax zur Verfügung stellt, mit der die Struktur eines Dokumentes beschrieben werden kann. Diese Struktur ist unabhängig von der weiteren Verwendung oder Darstellung des Dokumentes [KST02]. Dem Inhalt der XML-Dokumente sind keine Grenzen gesetzt. Für XML existieren Schemasprachen (z.B. Document Type Definition (DTD) oder XML-Schema), mit denen Grammatiken für Dokumente definiert werden können.

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

Diese Grammatiken bilden die Sprachdefinition für XML basierte Sprachen. Ein Beispiel hierfür ist XHTML.

Ein beispielhaftes XML-Dokument, welches eine Liste von Adressen enthält, sieht folgendermaßen aus:

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <addressList xmlns="http://www6.cs.fau.de/xmlns/addressList">
3    <address>
4      <name>John Doe</name>
5      <street>Park Avenue</street>
6      <number>24</number>
7      <zip-code>1234</zip-code>
8      <city>Los Angeles/CA</city>
9      <country>USA</country>
10   </address>
11 </addressList>
```

Listing 3.1: XML-Beispieldatei

Die erste Zeile des Beispiels enthält die XML-Präambel, in der angegeben wird, welche XML-Version verwendet wird und mit welchem Zeichensatz (z.B. UTF-8) das Dokument kodiert ist. Wenn keine Kodierung angegeben ist, wird als Standardzeichenkodierung „UTF-8“ verwendet. In der zweiten Zeile wird ein Element addressList geöffnet, dem ein Attribut zugewiesen wird. Dieses Attribut heißt xmlns und gibt an, in welchem Namensraum die im Element addressList geschachtelten Elemente liegen. Sollte in einem dieser Elemente ein neuer Namensraum definiert werden, gehört dieses Element und alle darin geschachtelten Elemente zu dem neuen Namensraum. Ein anderes Attribut-Beispiel (siehe Beispiel 3.2) wäre, dem Element address das Attribut type hinzuzufügen, um unterscheiden zu können, ob es sich um die Heim- oder Arbeitsadresse handelt.

```
1  <address type="home">
```

Listing 3.2: Beispiel für ein Element mit Attribut

Nachdem das Element addressList geöffnet ist, wird in den folgenden Zeilen ein address-Element erstellt, welches geschachtelt eine Reihe von Elementen enthält, welche die Adresse spezifizieren. Anschließend werden alle offenen Elemente wieder geschlossen. Wenn die später in diesem Kapitel vorgestellte Beispielgrammatik als Dokumenttyp für dieses XML-Dokument verwendet wird, muss das Element addressList mindestens eine, darf aber beliebig viele Adressen enthalten. Bei den Namen der Elemente muss darauf geachtet werden, dass die Schreibweise, auch Groß-/Kleinschreibung, beim Öffnen und Schließen exakt gleich ist. Ansonsten wird bei der Verarbeitung nicht erkannt, dass es sich um dasselbe Element handelt.

Da es später notwendig ist zwischen syntaktischer Validität und logischer Konsistenz zu unterscheiden werden im Folgenden die Grundbegriffe „wohlgeformt“ und „valide“, so wie sie im XML-Kontext verwendet werden, erläutert. Ein Dokument ist nur dann ein XML-Dokument, wenn es wohlgeformt (engl. well-formed) ist. Damit ein Dokument als wohlgeformt bezeichnet werden kann, muss es verschiedene Kriterien erfüllen. Im Folgenden werden die wichtigsten Kriterien aufgezählt:

- Es muss ein eindeutiges Wurzel-Element existieren.
- Jedes Element, das geöffnet wird, muss auch wieder geschlossen werden.
- Ein leeres Element kann auch selbstschließend sein. Das Element <empty /> ist gleichzeitig ein öffnendes und ein schließendes Element und enthält keine Daten.
- Jedes Element, das nicht leer ist, muss korrekt geklammert werden. Das bedeutet, dass ein Element erst dann wieder geschlossen werden darf, wenn alle inneren Elemente geschlossen sind. Am Beispiel der addressList bedeutet dies, dass das addressList-Element erst dann geschlossen werden darf, wenn das letzte geöffnete address-Element geschlossen wurde.

Neben dem Kriterium der Wohlgeformtheit gibt es noch das Kriterium der Validität. Bei der Validität handelt es sich um eine Steigerung der Wohlgeformtheit, da jedes valide Dokument auch wohlgeformt sein muss. Ein Dokument ist weiterhin nur dann valide, wenn es einen Verweis auf die Grammatik, die dem Dokument zugrunde liegt, enthält und der Aufbau des Dokumentes mit den Vorgaben der Grammatik übereinstimmt. Um zu entscheiden, ob sich ein XML-Dokument an die Grammatik hält, wird ein validierender Parser verwendet, welcher das XML-Dokument auf seine Übereinstimmung mit den in der Grammatik definierten Regeln prüft.

Es gibt mehrere Schemasprachen mit denen die Grammatik eines XML-Dokumentes beschrieben werden kann. Die bekanntesten sind XML-Schema[2], Document Type Definition[3] (DTD) und Relax NG[4]. Die mit den Schemasprachen erstellten Grammatiken enthalten eine formale Definition für den syntaktischen Aufbau der zugehörigen XML-Dokumente. Für jedes Element oder Attribut wird angegeben, wie es heißt und welchem Datentyp die Werte entsprechen. Bei den Datentypen ist es nennenswert, dass DTD nicht die Datentypen der XML-Schema Spezifikation unterstützt, sondern eigene Datentypen besitzt. Schemasprachen erlauben es außerdem anzugeben, welche Werte ein Element/Attribut annehmen darf und welchen Standardwert das Attribut annehmen soll. Allerdings ist ein Attribut immer an ein Element gebunden und kann nicht alleine existieren. XML-Dokumente sind hierarchisch aufgebaut, weshalb ein

[2] http://www.w3.org/TR/xmlschema-0/
[3] http://www.w3.org/XML/1998/06/xmlspec-report-19980910.htm
[4] http://relaxng.org/

Element sowohl Nutzdaten als auch weitere Elemente (siehe Beispiel addressList) enthalten kann. Eine Grammatik, in DTD-Syntax, für das Beispiel addressList sieht wie folgt aus:

```
1 <!ELEMENT addressList (address+)>
2 <!ELEMENT address (name, street, number, zip-code, city, country)>
3 <!ELEMENT name (#PCDATA)>
4 <!ELEMENT street (#PCDATA)>
5 <!ELEMENT number (#PCDATA)>
6 <!ELEMENT zip-code (#PCDATA)>
7 <!ELEMENT city (#PCDATA)>
8 <!ELEMENT country (#PCDATA)>
```

Die mit einer Schemasprache erstellte Grammatik ist die Typdefinition für XML-Dokumente. Die Grammatik ist mit der Definition einer Tabelle in einem relationalen Datenbanksystem vergleichbar, da sie vorgibt, was in einem zugehörigen XML-Dokument enthalten sein darf. Die XML-Dokumente, die valide zu der Grammatik sind, sind Instanzen der Grammatik und mit einzelnen Zeilen in einer Tabelle bei relationalen Datenbanksystemen vergleichbar.

In XML ist es üblich, dass die Grammatik und die darauf basierenden XML-Daten in unterschiedlichen Dokumenten abgelegt sind. Daher ist es zwingend notwendig, dass ein Verweis auf die Grammatik im XML-Dokument enthalten ist.

3.2 XML Query Language (XQuery)

XQuery 1.0 [BCF+07] wurde vom W3C als Anfragesprache für Daten, die in XML gespeichert sind, spezifiziert. Der Grund für die Entwicklung war, dass immer mehr Daten in XML gespeichert werden, auf die mit intelligenten Anfragen zugegriffen werden soll. Ein Vorgänger von XQuery ist XPath [CD99] von welchem Eigenschaften in XQuery weiterverwendet wurden. Die Adressierung von bestimmten Blattknoten oder Unterbäumen im XML-Dokument erfolgt beispielsweise in der von XPath bekannten Syntax, die an den Pfad eines Linux Dateisystems erinnert.

Mit XQuery 1.0 wird eine Anfragesprache zur Verfügung gestellt, die mit der Structured Query Language (SQL) für relationale Datenbanksysteme vergleichbar ist. Bei diesem Vergleich ist darauf zu achten, dass nur der Teil, der von SQL mit den „SELECT FROM WHERE“-Ausdrücken zur Verfügung gestellt wird, betrachtet wird. XQuery 1.0 unterstützt nur Anfragen aber keine Veränderungsoperationen.

Anfragen werden in XQuery 1.0 mit FLWOR-Ausdrücken gestellt. FLWOR steht für „FOR LET WHERE ORDER BY RETURN“. Beim Aufbau einer Anfrage muss darauf geachtet werden, dass ein FOR, ein LET oder beides enthalten sein muss.

Die Teilausdrücke FOR oder LET weisen einer Variable die Ergebnismenge zu. Die Ausdrücke WHERE und ORDER BY sind optional, WHERE dient dazu die Ergebnismenge einzuschränken und mit ORDER BY kann das Ergebnis in die gewünschte Reihenfolge gebracht werden. Der Ausdruck RETURN schließt jede Anfrage ab und gibt an, was als Ergebnis zurückgegeben wird.

Eine Anfrage an die in Abschnitt 3.1 definierte Adressliste sieht beispielsweise folgendermaßen aus:

```
1 FOR $address in //address
2 WHERE $address/name='John Doe'
3 RETURN $address
```

Listing 3.3: Beispiel eines XQuery FLWOR-Ausdrucks

Im FOR-Ausdruck wird eine Menge von Adressen, die in der Adressenliste enthalten sind, der Variable address zugewiesen. Die Menge, die der Variable zugewiesen wird, ist vom Ergebnis des WHERE-Ausdruckes abhängig, da dieser die ausgewählten Adressen einschränkt. Im obigen Beispiel werden lediglich jene Adressen aus der Liste ausgewählt, bei denen der Adressat „John Doe“ heißt. Der RETURN-Ausdruck gibt die address-Variable mit der Menge aller Adressen, deren Bewohner „John Doe“ heißen zurück.

Mit der XQueryUpdateFacility 1.0 [CFM+08] wurde vom W3C eine Erweiterung spezifiziert, die XQuery um eine Syntax erweitert, welche es ermöglicht, permanente Änderungen an Instanzen des XQuery 1.0 und XPath 2.0 Datenmodells zu vollziehen. Eine Beispielanfrage an die in Abschnitt 3.1 vorgestellte Liste aus Adressen sieht wie folgt aus:

```
1 FOR $address in //address
2 WHERE $address/city='Erlangen'
3 RETURN replace node $address/zip−code with <zip−code>91052</zip−code>
```

Listing 3.4: Beispiel einer Änderung durch einen XQueryUpdateFacility-Ausdrucks

Mit der Anfrage 3.4 soll bei allen Adressen, die als Stadt „Erlangen“ enthalten, die Postleitzahl (zip-code) auf „91052“ geändert werden. Dazu wird die Änderungsanweisung in Kombination mit einem FLWOR-Ausdruck verwendet. Die ersten beiden Zeilen sorgen dafür, dass die Variable address nur die Adressen enthält, die in Erlangen liegen. In Zeile 3 wird im Rahmen des RETURN der Änderungsausdruck angewendet. Dieser sorgt dafür, dass in jedem Element, welches in der Variable address enthalten ist das Element zip-code durch ein neues ersetzt wird, welches den Wert „91052“ besitzt.

3.3 XML-Datenbanksysteme

Ein XML-Datenbanksystem hat wie ein relationales Datenbanksystem die Aufgabe Daten möglichst effizient und dauerhaft zu speichern. Im Fall von XML-Datenbanksystemen sind die zu speichernden Daten XML-Dokumente. Auf diesen sollen wie bei anderen Datenbanksystemen (z.B. relationale Datenbanksysteme (RDBS)) Anfragen und Updates ausgeführt werden können. Für XML-Datenbanksysteme existiert hierfür beispielsweise XQuery. XML-Datenbanksysteme wurden von der „Initiative für XML-Datenbanken[5]" in drei Typen eingeteilt: native XML-Datenbanksysteme (NXD), XML-Enabled-Datenbanksysteme (XEDB) und Hybrid-XML-Datenbanksysteme (HXD) [Ini03].

Die erste Kategorie sind native XML-Datenbanksysteme, die laut Definition ein logisches Modell definieren müssen, welches zum Speichern und Ausgeben der XML-Dokumente verwendet wird. XML-Dokumente sind die fundamentalen Speichereinheiten in dem Datenbanksystem. Weiterhin enthält die Definition keine Einschränkungen hinsichtlich der Abbildung der Daten auf den physischen Speicher.

Die zweite Kategorie sind XML-Enabled-Datenbanksysteme (XEDB). Diese Datenbanksysteme bestehen aus einem vorhandenen Datenbanksystem, über das eine weitere Schicht gelegt wird, welche die Abbildung von XML-Dokumenten auf die datenbankinterne Speicherstruktur übernimmt. Der Unterschied zwischen einem nativen XML-Datenbanksystem und diesem Typ eines XML-Datenbanksystems ist, die Abhängigkeit des fundamentalen Speichertyps von der Implementierung des XEDB.

Die letzte Kategorie sind hybride-XML-Datenbanksysteme (HXD). Diese Datenbanksysteme stellen eine Kombination aus den beiden oben genannten Typen dar. Dadurch besitzt ein hybrides-XML-Datenbanksystem sowohl die Eigenschaften eines nativen XML-Datenbanksystems als auch die eines XML-Enabled-Datenbanksystems. Das hybride-XML-Datenbanksystem kann von einer Anwendung je nach Bedarf als natives XML-Datenbanksystem oder als XML-Enabled-Datenbanksystem behandelt werden.

Außer der Einteilung von XML-Datenbanksystemen in verschiedene Typen hat die „Initiative für XML-Datenbanken" auch das XML Database Applications Programming Interface (XML:DB API) [SXAML01] spezifiziert. XML-Datenbanksysteme, die über diese Schnittstelle verfügen, können hierdurch als Modul in andere Softwareprojekte eingebettet werden. Die Verwendung der XML:DB API bietet den Vorteil, dass das XML-Datenbanksystem im Softwareprojekt jederzeit gegen eine anderes, das auch die XML:DB API besitzt ausgetauscht werden kann, ohne dass dabei Änderungen am Quelltext des Softwareprojektes vorgenommen werden müssen.

[5]Zu finden unter: http://xmldb-org.sourceforge.net/

3.4 Netzwerkschnittstellen

In diesem Abschnitt werden zwei Netzwerkschnittstellen kurz vorgestellt. Diese Schnittstellen werden häufig von nativen XML-Datenbanksystemen bereitgestellt. Zuerst wird die REST-Schnittstelle und anschließend die XML-RPC-Schnittstelle vorgestellt.

3.4.1 Representational State Transfer (REST)

REST [Fie00] wurde von R. T. Fielding in seiner Doktorarbeit definiert. Die Dissertation stellt mehrere Bedingungen an die REST Architektur. Es muss eine Trennung zwischen Client und Server stattfinden, wodurch eine unabhängige Entwicklung von Client und Server gewährleistet wird. Die Verbindung zwischen Client und Server ist zustandslos, da jede Client-Anfrage alle Daten enthält, die für eine Verarbeitung notwendig sind. Bei der Antwort des Server auf eine Anfrage des Clients muss eine implizite oder explizite Kennzeichnung vorliegen, ob sie vom Client zur Wiederverwendung im Cache gespeichert werden darf. Client und Server müssen durch eine einheitliche Schnittstelle (uniform interface) verbunden sein. REST wird häufig in Verbindung mit HTTP[6] (PUT, GET, DELETE) verwendet, ist aber nicht darauf beschränkt. HTTP wird bei der Verwendung mit REST nicht verändert, sondern REST verwendet die vorhandene Semantik von HTTP. Als nächstes muss das System in hierarchische Schichten eingeteilt werden, so dass jede Komponente nur die Schicht sieht, mit der sie interagiert. Schichten die darüber hinausgehen sind für sie unsichtbar. Über die Code-on-Demand Funktion ist es möglich die Funktionalität des Clients zu erweitern, diese Bedingung ist jedoch optional. Client und Server kommunizieren über Ressourcen, die beispielsweise durch eine URL adressiert werden.

3.4.2 XML-RPC

Bei der XML-RPC-Schnittstelle [Win03] handelt es sich um einen Vorgänger von SOAP. Die Schnittstelle verwendet HTTP POST um Daten von einem Client an den Server zu übertragen. Die Daten sind in wohlgeformtem XML verfasst. Mit den Daten wird auf dem Server ein Remote-Procedure-Call (RPC) ausgelöst. Das Ergebnis des Remote-Procedure-Calls wird ebenfalls in XML verfasst und an den Client übermittelt.

[6] Abkürzung für: Hypertext Transfer Protocol

3.5 Resource Description Framework (RDF)

Das Resource Description Framework (RDF) ist eine vom W3C empfohlene Sprache, die zur Darstellung von Beschreibungsdaten, in Form von Graphen verwendet wird. Zur Speicherung der Graphen wird XML verwendet. Die Grundbausteine von RDF sind Ressourcen (resources), Eigenschaften (properties) und Aussagen (statements).

In RDF [MMM04] kann alles eine Ressource sein, egal, ob es sich um ein Konzept, oder um einen Gegenstand der wirklichen Welt handelt, solange ein Uniform Resource Identifier (URI) darauf zeigen kann. Bei den von RDF definierten Eigenschaften handelt es sich um eine Spezialform von Ressourcen, die eine Beziehung zwischen anderen Ressourcen angeben. Aussagen in RDF sind Tripel, die nach dem Schema „Subjekt, Prädikat, Objekt“ aufgebaut sind und die Eigenschaften von Ressourcen beschreiben. Das Subjekt ist in diesem Fall eine Ressource, über die eine Aussage gemacht werden soll. Beim Prädikat handelt es sich um eine Eigenschaft, welche Subjekt und Objekt miteinander verbindet. Beim Objekt kann es sich entweder um eine Ressource, oder um ein Literal handeln. Ein Literal ist eine Zeichenfolge zum Darstellen von Basistypen (Boolean, Integer, String, etc.). Im Folgenden werden an einem Beispiel die verschiedenen Darstellungen von Aussagen gezeigt.

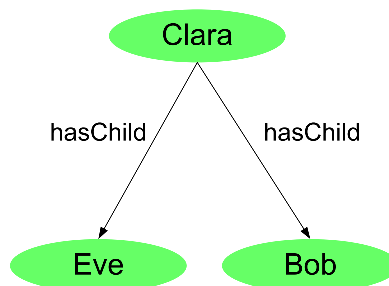


Abbildung 3.1: Clara hat die Kinder Eve und Bob

Aus obigem Graphen ergeben sich folgende Tripel:

```
1 Clara hasChild Eve
2 Clara hasChild Bob
```

Listing 3.5: Beispiel in Tripelstruktur

Zum Speichern der Graphen stellt RDF eine XML Syntax zur Verfügung. Um die Tripel eines Graphen in XML darzustellen werden RDF Descriptions verwendet. Das würde für den oberen Tripel wie folgt aussehen:

```
1 <rdf:Description rdf:about="http://www6.cs.fau.de/people/Clara">
2   <hasChild rdf:resource="http://www6.cs.fau.de/people/Eve"/>
3   <hasChild rdf:resource="http://www6.cs.fau.de/people/Bob"/>
4 </rdf:Description>
```

Listing 3.6: RDF-Beispiel

Die erste Zeile enthält das Subjekt des Tripels, über das eine Aussage getroffen werden soll. Das Subjekt ist mit Hilfe des URIs der Ressource identifiziert und ist in diesem Fall „Clara“. Der von der Beschreibung (rdf:Description) umschlossene Ausdruck enthält das Prädikat, gefolgt von dem Objekt. Der Ausdruck enthält zuerst die Eigenschaft des Subjekts, in diesem Fall „hasChild“. Auf die Eigenschaft folgt ein URI, welcher auf die Ressource „Eve“ zeigt. Das obige Beispiel verdeutlicht, dass eine Aussage aus mehreren Tripeln bestehen kann.

Das RDF Schema (RDFS) stellt für RDF eine Menge von vordefinierten Elementen für die Beschreibung von Ontologien zur Verfügung. Die Elemente Klasse (classes) und Eigenschaft (properties) sind die Grundbausteine einer Ontologie. Bei den Elementen Typ (type) und Unterklasse (subClassOf) handelt es sich um mitgelieferte Eigenschaften. Die Elemente Domain (domain) und Spannweite (range) sind wiederum Eigenschaften von Eigenschaften.

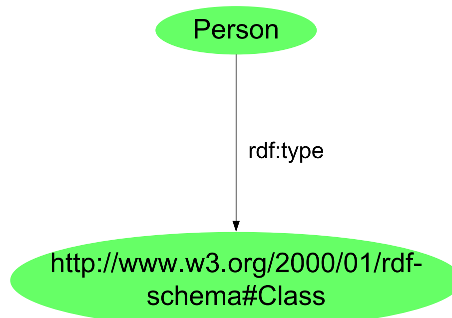


Abbildung 3.2: Eine Person ist vom Typ Klasse

Aus obigem Graphen ergeben sich folgende Tripel:

```
1 Person rdf:type rdfs:Class
```

In RDF-Syntax würde der Tripel folgendermaßen aussehen:

```
1 <rdf:Description rdf:ID="Person">
2   <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
3 </rdf:Description>
```

Listing 3.7: RDF-Beispiel

Auf den ersten Blick gibt es keinen großen Unterschied zwischen den beiden Beispielen. Der Unterschied ist dennoch wichtig. Beim ersten Beispiel handelt es sich um eine Aussage über ein Individuum, also um ein explizites Element. Im Gegensatz dazu wird im zweiten Beispiel eine Aussage über das Konzept Person aus einer Ontologie getroffen.

Die hohe Modellierungsfreiheit von RDF/RDFS ist einerseits von Vorteil, wenn komplexe Systeme beschrieben werden. Andererseits ist sie ein Nachteil, da durch sie die Entscheidbarkeit[7] von RDF nicht mehr garantiert werden kann.

3.6 OWL-DL-Ontologien

Der Begriff Ontologie [Gru93] stammt aus dem Griechischen und bedeutet soviel wie „Lehre des Seienden“. Dieser Begriff stammt ursprünglich aus der Philosophie und wurde in die Informatik übertragen. Ontologien wurden im Bereich der künstlichen Intelligenz übernommen und darin für Wissensrepräsentationen verwendet. In der Philosophie gibt es nur eine einzige Ontologie, die einen ganzheitlichen Anspruch besitzt. In der Informatik gibt es jedoch beliebige Ontologien, die jeweils nur einen beschränkten Kontext besitzen.

Eine Ontologie in der Informatik definiert das Modell eines Gegenstands- oder Wissensbereiches. Dieses Modell beschreibt die Menge aller darstellbaren Elemente. Zusätzlich existieren explizite Elemente dieser Menge, welche als Individuen bezeichnet werden.

Für die Definition von Ontologien existiert beispielsweise die Web Ontology Language (OWL). OWL [SWM04] wurde vom W3C aus der Sprache DAML+OIL entwickelt. Der Vorgänger von OWL (DAML+OIL) wurde in Zusammenarbeit von amerikanischen und europäischen Forschern hervorgebracht. Zum Speichern von Ontologien verwendet OWL die XML-Syntax, welche in RDF/RDF-Schema definiert ist. OWL verwendet RDF um Ontologien zu definieren. RDF bietet zur Definition von Konzepte Klassen (classes) und Eigenschaften (properties) um Rollen zu definieren. Im Gegensatz zu RDF unterscheidet OWL verschiedene Arten von Eigenschaften, beispielsweise Objekteigenschaften (object properties) und Datentypeigenschaften (datatype properties). Die Spezifizierung des W3C teilt OWL in drei Teilsprachen auf. Diese Teilsprachen haben aufsteigend eine immer höhere Ausdrucksmächtigkeit. Die folgende Übersicht gibt einen Einblick in die Funktionalität der OWL-Teilsprachen: OWL-Lite, OWL-DL und OWL-Full.

[7]Entscheidbarkeit bedeutet in diesem Fall, dass nicht für jeden RDF Graphen in endlicher Zeit entschieden werden kann, ob er konsistent ist oder nicht.

- Mit OWL-Lite können Daten hierarchisch klassifiziert werden (beispielsweise: Menge aller Lebewesen, Säugetiere sind eine Untermenge von Lebewesen, Menschen sind bestimmte Säugetiere und folglich auch Lebewesen) und es können einfache Einschränkungen definiert werden (beispielsweise: Kardinalität nur 0 oder 1). Durch die sehr eingeschränkte Ausdrucksmächtigkeit von OWL-Lite im Gegensatz zu OWL-DL oder OWL-Full ist es einfacher zu implementieren und effizienter in der Verarbeitung als OWL-DL.

- In OWL-DL sind alle von OWL bereitgestellten Sprachkonstrukte verfügbar. Bei der Verwendung dieser Sprachkonstrukte gibt es einige Einschränkungen. OWL-DL bietet einerseits eine hohe Ausdrucksmächtigkeit, andererseits ist die Entscheidbarkeit der Ontologie, beispielsweise bei einem Konsistenztest, garantiert. Eine Einschränkung ist, dass die RDFS/OWL Ausdrücke nur einen Typen haben dürfen. Beispielsweise darf eine Klasse die Unterklasse einer anderen Klasse sein, aber nicht gleichzeitig die Instanz einer anderen Klasse. Des Weiteren darf in OWL-DL fast das gesamte RDF-Schema Vokabular nicht verwendet werden.

- OWL Full bietet die größtmögliche Ausdrucksmächtigkeit und die syntaktische Freiheit von RDF, gleichzeitig aber keine Garantie für die Entscheidbarkeit. In OWL-Full sind die in RDF-Schema definierten Konstrukte (Class, Property, etc.) äquivalent zu denen von OWL. Beispielsweise ist rdfs:Class äquivalent zu owl:Class. Alle Einschränkungen, die für OWL-DL oder OWL-Lite gelten, sind für OWL-Full nicht gültig. OWL-Full stellt alle OWL-Sprachkonstrukte zur Verfügung und erlaubt die Verwendung aller RDF-Schema-Konstrukte.

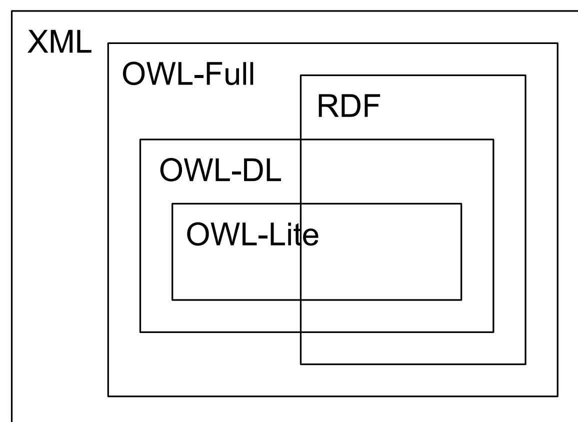


Abbildung 3.3: Beziehung der Teilsprachen von OWL zu RDF

Die Abbildung 3.3 zeigt, wie die einzelnen Teilsprachen von OWL zueinander und zu RDF in Beziehung stehen. OWL-Full stellt eine Erweiterung von RDF dar. OWL-DL

und OWL-Lite übernehmen nur einen Teil des Vokabulars und der Ausdrucksmächtigkeit von RDF und stellen darauf aufbauend eine Erweiterung der übernommenen Teile dar. Die Abbildung zeigt außerdem, dass OWL-DL eine echte Erweiterung von OWL-Lite darstellt, genau wie OWL-Full eine Erweiterung von OWL-DL ist. Daraus folgt laut der Spezifikation [SWM04]:

- Jede zulässige OWL-Lite Ontologie ist eine zulässige OWL-DL Ontologie.
- Jede zulässige OWL-DL Ontologie ist eine zulässige OWL-Full Ontologie.
- Jede gültige OWL-Lite Aussage ist eine gültige OWL-DL Aussage.
- Jede gültige OWL-DL Aussage ist eine gültige OWL-Full Aussage.

OWL-DL ist die Untersprache, welche dem ursprünglichen DAML+OIL am ähnlichsten ist. In der Version 1.0 verwendet OWL-DL die SHOIN(D) [BCM+07a] Beschreibungslogik. SHOIN steht für die verwendete Beschreibungslogik, wohingegen das „(D)“ beschreibt, dass Datentypen verwendet werden. In OWL werden beispielsweise XSD-Datentypen verwendet. Die Beschreibungslogik kennt Konzepte (Klassen), Rollen (Eigenschaften) und Individuen. Die Beschreibungslogik SHOIN erlaubt die Definition atomarer Konzepte, die Negation von Konzepten, und Schnittmengenbildung und die Vereinigung von Konzepten. Fundamentale Konzepte sind `owl:Thing` und `owl:Nothing`, sie entsprechen $\top$ (wahr) und $\bot$ (falsch). Die Definition von Konzepten durch Aufzählung ihrer Individuen ist erlaubt. Die Rollen werden unterteilt in abstrakte Rollen (object properties) und konkrete Rollen (datatype properties). Rollen dürfen die Operatoren $\exists$ und $\forall$ in ihrem vollen Funktionsumfang verwenden. Außerdem existieren hierarchische Rollen (transitiv) wie beispielsweise `subClassOf` oder `subPropertyOf`. Für abstrakte Rollen kann die inverse Rolle definiert werden. Für Rollen kann angegeben werden, wie oft sie mindestens vorhanden sein müssen (`minCardinality`) beziehungsweise wie oft sie maximal vorhanden sein dürfen (`maxCardinality`).

Mit der Zeit zeigte sich, dass einige Ausdruckskonstrukte, die für die Modellierung von Ontologien nützlich sind, nicht in OWL-DL enthalten sind. Daher wurde die Beschreibungslogik SHOIN zu SROIQ [HKS06] erweitert, damit diese für die neue OWL2 Spezifikation verwendet werden kann. Durch diese Erweiterung ergeben sich folgende neue Funktionen:

- disjunkte, reflexive, irreflexive Rollen
- negierte Rollenaussagen
- qualifizierte Zahlenrestriktion
- „Self“ Konzept

- universelle Rolle
- Rolleninklusionsgrundsatz
- T-Box wird in T-Box (Konzepte) und R-Box (Rollen) unterteilt

OWL-DL-Ontologien sind identisch zu den Wissensdatenbanken (knowledge bases) in der Beschreibungslogik. Die Ontologien können in eine T-Box und eine A-Box unterteilt werden. Die T-Box besteht in OWL-DL aus einer Menge von Konzepten und Rollen, die wiederum durch Klassen (classes) und Eigenschaften (properties) modelliert werden. Eine A-Box enthält eine oder mehrere Aussagen über Individuen, welche in dem von den Konzepten der T-Box vorgegebenen Gegenstandsbereich (Menge) liegen sollen. Aus Sicht von OWL spricht nichts gegen eine Speicherung von T-Box und A-Box in separaten Dokumenten, solange beide weiterhin wohlgeformtes XML enthalten. Bei einer Aufspaltung einer T-Box auf mehrere Dokumente muss darauf geachtet werden, dass dasselbe Konzept nachträglich nicht in zwei T-Box-Fragmente mit unterschiedlicher Definition eingefügt wird.

Die für diese Arbeit interessante Untersprache von OWL ist OWL-DL, da mit ihr modellierte Ontologien, sowohl berechenbar, als auch entscheidbar sind. Daher wird sie in der Wissenschaft sehr häufig für die Repräsentation von Ontologien herangezogen.

3.7 OWL-Reasoner

Ein OWL-Reasoner ist ein Programm, mit dem es möglich ist Anfragen an OWL-Dokumente, welche vom Reasoner eingelesen wurden, zu stellen. Im Rahmen dieser Arbeit steht in Bezug auf OWL-Reasoner die Konsistenzprüfung von OWL-Dokumenten im Mittelpunkt. Um eine Aussage über die Konsistenz treffen zu können, muss zuerst die T-Box des Dokumentes untersucht werden. Dazu prüft der OWL-Reasoner, ob die T-Box widersprüchliche Konzepte enthält.
Das folgende Beispiel zeigt, wie ein solcher Widerspruch aussehen kann:

$\texttt{Vogel} \equiv \texttt{Tier} \sqcap \texttt{kannFliegen}$
$\texttt{Pinguin} \equiv \texttt{Vogel} \sqcap \neg\texttt{kannFliegen}$

Im obigen Beispiel wird das Konzept `Vogel` definiert als `Tier`, welches außerdem fliegen kann (`kannFliegen`). Im nächsten Schritt wird das Konzept `Pinguin` definiert. Ein Pinguin ist laut Definition ein `Vogel`, der aber nicht fliegen kann (`kannFliegen`). Dadurch kommt es zu einem Widerspruch, da der Pinguin nicht gleichzeitig fliegen und nicht fliegen kann.

Anschließend wird die Konsistenz der A-Box in Bezug auf die T-Box überprüft. Diese Prüfung wird durchgeführt, indem die Aussagen der A-Box dahingehend geprüft

werden, ob sie Konzepten der T-Box widersprechen. Widerspricht eine Aussage der A-Box der T-Box, so ist die A-Box inkonsistent.

Zum besseren Verständnis wird der Sachverhalt an einem Beispiel aus dem Buch „The Description Logic Handbook“ [BCM+07b] veranschaulicht. Eine A-Box enthält die Aussagen `{Mother(MARY), Father(MARY)}`, das Individuum `MARY` ist also gleichzeitig Vater (`Father`) und Mutter (`Mother`). Wenn diese A-Box auf ihre Konsistenz zu einer leeren T-Box geprüft wird, existiert kein Widerspruch und folglich ist die A-Box konsistent. Wird die A-Box aber auf die Konsistenz zu der „Family T-Box“, die Konzepte über familiäre Beziehungen enthält geprüft, ist sie nicht mehr konsistent. Die „Family T-Box“ enthält unter anderem folgende Konzepte:

$\texttt{Woman} \equiv \texttt{Person} \sqcap \texttt{Female}$
$\texttt{Men} \equiv \texttt{Person} \sqcap \neg\texttt{Woman}$
$\texttt{Father} \equiv \texttt{Man} \sqcap \exists\texttt{hasChild.Person}$
$\texttt{Mother} \equiv \texttt{Woman} \sqcap \exists\texttt{hasChild.Person}$
...

Die Konzepte Vater und Mutter der T-Box schließen sich gegenseitig aus. Das Konzept Mutter (`Mother`) ist definiert als Frau (`Woman`), die ein Kind hat (`hasChild.Person`). Das Konzept Vater (`Father`) als Mann (`Man`), der ein Kind hat (`hasChild.Person`). Endgültig sich ausschließend werden die Konzepte Mutter und Vater durch die Verbindung mit Mann und Frau, da Frau als Person (`Person`) und weiblich (`female`) und der Mann als Person und nicht Frau ($\neg$`Woman`) definiert ist.

Aus diesen Erklärungen folgt, dass bei einer Konsistenzprüfung zuerst die T-Box auf widersprüchliche Konzepte untersucht wird. Um die Konsistenz der A-Box zu prüfen werden ihre Individuen gegen die in der T-Box definierten Konzepte geprüft. Wenn weder die T-Box, noch die A-Box in Bezug auf die T-Box Widersprüche enthält, so ist die OWL-Ontologie konsistent.

3.8 DIG-Schnittstelle

Um die Kommunikation zwischen einem OWL-Reasoner und einem Client zu ermöglichen wurde von der DL Implementation Group[8] (DIG) die so genannte DIG-Schnittstelle [Bec03] entwickelt. Die DIG-Schnittstelle wird von den gängigen OWL-Reasonern, wie RacerPro oder FaCT++, unterstützt. Die Kommunikation zwischen Client und OWL-Reasoner basiert auf dem Hypertext Transfer Protocol (HTTP) und verwendet dessen PUT/GET Mechanismen. Die eigentliche Kommunikation erfolgt in XML-Syntax, für die das DIG-Schema in Form eines XML-Schemas definiert wurde.

[8] Zu finden unter: http://dl.kr.org/dig/

Dem Client steht eine ASK und TELL Syntax zur Verfügung, mit der dem OWL-Reasoner Fragen gestellt beziehungsweise Daten übermittelt werden können. Die Schnittstelle ermöglicht es OWL-Dokumente direkt über das Netzwerk zu übertragen. Dazu müssen die Dokumente in die vom DIG-Schema definierte „Konzeptsprache“ umgewandelt werden.

Die gängigen OWL-Reasoner unterstützen die DIG-Schnittstelle in der Version 1.1 (DIG 1.1), die allerdings ein großes Problem mit sich bringt: Die von DIG 1.1 [Bec06] angebotene Ausdrucksmächtigkeit ist nicht ausreichend um damit jede OWL-DL-Ontologie zu beschreiben. Dies ist besonders an den Datentypen zu erkennen. Wenn versucht wird ein Dokument, welches Datentypen enthält, die mit DIG 1.1 nicht beschreibbar sind umzuwandeln, so werden diese Tripel ignoriert. Dies hat die Auswirkung, dass das Ergebnis der Konsistenzprüfung nur für den eingeschränkten, mit DIG 1.1 beschreibbaren Teil gültig ist.

3.9 nRQL-Schnittstelle

Die nRQL-Schnittstelle wird nur vom OWL-Reasoner RacerPro[9] verwendet, da sie die RacerPro eigene new Racer Query Language (nRQL) für die Kommunikation zwischen OWL-Reasoner und Client verwendet. Die nRQL besitzt eine Lisp-artige Syntax. Damit können Anfragen an vorhandene OWL-DL-Ontologien gestellt werden, neue OWL-DL-Ontologien definiert werden, oder vorhandene OWL-DL-Ontologien verändert werden. Das der nRQL-Schnittstelle zugrunde liegende Netzwerkprotokoll ist TCP.

Ein großer Nachteil der nRQL-Schnittstelle ist, dass über die Netzwerkschnittstelle nur Kommandos an eine „Managementkonsole“ übertragen werden können, nicht aber ganze Dokumente: Wenn ein vorhandenes Dokument an RacerPro übergeben werden soll, muss dieses Dokument zuerst auf die Festplatte geschrieben werden, anschließend wird an RacerPro über nRQL der Dateiname und der Befehl die Datei einzulesen übergeben. Daraufhin liest RacerPro die Datei von der Festplatte ein. Die Interaktion mit der Festplatte kostet dabei Zeit, da Festplatten langsam sind und zuerst ein Schreib- und anschließend ein Lesezugriff notwendig ist. Dieser Mechanismus erzeugt sofort ein zweites Problem, da der Client und RacerPro auf demselben Rechner ausgeführt werden, oder zumindest auf ein gemeinsames Dateisystem Zugriff haben müssen.

[9] Zu finden unter: http://www.racer-systems.com/

3.10 Zusammenfassung

In diesem Kapitel wurde ein Einblick in die Grundlagen der beiden Themengebiete, die es im Rahmen dieser Arbeit zu verbinden heißt, gegeben. Einerseits wurde ein Einblick von der Datenbankseite in XML, XQuery und XML-Datenbanksysteme gegeben. Dies stellt die Grundlagen für XML-Dokumente, Anfragen an XML-Dokumente und die Speicherung von XML-Dokumenten in einem Datenbanksystem dar. Andererseits wurde ein Einblick in RDF, OWL-DL-Ontologien, OWL-Reasoner und die DIG- und nRQL-Schnittstelle gegeben. OWL-DL-Ontologien bauen auf RDF auf, welches wiederum in XML-Syntax formuliert werden kann. OWL-DL ermöglicht es somit Ontologien in XML-Syntax zu formulieren. OWL-Reasoner sind dazu da, um in OWL formulierte Ontologien auszuwerten, was in diesem Fall bedeutet, sie auf ihre Konsistenz zu überprüfen. Als letztes wurden mit der DIG- und nRQL-Schnittstelle zwei mögliche netzwerkbasierte Schnittstellen eines OWL-Reasoners vorgestellt.

4 Anforderungsanalyse

Zu Beginn dieses Kapitels wird kurz auf das Endnutzerszenario eingegangen. Dazu wird zuerst der Ist- und dann der Soll-Zustand beschrieben. Im zweiten Teil dieses Abschnittes wird erklärt, was bei der Konsistenzprüfung einer OWL-DL-Ontologie passiert. Anschließend werden mehrere Szenarien einer Konsistenzprüfung vorgestellt. Danach werden zwei Ansätze, wie OWL-Dokumente im Datenbanksystem abgelegt werden könnten, vorgestellt. Die Ansätze werden daraufhin auf ihre Vor- und Nachteile untersucht. Der dritte Teil beinhaltet eine Reihe von Anforderungen, welche das native XML-Datenbanksystem bzw. der OWL-Reasoner erfüllen muss, damit es/er für die Verwendung im Rahmen der vorliegenden Arbeit tauglich ist.

4.1 Endnutzer-Szenario

Im Folgenden wird zunächst erklärt, wie Speicherung und Konsistenzprüfung eines OWL-Dokumentes traditionell ablaufen. Anschließend wird erläutert, was die geplanten Erweiterungen aus der Sicht des Endnutzers an der Speicherung und Konsistenzprüfung ändert.

4.1.1 Ist-Zustand

Die Abbildung 4.1 stellt den gängigsten Fall des Ist-Zustandes dar. Der Ist-Zustand zeigt, dass die Konsistenzprüfung und die Speicherung der in OWL formulierten Ontologien fast immer voneinander getrennt sind. Ein Benutzer kann eine Ontologie in einem Editor entwerfen oder verändern. Anschließend muss er das OWL-Dokument, in dem die Ontologie enthalten ist auf ihre Konsistenz prüfen lassen. Einige spezielle OWL-Editoren wie beispielsweise Protégé[1] bieten mittlerweile eine automatische Prüfung der OWL-Dokumente an. Dies ist aber die Ausnahme, da OWL-Ontologien auch mit einem einfachen Texteditor wie VIM[2] oder Windows Notepad bearbeitet oder erstellt werden können.

[1] Zu finden unter: http://protege.stanford.edu/
[2] Zu finden unter: http://www.vim.org/

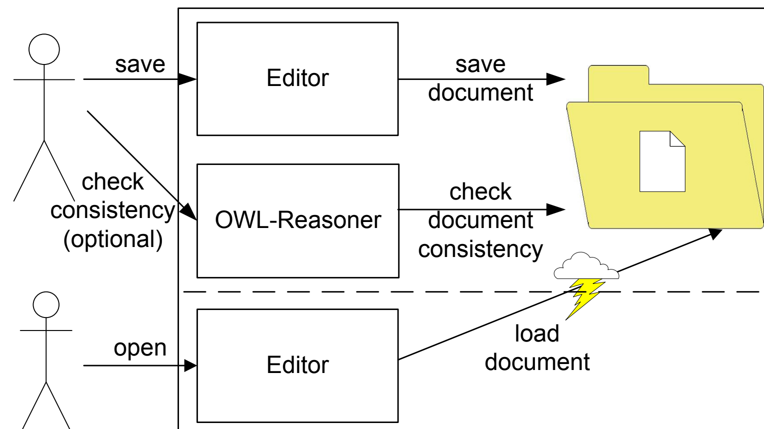


Abbildung 4.1: Ist-Zustand

Der Nachteil hieran ist, dass die Konsistenzprüfung nicht zwingend ist! Dadurch kann es vorkommen, dass OWL-Dokumente, die nicht geprüft wurden, in einem inkonsistenten Zustand abgelegt wurden. Ein weiterer Nachteil an der lokalen Speicherung ist, dass anderen Nutzern der Zugriff auf die OWL-Dokumente nur möglich ist, wenn der „Eigentümer“ sie ihnen zugänglich macht. Dies bringt das Problem mit sich, dass verschiedene Versionen derselben Ontologie entstehen können, wenn mehrere Personen Änderungen an ihrer lokalen Kopie vornehmen.

4.1.2 Soll-Zustand

Ziel der vorliegenden Arbeit ist es, ein vorhandenes natives XML-Datenbanksystem so zu erweitern, dass es im Stande ist, OWL-Dokumente automatisch auf ihre Konsistenz zu prüfen. Die Abbildung 4.2 zeigt, wie die Speicherung in einem Datenbanksystem aus Endnutzersicht abläuft. Dokumente können direkt vom Editor oder wenn dies nicht unterstützt wird, per Hand in das Datenbanksystem eingefügt werden.

Der Vorteil an diesem Szenario ist, dass die Speicherung des Dokumentes im Datenbanksystem und die Prüfung der Konsistenz des Dokumentes so miteinander verknüpft sind, dass das Datenbanksystem ein Dokument nur aufnimmt, wenn es konsistent ist. Der Vorteil ein solches Datenbanksystem zum Speichern der OWL-Dokumente zu verwenden ist außerdem, dass das Datenbanksystem eine zentrale Datenhaltung bietet, die vor allem nützlich ist, wenn mehrere Personen im Netzwerk auf dieselbe Ontologie zugreifen wollen. In diesem Fall bietet das Datenbanksystem den Vorteil, dass immer auf die aktuellste Version der Ontologie zugegriffen wird. Das Datenbanksystem sorgt durch Sperren dafür, dass entweder lesend, oder schreibend auf ein Dokument zuge-

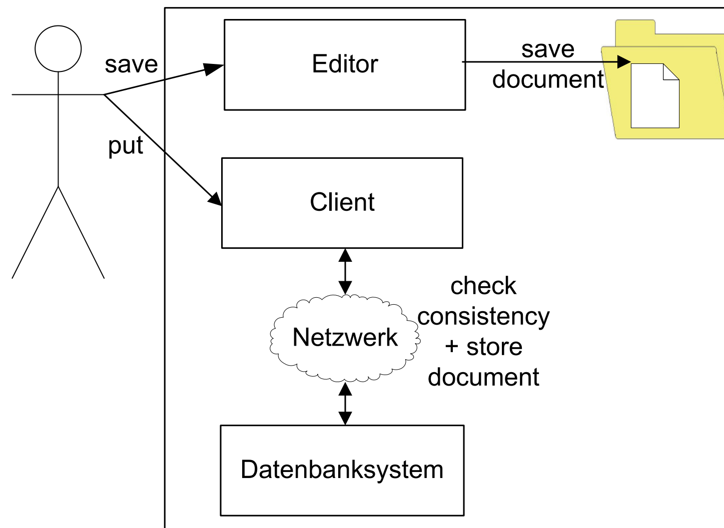


Abbildung 4.2: Soll-Zustand

griffen werden kann. Es kann entweder eine Person Änderungen an dem Dokument vornehmen, oder mehrere Personen lesend auf das Dokument zugreifen.

4.2 Fachliche Anforderungsanalyse

Eine logische Validierung bei Ontologien bedeutet, dass zuerst überprüft wird, ob die T-Box keine Widersprüche enthält. Wenn die T-Box keine Widersprüche enthält, liegt eine konsistente (engl. consistent) T-Box vor. Ist die T-Box konsistent, kann die A-Box untersucht werden. Bei der Überprüfung einer A-Box wird untersucht, ob die darin enthaltenen Aussagen, zu den in der T-Box definierten Konzepten passen. Ist dies der Fall, liegt eine A-Box vor, die konsistent (engl. consistent) zur T-Box ist.

In Abbildung 4.3 werden zunächst mögliche Kombinationen von T-Box und A-Box betrachtet. In der nächsten Stufe wird für jede der Kombinationen betrachtet, wie die Boxen beim Abbilden auf Dokumente fragmentiert werden können. Die Grundannahme ist, dass T-Box und A-Box separat gespeichert werden:

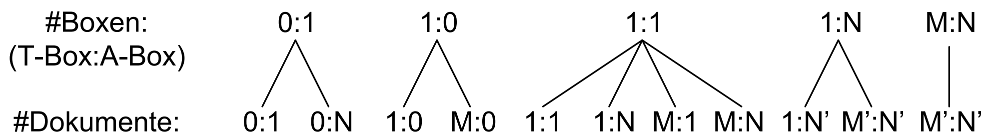


Abbildung 4.3: T-/A-Box Abbildung auf Dokumente

4.2.1 0 x T-Box und 1 x A-Box (0:1)

Ist eine A-Box ohne eine T-Box vorhanden, kann diese A-Box entweder gesamt auf ein Dokument abgebildet werden, oder die A-Box wird aufgeteilt und auf N Dokumente abgebildet.

- 0 x T-Box-Dokumente und 1 x A-Box-Dokument (0:1):
  Ein A-Box-Dokument, zu dem kein T-Box-Dokument vorhanden ist, wird bei der Konsistenzprüfung gegen eine leere T-Box geprüft. Das hat zur Folge, dass die Prüfung immer ergibt, dass das A-Box-Dokument konsistent ist, da die T-Box keine Konzepte enthält, denen die Aussagen des A-Box-Dokuments widersprechen könnten. In diesem Szenario kann nicht von einer Ontologie gesprochen werden, da kein Gegenstandsbereich definiert ist, in dem die in der A-Box definierten Individuen liegen.

- 0 x T-Box-Dokumente und N x A-Box-Dokumente (0:N):
  Wenn die A-Box auf N A-Box-Dokumente aufgespaltet wurde, müssen die Fragmente bevor eine Konsistenzprüfung möglich ist wieder zu einem Dokument zusammengefasst werden. Nach der Zusammenfassung zu einem Dokument gilt dasselbe wie im vorherigen Fall.

4.2.2 1 x T-Box und 0 x A-Box (1:0)

Ist eine T-Box ohne eine A-Box vorhanden, kann diese T-Box entweder gesamt auf ein Dokument abgebildet werden, oder die T-Box wird aufgeteilt und auf N Dokumente abgebildet.

- 1 x T-Box-Dokument und 0 x A-Box-Dokumente (1:0):
  Ist ein T-Box-Dokument vorhanden, zu dem kein A-Box-Dokument existiert, wird die Konsistenzprüfung des T-Box-Dokuments durchgeführt. Diese Ontologie ist genau dann konsistent, wenn das T-Box-Dokument konsistent ist. Die Ontologie enthält dann nur die Definition des Gegenstandsbereiches, aber keine expliziten Individuen.

- M x T-Box-Dokument und 0 x A-Box-Dokumente (M:0):
  Wenn die T-Box auf M T-Box-Dokumente abgebildet wird, dann bedeutet dies, dass alle M T-Box-Dokumente wieder zu einem T-Box-Dokument zusammengefasst werden müssen. Diese zusammengefasste T-Box wird wie im vorhergehenden Fall beschrieben auf ihre Konsistenz geprüft. (Anmerkung: Die Konsistenz jedes einzelnen T-Box-Dokumentes garantiert nicht, dass auch die kombinierte T-Box konsistent ist. Dieses Problem kann auftreten, da un-

terschiedliche T-Boxen gleichnamige Konzepte enthalten können, was bei der Kombination zu einer Inkonsistenz führt.)

4.2.3 1 x T-Box und 1 x A-Box (1:1)

Durch das Vorhandensein von genau einer T-Box und einer A-Box sind vier Fälle möglich. Die T-Box und die A-Box können in je einem Dokument abgelegt werden. In den Fällen zwei und drei wird einmal die T-Box und einmal die A-Box auf mehrere Dokumente aufgeteilt. Im letzten Fall werden sowohl A-Box als auch T-Box auf mehrere Dokumente aufgeteilt.

- 1 x T-Box-Dokument und 1 x A-Box-Dokument (1:1):
  Das T-Box-Dokument und das A-Box-Dokument werden vor der Konsistenzprüfung zu einem Dokument zusammengefasst. Bei der Konsistenzprüfung dieses Dokumentes wird zuerst die enthaltene T-Box auf ihre Konsistenz geprüft. Anschließend wird die enthaltene A-Box auf ihre Konsistenz in Bezug auf die T-Box geprüft. Sind sowohl die T-Box als auch die A-Box konsistent, ist die im Dokument enthaltene Ontologie konsistent.

- M x T-Box-Dokumente und 1 x A-Box-Dokument (M:1):
  Wenn die T-Box auf mehrere Dokumente aufgeteilt wurde, muss diese im ersten Schritt wieder zu einem T-Box-Dokument zusammengefasst werden. Anschließend wird dieses T-Box-Dokument wie im vorherigen Fall beschrieben mit dem A-Box-Dokument zusammengefasst und anschließend auf seine Konsistenz geprüft.

- 1 x T-Box-Dokument und N x A-Box-Dokumente (1:N):
  Wenn die A-Box auf mehrere Dokumente aufgeteilt wurde, muss diese im ersten Schritt wieder zu einem A-Box-Dokument zusammengefasst werden. Anschließend wird dieses A-Box-Dokument wie im ersten Fall beschrieben mit dem T-Box-Dokument zusammengefasst und anschließend auf seine Konsistenz geprüft.

- M x T-Box-Dokumente und N x A-Box-Dokumente (M:N):
  In diesem Fall werden die T-Box-Dokumente zu einem T-Box-Dokument und die A-Box-Dokumente zu einem A-Box-Dokument zusammengefasst. Die sich daraus ergebenden Dokumente werden zu einem Dokument, welches sowohl das T-Box-Dokument, als auch das A-Box-Dokument enthält, zusammengefügt. Das sich daraus ergebende Dokument wird wie im ersten Fall beschrieben auf seine Konsistenz geprüft.

4.2.4 1 x T-Box und N x A-Box (1:N)

Für das Szenario, dass eine T-Box und dazu N voneinander unabhängige A-Boxen existieren, werden zwei mögliche Fälle betrachtet. Im ersten Fall wird die T-Box auf ein Dokument und jede der N A-Boxen auf eines oder mehrere Dokumente abgebildet. Beim zweiten Fall wird die T-Box auf mehrere Dokumente aufgespaltet, wobei jede A-Box weiterhin auf eines oder mehrere Dokumente abgebildet wird.

- 1 x T-Box-Dokument und N' x A-Box-Dokumente (1:N'):
  Die N A-Boxen sind voneinander unabhängig. Für die Konsistenzprüfung wird die gesamte A-Box benötigt, weshalb alle zusammengehörigen A-Box-Dokumente in einen A-Box Dokument vereinigt werden. Dies geschieht für jede der N A-Boxen. Anschließend werden N Dokumente erstellt, die das T-Box-Dokument und immer genau eine der A-Boxen enthalten. Anschließend muss jedes dieser N Dokumente auf seine Konsistenz geprüft werden. Jedes Dokument, das eine konsistente T-Box und eine A-Box, die in Bezug auf die T-Box konsistent ist besitzt, enthält eine konsistente Ontologie.

- M x T-Box-Dokumente und N' x A-Box-Dokumente (M:N'):
  Falls die T-Box auf mehrere T-Box-Dokumente abgebildet wurde, ist es notwendig diese zuerst zu einem T-Box-Dokument zu verschmelzen. Anschließend werden, wie im letzten Fall beschrieben N Dokumente erstellt, welche die T-Box und je eine A-Box enthalten. Anschließend werden diese N Dokumente auf ihre Konsistenz geprüft.

4.2.5 M x T-Box und N x A-Box

In diesem Szenario wird der Fall betrachtet, dass M voneinander unabhängige T-Boxen und N voneinander unabhängige A-Boxen vorliegen. Jede der M T-Boxen hat N_i A-Boxen und jede A-Box gehört zu genau einer T-Box. Jede T-Box oder A-Boxen wird auf eines oder mehrere Dokumente abgebildet.

- M' x T-Box-Dokumente und N' x A-Box-Dokumente (M':N'):
  In diesem Fall sind M' T-Box-Dokumente und N' A-Box-Dokumente vorhanden. Bei der Konsistenzprüfung muss die T-Box und die A-Box wieder in ihrer Gesamtheit vorliegen. Jede T-Boxen wird aus ihren Fragmenten, die in T-Box-Dokumenten abgelegt sind zusammengesetzt. Dasselbe passiert für die A-Boxen. Anschließend wird jede A-Box mit der T-Box, zu der sie gehört, in einem Dokument zusammengefügt. Aus den T-Boxen und A-Boxen werden insgesamt N Dokumente mit M verschiedenen T-Boxen erstellt. Jedes Dokument enthält genau eine T-Box und eine A-Box. Jedes dieser Dokumente muss anschließend auf die Konsistenz geprüft werden. Das Dokument enthält genau dann eine

konsistente Ontologie, wenn die T-Box konsistent ist und die A-Box in Bezug auf die T-Box konsistent ist.

4.2.6 Überblick

Sowohl in Abschnitt 4.2.3, als auch im Abschnitt 4.2.4 wurde ein Fall vorgestellt, in dem ein T-Box-Dokument und mehrere A-Box-Dokumente existieren. Der Vergleich der beiden Fälle zeigt sofort, dass sie sich grundlegend unterscheiden. Im Fall in Abschnitt 4.2.3 wird die A-Box in N A-Box-Dokumente zerlegt, diese N A-Box-Dokumente ergeben bei der Konsistenzprüfung zusammen mit dem T-Box-Dokument ein Dokument, das eine Ontologie enthält. Im Gegensatz dazu beschreibt der Fall in Abschnitt 4.2.4 mehrere A-Box-Dokumente, die je eine vollständige und von den anderen A-Box-Dokumenten unabhängige A-Box, oder ein A-Box Fragment enthalten. Für die Konsistenzprüfung werden die A-Box-Dokumente zu N voneinander unabhängigen A-Boxen zusammengefügt. Die A-Boxen werden mit dem T-Box-Dokument kombiniert. Insgesamt ergeben sich daraus also N Dokumente. Der Unterschied liegt demnach darin, dass die Dokumente in Abschnitt 4.2.3 eine Ontologie repräsentieren und im zweiten Abschnitt N Ontologien.

Als nächstes wird der Fall, dass mehrere T-Box-Dokumente und mehrere A-Box-Dokumente vorhanden sind verglichen. Das trifft in den Abschnitten 4.2.3, 4.2.4 und 4.2.5 zu. Zuerst werden die Fälle aus den Abschnitten 4.2.3 und 4.2.4 verglichen. Anschließend werden die Fälle der Abschnitte 4.2.4 und 4.2.5 verglichen.

Der erste Vergleich ergibt, dass sich die T-Box-Dokumente in Abschnitt 4.2.3 zu einer T-Box zusammenfügen lassen, da die T-Box beim Abbilden auf Dokumente fragmentiert wurde. Dasselbe gilt für die A-Box-Dokumente. Die Boxen lassen sich wiederum zu einem aus T-Box und A-Box bestehenden Dokument zusammenfügen. In Abschnitt 4.2.4 können die T-Box-Dokumente zu einer T-Box zusammengefasst werden. Die A-Box-Dokumente gehören nicht alle zu derselben fragmentierten A-Box, sondern ergeben N voneinander unabhängige A-Boxen. Bei der Konsistenzprüfung werden aus den T-Box- und A-Box-Dokumenten folglich N Dokumente erzeugt, die je eine T-Box und eine A-Box enthalten. Der Unterschied liegt also darin, dass die Dokumente im ersten Abschnitt eine Ontologie repräsentieren und im zweiten Abschnitt N Ontologien.

Beim zweiten Vergleich ergibt der Abschnitt 4.2.4, wie bereits im letzten Vergleich erläutert, insgesamt N Ontologien. Die Dokumente im Abschnitt 4.2.5 enthalten entweder eine T-Box, eine A-Box, oder das Fragment einer T-/A-Box. Bei der Konsistenzprüfung ergeben alle T-Box-Dokumente insgesamt M voneinander unabhängige T-Boxen. Die A-Box-Dokumente ergeben insgesamt N A-Boxen. Für jede T-Box gilt, dass sie beliebig viele A-Boxen besitzen darf. Jede A-Box gehört zu genau einer T-Box.

Daraus ergeben sich nach dem Zusammenfügen N Ontologien, die M verschiedene T-Boxen besitzen.

Die in diesem Abschnitt beschriebene Separierung von T-Box und A-Box und die Fragmentierung der einzelnen T-Boxen und A-Boxen bei der Speicherung bringt einen Mehraufwand mit sich, da die einzelnen Dokumente für die Konsistenzprüfung wieder zusammengesetzt werden müssen. Daher werden im Folgenden zuerst die Vor- und Nachteile eines Ansatzes aufgezählt, bei dem jedes Dokument eine eigenständige Ontologie enthält:

+ OWL-Dokumente, die in das Datenbanksystem eingefügt werden, können nach einer erfolgreichen Validierung im Datenbanksystem gespeichert werden.
+ Im Fall einer Konsistenzprüfung wird das gesamte Dokument an den OWL-Reasoner übergeben.
+ Das Datenbanksystem kann beliebig viele T-Boxen enthalten, da jede T-Box mit ihrer A-Box in einem Dokument gekapselt ist.
+ Keine Änderungen am Datenbanksystem notwendig, da Funktionalität bereits vorhanden ist.
– Redundante Speicherung von Boxen, die in mehreren Dokumenten enthalten sind.

Wenn die Ontologien stets in T-Box und A-Box aufgeteilt werden und zusätzlich eine Aufteilung der einzelnen Boxen bei der Abbildung auf Dokumente zulässig ist, ergeben sich daraus eine Reihe von Vor- und Nachteilen:

+ Trennung von T-Box, die von Fachleuten entwickelt wurde und der A-Box, deren Daten von Sachbearbeitern eingepflegt werden.
+ Ähnliche T-Boxen können in mehrere Teile aufgespaltet werden, so dass die identischen Teile nur einmal im Datenbanksystem liegen.
+ Jede T-Box wird nur exakt einmal gespeichert.
+ T-Box-Dokumente können von unterschiedlichen Personengruppen fortentwickelt werden.
+ A-Box-Dokumente können von verschiedenen Sachbearbeitern erfasst werden.
– Die vorgestellten Szenarien für getrennt gespeicherte T- und A-Box müssen im Datenbanksystem umgesetzt werden.
– Dokumente müssen beim Einfügen auf ihren Inhalt (T-Box/A-Box) untersucht und gegebenenfalls aufgeteilt werden.

– Vor der Prüfung müssen T-Box(en) und A-Box(en) wieder zusammengesetzt werden.

Die Betrachtung der beiden Ansätze hat gezeigt, dass die Trennung von T-Box und A-Box eine Reihe von Vorteilen mit sich bringt. Der Mehraufwand, der für die Umsetzung dieses Ansatzes notwendig wäre, ist so hoch, dass hier bei der Entwicklung des Prototypen der Ansatz verwendet wird, bei dem Dokumente im Datenbanksystem die gesamte Ontologie enthalten. Die zukünftige Umsetzung des anderen Ansatzes wird bei der Entwicklung des Prototypen in der Architektur bereits vorbereitet.

4.3 Technische Anforderungsanalyse

Für dieses Projekt werden zwei Softwarebausteine benötigt, die kombiniert werden sollen. Der erste Baustein ist ein Programm, mit dem Ontologien, die in OWL-DL beschrieben sind, auf ihre Konsistenz untersucht werden können. Ein solches Programm heißt OWL-Reasoner. Ein OWL-Reasoner, der für dieses Projekt verwendet werden soll, muss eine Reihe von Anforderungen erfüllen:

(R1) Der OWL-Reasoner muss als Black-Box einsetzbar sein. Der Anwender soll ein OWL-Dokument an den OWL-Reasoner übergeben können und auf Befehl das Ergebnis der Konsistenzprüfung erhalten.

(R2) Der OWL-Reasoner muss sowohl T-Boxen als auch A-Boxen verarbeiten können.

(R3) Der OWL-Reasoner muss kompatibel zu OWL-DL und den XSD-Datentypen sein.

(R4) Es muss möglich sein OWL-DL Dokumente mit dem OWL-Reasoner auf ihre Konsistenz zu testen.

(R5) Der OWL-Reasoner muss eine Netzwerk- oder Programmschnittstelle besitzen.

(R6) Der OWL-Reasoner muss im Rahmen dieses Projektes kostenlos verfügbar sein.

Der zweite für diese Arbeit benötigte Baustein ist ein natives XML-Datenbanksystem. In diesem sollen die mit dem OWL-Reasoner auf ihre Konsistenz geprüften OWL-Dokumente gespeichert werden. Wie für den OWL-Reasoner gelten auch für das native XML-Datenbanksystem eine Reihe von Anforderungen, die es erfüllen muss, damit es für dieses Projekt verwendet werden kann:

(R7) Das native XML-Datenbanksystem muss der Definition eines nativen XML-Datenbanksystems (vgl. Abschnitt 3.3) entsprechen.

(R8) Das native XML-Datenbanksystem muss frei verfügbar sein.

(R9) Der Quelltext des Datenbanksystems muss verfügbar sein, da Veränderungen daran vorgenommen werden sollen.

(R10) Das native XML-Datenbanksystem muss XML-Dokumente anhand von Schemasprachen (z.B. DTD, XML-Schema, RelaxNG) validieren können.

(R11) Der XML-Validator sollte möglichst modular im Quelltext vorhanden sein, damit ein einfaches Austauschen möglich ist.

(R12) Das native XML-Datenbanksystem muss im Server-Modus betrieben werden können, damit über das Netzwerk von verschiedenen Rechnern auf es zugegriffen werden kann.

(R13) Das native XML-Datenbanksystem muss eine Möglichkeit bieten, Anfragen an die in ihm gespeicherten Daten auszuführen.

(R14) Das native XML-Datenbanksystem sollte möglichst große Datenmengen speichern können, da OWL-Dokumente umfangreich sein können.

4.4 Zusammenfassung

Der Abschnitt über das Endnutzerszenario zeigt, was die Verwendung eines Datenbanksystems zum Speichern der Dokumente für den Benutzer ändert und welche Vorteile der Benutzer durch diese Änderungen hat.

Bei der Entwicklung des Prototypen liegt der Fokus auf dem einfachen Ansatz, der gemischte Dokumente, die T-Box und A-Box enthalten, verwendet. Die Architektur des Prototypen wird jedoch so angelegt, dass eine Umsetzung des komplizierteren Ansatzes in Zukunft problemlos möglich ist. Dieser Ansatz hat den großen Vorteil, dass dies von nativen XML-Datenbanksystemen bereits unterstützt wird, da OWL-Dokumente in gültigem XML formuliert sind.

Zum Schluss wurden die Anforderungen, die an die Softwarebausteine, welche für dieses Projekt verwendet werden sollen, definiert. Bevor eine endgültige Entscheidung getroffen werden kann, müssen im Folgenden eine Reihe der zur Verfügung stehenden Bausteine dahingehend geprüft werden, ob sie diesen Anforderungen entsprechen.

5 Evaluation verfügbarer Softwarebausteine

Im folgenden Abschnitt werden zunächst verschiedene OWL-Reasoner auf ihre Funktionalität hin untersucht. Anschließend findet eine Untersuchung verschiedener XML-Datenbanksysteme statt. Die Funktionalität der OWL-Reasoner und der XML-Datenbanksysteme wird daraufhin mit den in Kapitel 4 beschriebenen Anforderungen verglichen, um zu eruieren, welche der Systembausteine die beste Wahl darstellen.

5.1 OWL-Reasoner

Bei einem OWL-Reasoner handelt es sich um ein Programm, welches OWL-Dokumente, wie in Kapitel 4 beschrieben, auf ihre Konsistenz prüft. Im weiteren Kapitel werden mehrere OWL-Reasoner vorgestellt und es wird untersucht, ob sie die Anforderungen, die in Kapitel 4 an OWL-Reasoner gestellt werden, erfüllen.

5.1.1 FaCT++

Bei FaCT++[1] [Tsa07] handelt es sich um eine Weiterentwicklung des Fast Classification of Terminologies (FaCT) [Hor98] Reasoners. Der FaCT++ Reasoner ist, zur Erhöhung der Effizienz in C++ implementiert und verwendet teilweise bereits in FaCT enthaltene Algorithmen. FaCT++ verwendet den Apache Xerces 2 XML-Parser bei der Verarbeitung der DIG Kommunikation.

FaCT++ liegt momentan in der Version 1.3.0 (Release 29.05.2009) vor. Für die Validierung eines OWL-Dokumentes ist kein Wissen darüber nötig, wie die Validierung intern funktioniert (R1). Mit FaCT++ ist es möglich Anfragen, die sich auf die Konzepte der T-Box, und Anfragen, die sich auf die Individuen in einer A-Box beziehen, zu stellen (R2). FaCT++ unterstützt nicht nur die SHOIN Logik von OWL 1.0 (R4), sondern sogar die Erweiterung SROIQ für OWL 2. Die XSD-Datentypen werden noch nicht vollständig unterstützt (R3), es werden beispielsweise die Datentypen String

[1] FaCT++ ist zu finden unter: http://code.google.com/p/factplusplus/

und Integer [TH06] unterstützt, andere primitive Datentypen beispielsweise Boolean jedoch nicht. FaCT++ stellt eine HTTP-Schnittstelle zur Verfügung. Über diese Schnittstelle kann mit dem DIG-Protokoll, über das Netzwerk auf den OWL-Reasoner zugegriffen werden (R5). FaCT++ [Tsa09] ist ein Open Source Projekt, das unter der GNU Public Licence (GPL) steht (R6).

5.1.2 Pellet

Die Entwicklung des Pellet[2] [SPG+07] OWL-Reasoners begann an der Universität Maryland, um zu beweisen, dass die Vorgaben des W3C zur Web Ontology Language (OWL) umsetzbar sind. Heute wird Pellet, weiterhin als Open Source OWL-Reasoner, von der Firma Clark & Parsia [Cla09] weiterentwickelt. Diese Firma, stellt für Open Source Projekte weiterhin eine kostenlose Version unter der GNU Affero General Public License (AGPL) Version 3 zur Verfügung (R6).

Zur Prüfung der Konsistenz eines OWL-Dokumentes, wird es an Pellet übergeben. Das Ergebnis der Konsistenzprüfung wird dem Benutzer von Pellet anschließend mitgeteilt (R1). Pellet liegt in der Version 2.0.0 RC7 vor (Release 11.06.2009) und ist in Java implementiert. In der aktuellen Version unterstützt Pellet die gesamte SHOIN Logik für die Verarbeitung von OWL-DL-Ontologien (R4) und besitzt Erweiterungen für die Verarbeitung von Ontologien im zukünftigen OWL 2 Standard (SROIQ Logik). Diese Logiken werden mit allen XSD-Datentypen unterstützt (R3). Wie die meisten OWL-Reasoner stellt Pellet eine HTTP-Schnittstelle, über die mit dem DIG Protokoll kommuniziert werden kann, zur Verfügung (R5). Neben dem DIG Interface gibt es auch eine OWL-API[3], welche es erlaubt, direkt aus einer Java Applikation auf den Reasoner zuzugreifen. Pellet unterscheidet die OWL-Dokumente in T- und A-Boxen, weiterhin können mit der Anfragesprache SPARQL Anfragen an die A-Boxen gestellt werden (R2).

5.1.3 RacerPro

RacerPro ist ein Produkt der Racer Systems GmbH & Co. KG, welches zur semantischen Validierung von Ontologien im OWL Format verwendet werden kann. RacerPro liegt in der Version 1.9.0 bzw. 1.9.2 BETA vor. Obwohl es sich bei RacerPro um ein kommerzielles Produkt handelt, werden zeitlich begrenzte Testlizenzen für Grundlagenforschung und Lehre gewährt (R6), falls es sich nach Erachten der Racer Systems GmbH & Co. KG um ein nicht-kommerzielles Projekt handelt. Um diese Lizenz zu

[2]Zu finden unter: http://clarkparsia.com/pellet/
[3]Zu finden unter: http://owlapi.sourceforge.net/

erhalten, ist es notwendig sich auf der Webseite[4] der Firma registrieren zu lassen und den geplanten Verwendungszweck der Software anzugeben.

RacerPro besitzt zwei verschiedene Ausführungsmodi, wobei der Standardmodus der Server-Modus ist (R5). Dabei öffnet RacerPro einen HTTP- und einen TCP-Port. Auf dem HTTP-Port bietet RacerPro eine DIG-Schnittstelle, auf die beispielsweise über Protégé zugegriffen werden kann, um Ontologien auf ihre Konsistenz untersuchen zu lassen. Der geöffnete TCP-Port stellt eine Schnittstelle für die RacerPro-eigene Sprache, die „New RacerPro Query Language“ (nRQL) zur Verfügung. Auf diese Schnittstelle kann mit Hilfe des `jracer`- oder des `lracer`-Moduls zugegriffen werden. Das `lracer`-Modul stellt eine Lisp API, das `jracer`-Modul eine Java API zur Verfügung. Diese APIs können schnell und einfach in Projekte integriert werden, um mit RacerPro zu kommunizieren. Hierbei ist zu beachten, dass die nRQL-/DIG-Schnittstelle bei einer Lizenz für Forschung und Lehre nur am lokalen PC und nicht über das Netzwerk zur Verfügung steht.

Neben dem Server-Modus kann RacerPro auch lokal ausgeführt werden. Dazu wird dem Programm beim Start auf der Kommandozeile eine Datei mit einer Ontologie und eine Datei mit Anfragen, z.B. zur Validierung, übergeben. Die Ergebnisse werden in diesem Fall für den Benutzer auf die Kommandozeile ausgegeben (R1).

RacerPro kann sowohl OWL-Lite als auch OWL-DL-Dokumente verarbeiten (R4), dafür gelten aber Einschränkungen, da Nominals[5] nicht direkt verarbeitet werden können, sondern approximiert werden müssen. Außerdem werden nur die XSD-Datentypen unterstützt (R3), aber keine benutzerdefinierten XML-Datentypen. RacerPro verwendet als Beschreibungslogik nicht SHOIN, sondern SHIQ [BCM+07a], bei der es sich um eine Vorgängerversion von SHOIN handelt [Rac05]. Dadurch ist beispielsweise zu erklären, warum RacerPro keine benutzerdefinierten Datentypen unterstützt. RacerPro unterstützt das Arbeiten mit mehreren T-Boxen und kann die Konsistenz einer A-Box in Bezug auf eine T-Box prüfen (R2).

5.1.4 Übersicht

Tabelle 5.1 zeigt, welche Anforderungspunkte die untersuchten OWL-Reasoner erfüllen. Daraus lässt sich leicht erkennen, dass die XSD-Datentypen nur von RacerPro und von Pellet vollständig unterstützt werden. Diese sind aber notwendig, damit die Konsistenztests in jedem Fall die korrekten Ergebnisse liefern. RacerPro und Pellet sind nach den in Kapitel 4 gestellten Anforderungen durchaus gleichwertig.

[4] http://www.racer-systems.com/

[5] Von Nominals wird gesprochen, wenn in der T-Box Klassen durch die Aufzählung ihrer Individuen definiert werden. Ein Beispiel dafür ist die Klasse Wochentage, die durch die Aufzählung der Tage Montag bis Sonntag definiert wird.

Ausschlaggebend für die Wahl von RacerPro waren schließlich die Kontakte der Universität zu den RacerPro Entwicklern und der damit einhergehende Support für den OWL-Reasoner.

OWL-Reasoner	R1	R2	R3	R4	R5	R6
FaCT++	ja	ja	nein	ja	ja	ja
Pellet	ja	ja	ja	ja	ja	ja
RacerPro	ja	ja	ja	ja	ja	ja

Tabelle 5.1: Übersicht über die erfüllten Anforderungen (OWL-Reasoner)

5.2 Native XML-Datenbanksysteme

In diesem Unterkapitel werden einige XML-Datenbanksysteme, welche der Definition eines nativen XML-Datenbanksystems aus Kapitel 3.3 entsprechen, kurz vorgestellt. Dabei wird untersucht, ob sie den Anforderungen aus Kapitel 4 entsprechen. Anschließend werden die Ergebnisse der Untersuchungen gegenübergestellt und das geeignetste Datenbanksystem ausgewählt.

5.2.1 Apache Xindice

Das Apache Xindice [Fou07] XML-Datenbanksystem wurde in den Jahren 2001 bis 2007 von der Apache Foundation entwickelt und liegt momentan in der Version 1.1 (Release 09.05.2007) vor. Das Datenbanksystem ist unter der Apache Licence[6] Version 2.0 verfügbar[7] (R8/R9) und ist ein natives XML-Datenbanksystem (R7). In dieser Version wird XPath als Anfragesprache und XUpdate zur Veränderung von Daten unterstützt (R13). Xindice ist nicht für die Verwaltung einzelner Riesendokumente, sondern für die Verwaltung vieler kleiner bis mittelgroßer Dokumente (< 5 MB) vorgesehen (R14). Das Datenbanksystem wird standardmäßig im Server-Modus ausgeführt, damit von jedem Punkt des Netzwerks darauf zugegriffen werden kann (R12). Das Datenbanksystem ist in Java geschrieben, was eine Plattformunabhängigkeit mit sich bringt. Um Xindice auszuführen wird mindestens ein Java Development Kit (JDK) der Version 1.3 benötigt.

Bei der Analyse der Datenbankarchitektur zeigte sich, dass zum Parsen von XML-Dokumenten der Xerxes XML-Parser von Apache verwendet wird. Bei der weiteren

[6] http://xml.apache.org/xindice/license.html
[7] http://xml.apache.org/xindice/download.cgi

Analyse zeigte sich allerdings, dass es nicht vorgesehen ist, XML-Dokumente, die in das Datenbanksystem eingefügt werden, automatisch von dem Parser validieren zu lassen (nicht-R10). Es ist anzunehmen, dass das vorliegende Datenbanksystem nicht gut für die Erweiterung auf OWL-Dokumente geeignet ist, da es einerseits keine Validierung vorsieht und andererseits für kleine bis mittelgroße Dokumente entwickelt wurde, wohingegen OWL-Dokumente sehr groß werden können.

5.2.2 XML Transaction Coordinator (XTC)

Bei dem XTC[8] [XTC08] handelt es sich um ein, an der Universität Kaiserslautern entwickeltes Forschungsdatenbanksystem, welches in den letzten Jahren unter der Leitung von Theo Härder, Karsten Schmidt, Christian Mathis und Michael Haustein entwickelt wurde. Das Datenbanksystem liegt zurzeit in der Version 0.12.4 vor und stellt die Anfragesprache XQuery zur Verfügung. Dabei ist zu beachten, dass XQuery nicht vollständig implementiert ist. Außerdem ergab ein Test von XTC, dass keine Schemavalidierung von XML-Dokumenten möglich ist (nicht-R10).

5.2.3 BaseX

Das BaseX Datenbanksystem [Grü09] wird von Christian Grün und Alexander Holupirek, in der DBIS[9] Forschungsgruppe der Universität Konstanz unter der Leitung von Prof. Marc H. Scholl, entwickelt. BaseX ist ein natives XML-Datenbanksystem (R7), das als Open-Source-Projekt entwickelt wird (R8/R9). In der aktuellen Version 5.5 (Release 23.03.2009) wurde die XPath 1.0 Implementierung zu Gunsten von XQuery, welches alle XPath Optimierungen unterstützt, entfernt (R13). Die aktuelle XQuery Implementierung erreicht bei der XQuery Test-Suite (XQTS) des W3C[10] einen Punktestand von 99,9%. BaseX ist vollständig in Java implementiert und benötigt eine Java Version von 1.5 oder höher.

Das Datenbanksystem wird standardmäßig mit einer grafischen Oberfläche (GUI[11]) gestartet, die es erlaubt eine grafische Darstellung des XML-Dokuments anzeigen zu lassen und in dieser mit der Maus zu navigieren, um sich einzelne Elemente detaillierter anzeigen zu lassen. Alternativ zu der GUI ist auch ein Client-/Serverbetrieb möglich (R12). XQueries lassen sich in der GUI direkt eingeben und das Ergebnis wird sofort angezeigt. Auf Wunsch kann eine Statistik zm aktuellen Query eingeblendet werden. Das Datenbanksystem ist auch für sehr große XML-Dokumente ausgelegt (R14).

[8] Zu finden unter: www.xtc-project.de/
[9] http://www.informatik.uni-konstanz.de/arbeitsgruppen/dbis
[10] http://www.w3.org/XML/Query/test-suite/
[11] Abkürzung für: Graphical User Interface

Dieses Datenbanksystem bietet keinen eigenen Befehl an, um XML-Dokumente, die sich schon in dem Datenbanksystem befinden, zu validieren. Bei der Konfiguration des XML-Parsers im Quellcode wird die XML-Validierung explizit deaktiviert (nicht-R10). Da anzunehmen ist, dass bei diesem Datenbanksystem ein hoher Arbeitsaufwand mit der Umstellung auf OWL-Dokumente einhergehen würde, ist sie für die vorliegende Arbeit ungeeignet.

5.2.4 eXist

Die Entwicklung des eXist[12] Datenbanksystems begann im Jahr 2000 durch Wolfgang Meier, der bis heute Projektleiter und Hauptverantwortlicher für die Datenbankkernentwicklung ist. Das Datenbanksystem steht unter der GNU LGPL[13] Lizenz zum Download zur Verfügung (R8/R9). Es handelt sich bei eXist um ein natives XML-Datenbanksystem (R7). Das eXist Datenbanksystem liegt zum Zeitpunkt dieser Arbeit in der Version 1.4 (Release 08.09.2009) vor.

Ein großer Vorteil von eXist ist, dass es mehrere Modi zur Validierung zur Verfügung stellt (R10). In der Konfigurationsdatei des Datenbanksystems existiert bei der Validierungskonfiguration ein Parameter `mode`. Über diesen Parameter kann die Validierung beim Einfügen von Dokumenten ein- oder ausgeschaltet werden.

Unabhängig vom Einfügen können in eXist die Dokumente, die sich im Datenbanksystem befinden explizit validiert werden. Dies geschieht entweder in einer XQuery Anfrage oder manuell (z.B. in der GUI). Es existieren drei Operationsmodi des Datenbanksystems. Die erste Möglichkeit ist das Datenbanksystem im Server-Modus (R12) zu starten, was den Vorteil bietet, dass mehrere Personen gleichzeitig und von verschiedenen Rechnern auf das Datenbanksystem zugreifen können. Eine weitere Möglichkeit ist das Datenbanksystem im Embedded-Modus auszuführen, wobei das Datenbanksystem in eine andere Anwendung integriert wird (z.B. Client), von der aus auf die Datenbank zugegriffen werden kann. Die letzte Möglichkeit ist das Datenbanksystem in ein „Web Application Archive“ zu packen und anschließend beispielsweise an einen Tomcat zu übergeben.

Wie jedes Datenbanksystem bietet auch eXist die Option, Anfragen an die beinhalteten Daten zu stellen. Im Fall von eXist kann dies mit Hilfe von XPath 2.0 oder XQuery 1.0 Ausdrücken geschehen (R13). Die Funktionalität der im Datenbanksystem enthaltenen XQuery Implementierung deckt sich so gut mit dem W3C Standard von XQuery, dass damit in der aktuellen XQuery Test-Suite[14] (XQTS) 99,4% der Punkte erreicht werden. Zusätzlich zu der Standardfunktionalität von XQuery stellt eXist mit der

[12] http://exist.sourceforge.net/
[13] http://www.gnu.org/copyleft/lesser.html
[14] http://www.w3.org/XML/Query/test-suite/

XQuery-Update-Extension eine Erweiterung zur Verfügung, mit der die Daten im Datenbanksystem, ähnlich wie mit SQL, verändert werden können. Das Schlüsselwort dabei ist „update“ gefolgt von „insert“, „replace“, „value“, „delete“ oder „rename“. Die XQuery-Update-Extension weist Ähnlichkeit mit der in Kapitel 3.2 vorgestellten XQuery Update Facility 1.0 auf und soll in einer der nächsten Versionen von eXist daran angeglichen werden.

Das eXist Datenbanksystem hat wie alle Datenbanksysteme, die in Java implementiert werden, den Vorteil, dass es eine Plattformunabhängigkeit besitzt. Um das Datenbanksystem verwenden zu können, ist außer dem Datenbanksystem und dem Betriebssystem, ein installiertes Java Development Kit (JDK) nötig. Dieses muss mindestens in der Version 1.4.2. vorliegen [eXi09].

Zum Parsen/Validieren der XML-Dokumente wird Apache Xerxes 2 verwendet. Bei der Analyse des Quelltextes zeigte sich, dass die Validierung von XML-Dokumenten, die im Datenbanksystem enthalten sind, in einem eigenen Paket gekapselt ist. Im Fall der automatischen Validierung beim Einfügen ist zwar keine Kapselung in eine eigene Klasse gegeben, sondern die Validierung der Dokumente wird in einer eigens dafür vorgesehenen Methode erledigt (R11).

Aus Benutzersicht besitzt das eXist Datenbanksystem eine hierarchische Struktur von Collections. Eine Collection kann Dokumente oder Collections enthalten. Diese Struktur ist mit der eines Dateisystems vergleichbar, wobei Collections den Verzeichnissen im Dateisystem entsprechen, während die Dokumente den Dateien entsprechen.

5.2.5 Übersicht

Die Tabelle 5.2 gibt einen Überblick darüber, welche Anforderungen von den einzelnen Datenbanksystemen erfüllt werden. Das XTC Datenbanksystem unterstützt keine Validierung von XML-Dokumenten, wodurch eine einfache Erweiterung im Rahmen der vorliegenden Arbeit ausgeschlossen wird. Apache Xindice hat den Nachteil, dass es in der Standardversion für kleine XML-Dokumente ausgelegt ist und bei großen Dokumenten Fehler auftreten können. Zusätzlich ist in Apache Xindice genau wie bei BaseX keine Validierung der XML-Dokumente möglich. Damit ist eXist das einzige Datenbanksystem, das XML-Dokumente validieren kann (R10). Dieser Punkt ist ausschlaggebend dafür, dass eXist als XML-Datenbanksystem für diese Arbeit eingesetzt wird, auch wenn die Validierungseinheit nicht komplett gekapselt ist.

Datenbanksystem	R7	R8	R9	**R10**	R11	R12	R13	R14
Apache Xindice	ja	ja	ja	**nein**	nein	ja	ja	nein
XTC	ja	ja	nein	**nein**	nein	ja	teilweise	nein
BaseX	ja	ja	ja	**nein**	nein	ja	ja	ja
eXist	ja	ja	ja	**ja**	teilweise	ja	ja	ja

Tabelle 5.2: Übersicht über die erfüllten Anforderungen (XML-Datenbanksysteme)

5.3 Zusammenfassung

Die Untersuchung der nativen XML-Datenbanksysteme lieferte „eXist“ als beste Grundlage für die Umsetzung des Projektes. Bei der Untersuchung des OWL-Reasoners ergab sich „RacerPro“ als beste Wahl. Nachdem die Softwarebausteine, die verwendet werden sollen ausgewählt sind, ist es notwendig diese auf ihre technischen Möglichkeiten hin zu untersuchen, um entscheiden zu können, welches die beste Methode ist, um die beiden Komponenten miteinander zu verbinden, damit am Ende die gewünschte Funktionalität zur Verfügung steht.

6 Grobanalyse

Im folgenden Kapitel werden die Ergebnisse der groben Analyse des nativen XML-Datenbanksystems dargestellt. Zunächst werden die drei Anwendungsmöglichkeiten des Datenbanksystems skizziert und die in den einzelnen Fällen vorhandenen Datenbankschnittstellen aufgezählt. Anschließend wird grob skizziert, welcher Pfad von den Datenbankschnittstellen aus beschritten wird, um Dokumente zu validieren. Daraufhin wird der im Datenbanksystem enthaltene Triggermechanismus erklärt.

6.1 Schnittstellen

Das eXist Datenbanksystem bietet, wie in Abschnitt 5.2.4 erklärt, drei Anwendungsmöglichkeiten. Der erste Anwendungsfall ist die Verwendung von eXist im Server-Modus. Dabei läuft das Datenbanksystem als eigenständige Anwendung und ein Benutzer oder eine andere Anwendung kann über eine Netzwerkverbindung am Port 8080 auf sie zugreifen. EXist stellt nicht nur eine, sondern mehrere Netzwerkschnittstellen zur Verfügung. Die vorhandenen Schnittstellen sind, wie aus Abbildung 6.1 ersichtlich, REST, XML-RPC, SOAP, AtomServices und WebDAV. Wenn eXist im Server-Modus betrieben wird, können mehrere Benutzer über das Netzwerk auf das Datenbanksystem zugreifen.

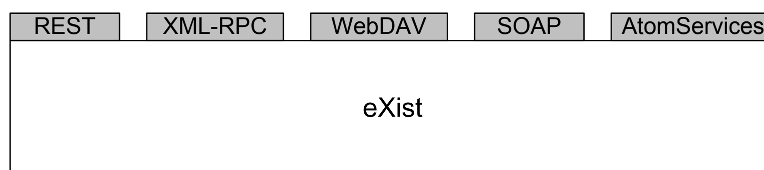


Abbildung 6.1: Netzwerkschnittstellen des Datenbanksystems

Im zweiten Fall wird das Datenbanksystem in eine andere Anwendung integriert. Für die Integration stellt eXist eine Java Schnittstelle zur Verfügung. Die verwendete Java Schnittstelle heißt XML:DB API und ermöglicht den Zugriff auf die vollständige Funktionalität des Datenbanksystems. Diese Schnittstelle kann verwendet werden, wenn das Datenbanksystem fest in eine Anwendung integriert werden soll.

Bei der letzten Möglichkeit wird das Datenbanksystem als Web Anwendung in ein „Web Application Archive“ (WAR) gepackt und anschließend an einen Tomcat oder Jetty übergeben. Dieser letzte Fall existiert, wurde im Rahmen der vorliegenden Arbeit jedoch nicht betrachtet.

6.2 Zugriffspfade

Das Datenbanksystem unterscheidet zwischen der impliziten Validierung und expliziten Validierung. Implizite Validierung bedeutet, dass Dokumente die in das Datenbanksystem eingefügt werden, automatisch auf ihre Validität geprüft werden. Explizite Validierung bedeutet, dass die Validierung entweder direkt vom Benutzer oder durch eine XQuery Anfrage ausgelöst wird.

6.2.1 Implizite Validierung

Die Abbildung 6.2 bezieht sich auf den Fall der impliziten Validierung. Dabei zeigt sich, dass die Validierung der Dokumente unabhängig davon, ob über Netzwerkschnittstellen auf das Datenbanksystem zugegriffen wird, oder ob über die XML:DB API, im Embedded-Modus, auf das Datenbanksystem zugegriffen wird, in der Klasse `Collection` ausgelöst wird.

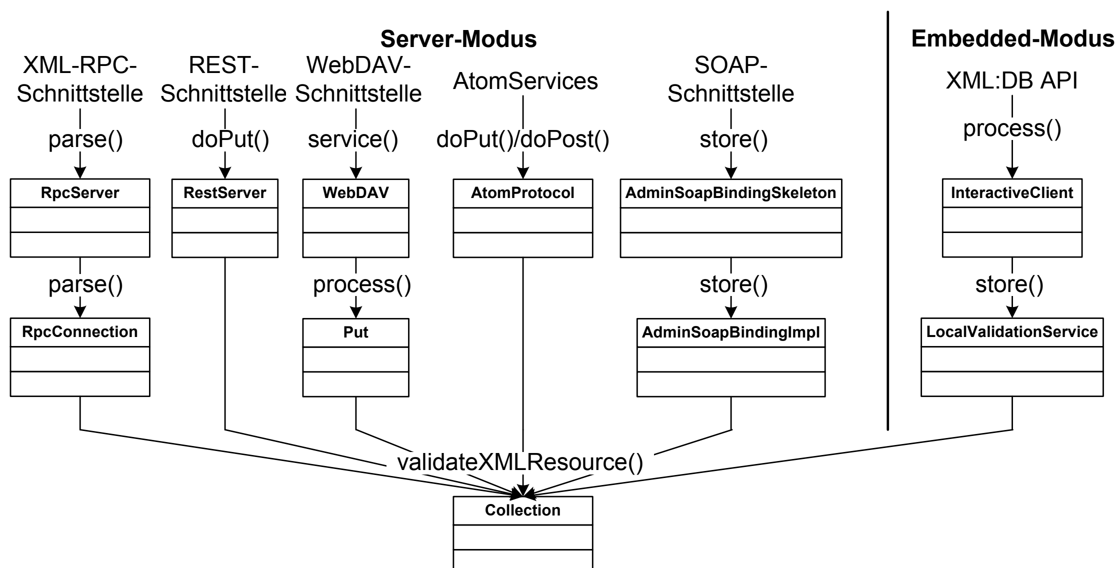


Abbildung 6.2: Zugriffspfade zur Klasse Collection

6.2.2 Explizite Validierung

Die Abbildung 6.3 bezieht sich auf die explizite Validierung. Wie bei der impliziten Validierung führen alle Pfade in eine Klasse. In diesem Fall ist dies die Klasse `Validator`. Diese Validierung kann vom Benutzer über die XML-RPC-Schnittstelle oder die XML:DB API veranlasst werden. Alternativ kann diese Validierung durch ein XQuery gestartet werden. Für die Ausführung von XQuery Anfragen existiert keine eigene Schnittstelle, sondern die Anfragen werden über die vorhandenen Datenbankschnittstellen übertragen. Die Validierung durch ein XQuery kann folglich über jede vorhandene Datenbankschnittstelle veranlasst werden, indem das XQuery zur Ausführung an das Datenbanksystem übermittelt wird.

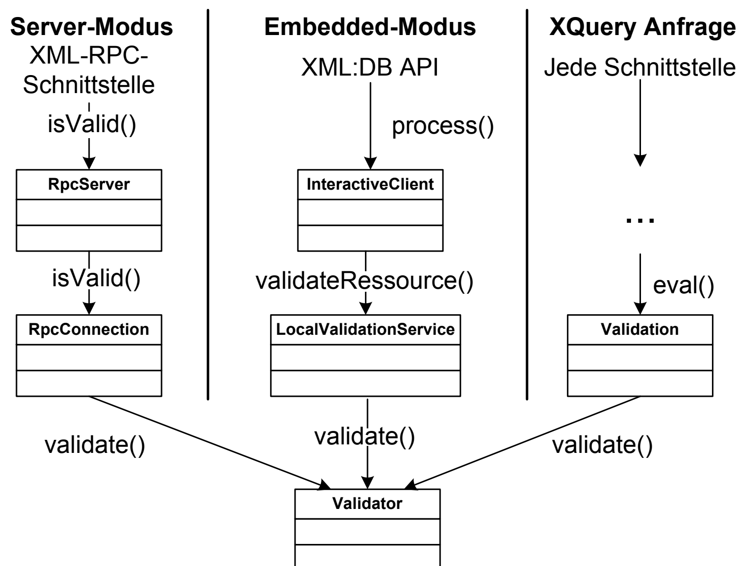


Abbildung 6.3: Zugriffspfade zur Validatorklasse

6.3 Trigger

Das eXist Datenbanksystem stellt wie viele relationale Datenbanksysteme Trigger[1] zur Verfügung. Es existieren zwei verschiedene Arten von Triggern. Einerseits gibt es XQuery-Trigger, andererseits Java-Trigger. Die Trigger sind jeweils in der, an der Namensgebung beteiligten „Sprache“ verfasst. Damit ein Trigger vom Datenbanksystem ausgeführt wird, muss der entsprechende Trigger in die Konfiguration

[1] Unter Triggern werden Mechanismen, die bei genau spezifizierten Ereignissen in dem Datenbanksystem ausgeführt werden und eine bestimmte Aktion ausführen, verstanden

der entsprechenden Collection (siehe 8.3.2), oder einer Eltern-Collection eingetragen werden. Ein Eintrag sieht folgendermaßen aus:

```
<collection xmlns="http://exist-db.org/collection-config/1.0">
   <triggers>
      <trigger event="update" class="org.exist.collections.triggers.HistoryTrigger">
         <parameter ... />
      </trigger>
   </triggers>
</collection>
```

Listing 6.1: Collection Konfigurationsdatei für einen Trigger

Alle Trigger, die in der Konfigurationsdatei definiert werden, sind in dem `triggers`-Element geschachtelt. In der dritten Zeile wird angegeben, welcher Trigger ausgelöst werden soll und bei welchem Anlass. In diesem Fall soll der HistoryTrigger jedes Mal ausgeführt werden, wenn ein Dokument in der aktuellen Collection oder einer Unter-Collection verändert wird. Über das Element `parameter` können Werte übergeben werden, auf die bei der Ausführung des Triggers zugegriffen werden kann. Der Trigger wird je nachdem, ob er für ein Dokument-Ereignis (insert, update, delete) oder ein Collection-Ereignis (create, rename, delete) konfiguriert ist, für jedes Dokument/Collection einzeln aufgerufen.

Jeder Trigger wird zweimal aufgerufen. Das erste Mal in der Vorbereitungsphase, bevor Änderungen vorgenommen werden und das zweite Mal in der Abschlussphase, wenn die Änderungen durchgeführt sind, die Transaktion aber noch nicht abgeschlossen ist. Die Entscheidung, welche Aktionen in welcher dieser beiden Phasen ausgeführt werden, wird nicht in der im Datenbanksystem enthaltenen Konfigurationsdatei getroffen. Diese Entscheidung wird im entsprechenden Trigger festgelegt.

Die Triggermechanismen von eXist wurden betrachtet, da das Datenbanksystem die implizite Validierung nur durchführt, wenn Dokumente in das Datenbanksystem eingefügt werden. Wenn Dokumente, die in dem Datenbanksystem enthalten sind durch einen XQuery-Update-Extension Ausdruck oder einen XUpdate Ausdruck verändert werden, wird keine Validierung durchgeführt. Die Idee ist, dass die Validierung in den eben genannten Fällen durch die Entwicklung eines geeigneten Triggers ergänzt werden kann.

6.4 Zusammenfassung

In diesem Abschnitt wurde zunächst gezeigt, in welchen Modi das Datenbanksystem ausgeführt werden kann und welche Schnittstellen in den unterschiedlichen Modi

verfügbar sind. Anschließend wurde verdeutlicht, dass die implizite Validierung, unabhängig davon, wie das Dokument eingefügt wird in der Klasse `Collection` veranlasst wird. Die explizite Validierung von Dokumenten wird für alle Pfade in der Klasse `Validator` angestoßen. Abschließend folgte eine Erklärung der im Datenbanksystem vorhandenen Trigger.

7 Grobentwurf

Im folgenden Kapitel wird grob skizziert, worauf bei einer Erweiterung eines XML-Datenbanksystems um die Konsistenzprüfung von OWL-Dokumenten, geachtet werden muss. Anschließend wird betrachtet, welche unterschiedlichen Fälle es bei einer Konsistenzprüfung von OWL-Dokumenten geben kann. Daraufhin wird untersucht, welche Übergabemethoden von OWL-Dokumenten an den OWL-Reasoner denkbar sind. Abschließend wird ein OWL-Modul entworfen, welches das Verbindungsstück zwischen dem Datenbanksystem und dem OWL-Reasoner darstellt.

7.1 Unterscheidung XML/OWL-Dokumente

Das XML-Datenbanksystem soll um die Funktionalität der logischen Validierung von Dokumenten anhand von Ontologien erweitert werden. Gleichzeitig soll im Datenbanksystem weiterhin die syntaktische Validierung von XML-Dokumenten möglich sein. Daher muss es eine Möglichkeit geben diese beiden Dokumenttypen zu unterscheiden, da die Prüfung der Gültigkeit des Dokumentes sonst ein falsches Ergebnis liefert. Zur Unterscheidung reicht es nicht aus, wenn geprüft wird, ob ein Dokument XML-Syntax enthält, da jedes OWL-Dokument gleichzeitig ein wohlgeformtes XML-Dokument sein muss. Dies ist der Fall, da OWL wie in Abschnitt 3.6 erklärt die XML-Syntax verwendet um die Ontologien zu modellieren. Der Unterschied zwischen den XML- und OWL-Dokumenten liegt in der Art der Validierung. Wie bereits erwähnt werden XML-Dokumente syntaktisch validiert, das heißt ein XML-Dokument wird mit Hilfe eines XML-Parsers gegen eine in einer Schemasprache verfasste Grammatik geprüft. Diese Grammatik gibt an, wie das Dokument aufgebaut sein darf. OWL-Dokumente müssen logisch validiert werden. Bei der logischen Validierung geht es nicht um den Aufbau des Dokumentes, sondern darum, dass die Aussagen in dem Dokument zu den definierten Konzepten passen. Die Unterscheidung zwischen den XML- und OWL-Dokumenten zur Laufzeit bedeutet einen deutlichen Mehraufwand, da zur Unterscheidung jedes Dokument beim Zugriff analysiert werden muss. Daher wird beim Start des Datenbanksystems mit Hilfe der Konfigurationsdatei festgelegt, ob das Datenbanksystem im XML- oder im OWL-Modus ausgeführt wird. Abhängig von dem ausgewählten Modus wird entweder die Schemavalidierung von XML-Dokumenten oder die Konsistenzprüfung von OWL-Dokumenten ausgeführt.

7.2 Konsistenzprüfung

In diesem Abschnitt wird erklärt, wie das XML-Datenbanksystem und der OWL-Reasoner interagieren, wenn die Konsistenz eines Dokumentes geprüft wird. Außerdem werden die Konsequenzen, die das Ergebnis des Konsistenztests auf die untersuchten Dokumente hat, erläutert. Auch hier wird berücksichtigt, dass eXist zwischen der impliziten und expliziten Validierung (Konsistenzprüfung) unterscheidet. Im Fall der impliziten Konsistenzprüfung existieren zwei Fälle. Die Abbildung 7.1 stellt einen Entscheidungsbaum für die beiden Fälle der impliziten Konsistenzprüfung dar. Im Entscheidungsbaum muss entschieden werden, ob durch das Einfügen des Dokumentes ein anderes, welches sich bereits in dem Datenbanksystem befindet überschrieben wird. Bei der expliziten Validierung ist keine Fallunterscheidung notwendig, da von einem Benutzer oder einem Programm spezifiziert wurde, welches Dokument geprüft werden soll.

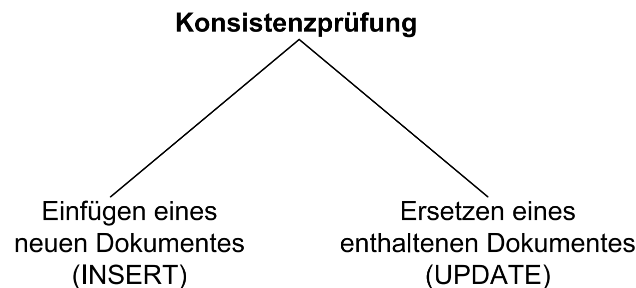


Abbildung 7.1: Entscheidungsbaum zur impliziten Konsistenzprüfung

7.2.1 Implizite Validierung

In diesem Abschnitt werden die beiden Fälle, zwischen denen in Abbildung 7.1 unterschieden wird, genauer beschrieben. Dabei wird zuerst auf INSERT und dann auf UPDATE eingegangen.

INSERT

Wenn ein neues Dokument in das Datenbanksystem eingefügt werden soll, muss es auf seine Konsistenz überprüft werden und darf erst dann in das Datenbanksystem eingefügt werden, wenn der Konsistenztest ergeben hat, dass es sich um ein konsistentes Dokument handelt. Handelt es sich nicht um ein konsistentes Dokument, wird es verworfen. Um das Dokument auf seine Konsistenz prüfen zu lassen, wird es von dem

XML-Datenbanksystem an den OWL-Reasoner übergeben. Für diese Übergabe gibt es verschiedene Möglichkeiten, die in Kapitel 3.8 und 3.9 beschrieben wurden. Nachdem das Dokument an den OWL-Reasoner übergeben wurde, fragt das Datenbanksystem nach, ob es konsistent ist. Das Ergebnis dieser Anfrage wird vom OWL-Reasoner an das Datenbanksystem übergeben. Das Datenbanksystem fügt das Dokument nur ein, wenn es konsistent ist.

UPDATE

In diesem Fall soll ein Dokument in das Datenbanksystem eingefügt werden, das ein anderes Dokument im Datenbanksystem überschreibt. Bevor das Dokument eingefügt werden darf, muss es auf seine Konsistenz geprüft werden. Dazu wird das Dokument, welches sich im Speicher befindet, an den OWL-Reasoner übergeben. Die Konsistenzprüfung läuft analog zum vorherigen Fall ab. Das Datenbanksystem wertet das Ergebnis des Konsistenztests aus. Ist das neue Dokument inkonsistent, dann wird es verworfen und das Dokument im Datenbanksystem, welches überschrieben werden sollte, wird nicht verändert. Ist das Dokument konsistent, wird es in das Datenbanksystem eingefügt und überschreibt das alte Dokument.

7.2.2 Explizite Validierung

Bei der expliziten Validierung soll die Konsistenzprüfung für ein Dokument, welches bereits im Datenbanksystem enthalten ist durchgeführt werden. Zu Beginn wird das Dokument aus dem Datenbanksystem in den Speicher geladen, um es an den OWL-Reasoner übergeben zu können. Die Kommunikation zwischen dem XML-Datenbanksystem und dem OWL-Reasoner läuft wie bei der impliziten Validierung ab. Die Auswertung des Ergebnisses des Konsistenztests wird an den Benutzer, oder das Programm, welches den Konsistenztest veranlasst hat, zurückgegeben. Das Datenbanksystem zieht aus dem Ergebnis keine Konsequenzen, dies obliegt alleine dem Benutzer/Programm, der das Ergebnis bekommen hat.

7.3 Verschiedene Übergabemethoden an den OWL-Reasoner

Es existiert kein fester Standard, der vorgibt, wie Dokumente an einen OWL-Reasoner übergeben werden. Aus diesem Grund werden im Folgenden drei Möglichkeiten betrachtet, die skizzieren, wie eine Übergabe eines Dokumentes an den OWL-Reasoner ablaufen könnte. Die Szenarien sind an Mechanismen angelehnt, die von einem oder

mehreren existierenden OWL-Reasonern angeboten werden. Wie die Abbildung 7.2 zeigt wird die Dateiübergabe, die Netzwerkübergabe und die Speicherübergabe von Dokumenten betrachtet.

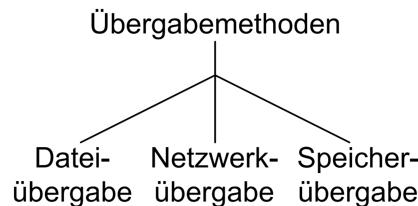


Abbildung 7.2: Übersicht über Übergabemethoden

7.3.1 Dateiübergabe

Die Dokumentübergabe und Kommunikation mit dem OWL-Reasoner funktioniert in diesem Fall so wie in Abbildung 7.3 dargestellt. Im ersten Schritt wird eine Netzwerkverbindung zu dem OWL-Reasoner aufgebaut. Diese Netzwerkschnittstelle dient nicht zur Übertragung des Dokumentes, sondern als Managementschnittstelle, über welche Managementkommandos an den OWL-Reasoner übermittelt werden können. In diesem Szenario wird das Dokument nicht über die Netzwerkschnittstelle von dem XML-Datenbanksystem an den OWL-Reasoner übergeben. Das Dokument wird, in eine temporäre Datei auf die Festplatte geschrieben. Das Datenbanksystem übermittelt an die Managementschnittstelle des OWL-Reasoners das Kommando die temporäre Datei einzulesen. Der OWL-Reasoner antwortet dem XML-Datenbanksystem mit dem Resultat des Befehls. Wenn die Datei vom OWL-Reasoner erfolgreich eingelesen wurde, schickt das Datenbanksystem das Kommando, das Dokument (T-Box und A-Box) auf seine Konsistenz zu prüfen an den OWL-Reasoner. Dieser führt das Kommando aus und antwortet mit dem Ergebnis der Ausführung auf das Kommando. Wenn dieser Rückgabewert ausgewertet ist und die daraus folgenden Konsequenzen gezogen sind, ist das Dateiübergabeszenario beendet. Nach der Ausführung der Dateiübergabe wird die temporäre Datei gelöscht und die Netzwerkverbindung abgebaut.

Die von RacerPro angebotene nRQL-Schnittstelle bietet diese Funktionalität. Sie funktioniert wie in Kapitel 3.9 beschrieben. Der große Nachteil der Dateiübergabemethode ist offensichtlich, da sowohl das Datenbanksystem, als auch der OWL-Reasoner Zugriff auf dasselbe Dateisystem haben müssen. Außerdem ist das Schreiben und Lesen von der Festplatte sehr zeitaufwändig.

Bei dieser Schnittstelle wird von den RacerPro Entwicklern ein Modul angeboten, das sich um den eigentlichen Verbindungsaufbau und die Kommunikation kümmert.

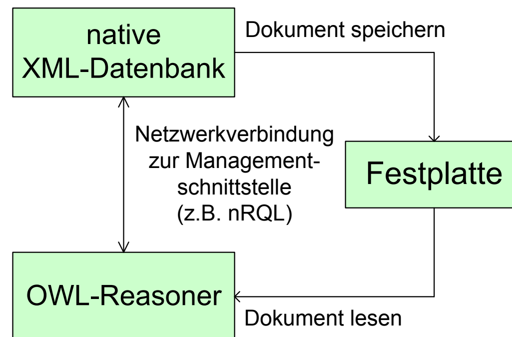


Abbildung 7.3: Dateiübergabe von Dokumenten an den OWL-Reasoner

Das Modul heißt `jracer` und bietet eine Java API, mit der er in beliebige Java Projekte eingebunden werden kann, beispielsweise in eXist. In der Abbildung 7.4 ist dargestellt, wie die Zuständigkeit verteilt ist, wenn eXist mit `jracer` kombiniert wird. Die Netzwerkkommunikation wird vollständig von `jracer` erledigt, während das Ablegen des Dokumentes weiterhin in den Zuständigkeitsbereich von eXist fällt.

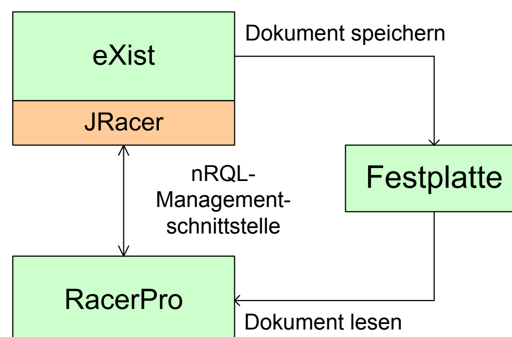


Abbildung 7.4: Dateiübergabe an RacerPro mit jracer

7.3.2 Dokumentübergabe über eine Netzwerkverbindung

Die Übergabe der Dokumente und die Kommunikation mit dem OWL-Reasoner funktioniert wie in der Abbildung 7.5 dargestellt. Dieses Szenario beginnt damit, dass das XML-Datenbanksystem eine Netzwerkverbindung zum OWL-Reasoner aufbaut, welche für die weitere Kommunikation verwendet wird. Über diese Verbindung wird, nach erfolgreichem Aufbau, das Dokument per HTTP PUT an den OWL-Reasoner übertragen. Nachdem der OWL-Reasoner das erfolgreiche Laden der Datei bestätigt hat, wird ein HTTP POST über die Netzwerkverbindung übertragen, welches die

Anfrage nach der Konsistenz (T-Box und A-Box) des Dokumentes enthält. Das Ergebnis des Konsistenztests wird dem XML-Datenbanksystem in der Antwort auf die HTTP POST Anfrage mitgeteilt.

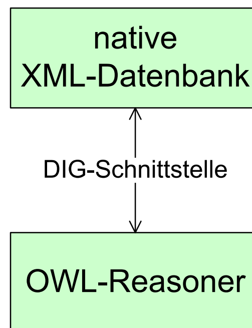


Abbildung 7.5: Netzwerkübergabe von Dokumenten

Diese Art der Dokumentübergabe ist über die DIG-Schnittstelle möglich. Die DIG-Schnittstelle wurde in Kapitel 3.8 beschrieben. Wie dort erläutert ist ein Problem dieser Schnittstelle, dass Dokumente, die übertragen werden sollen, in die, dem DIG-Schema entsprechende Syntax umgewandelt werden müssen. Die OWL-API enthält Klassen, mit denen eine Umwandlung von OWL-DL nach DIG 1.1 möglich ist. Mit dieser DIG-Syntax ist in der aktuellen Version (1.1) nicht die gesamte OWL-DL-Syntax darstellbar.

7.3.3 Speicherübergabe

Die Abbildung 7.6 stellt dar, wie der OWL-Reasoner über eine Java API als Bibliothek in das Datenbanksystem integriert werden könnte. Für einen Konsistenztest wird einer Instanz des OWL-Reasoners das in den Hauptspeicher geladene Dokument übergeben. Anschließend wird eine Methode aufgerufen, welche die Konsistenz (T-Box und A-Box) des Dokumentes prüft (z.B. consistencycheck()) und das Ergebnis als Rückgabewert liefert. Sollte an einer Stelle dieses Szenarios ein Fehler auftreten, so wird eine Fehlermeldung erzeugt, die dem Benutzer angezeigt wird. Eine Möglichkeit einen OWL-Reasoner über eine Java API in eine Anwendung einzubinden bietet die OWL-API[1]. In der Version 1.9.0 bietet der OWL-Reasoner RacerPro, welcher für die Konsistenzprüfung verwendet wird, keine OWL-API.

[1]Zu finden unter: http://owlapi.sourceforge.net/

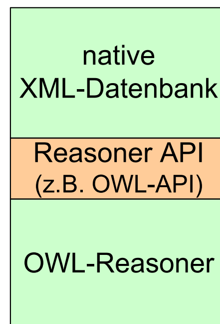


Abbildung 7.6: Speicherübergabe von Dokumenten

7.3.4 Übersicht

Es stehen drei Ansätze zur Verfügung, mit denen ein OWL-Reasoner an ein Datenbanksystem angebunden werden kann. Bei der geplanten Verwendung von RacerPro als OWL-Reasoner sind nur die ersten beiden Ansätze möglich, da die OWL API in der Version 1.9.0 noch nicht verfügbar ist.

Für die Entscheidung zwischen der DIG-Schnittstelle und der nRQL-Schnittstelle, müssen deren Vor- und Nachteile gegeneinander abgewogen werden. Die nRQL-Schnittstelle hat den Vorteil, dass sich die Resultate der Konsistenzprüfung auf das gesamte OWL-Dokument beziehen. Der Nachteil daran ist, dass die Übergabe des OWL-Dokumentes über das Dateisystem läuft. Die DIG-Schnittstelle hat den Vorteil, dass das OWL-Dokument über das Netzwerk übertragen werden kann. Dafür muss das OWL-Dokument für die Übertragung in die DIG-Syntax umgewandelt werden, die allerdings nicht dieselbe Ausdrucksmächtigkeit wie OWL-DL besitzt. Dadurch kommt es beispielsweise bei der Repräsentation von Datentypen zu Problemen, wodurch es sein kann, dass bestimmte Teile einer OWL-DL-Ontologie ignoriert werden müssen und die Konsistenzaussage sich nur auf den übrigen Teil bezieht.

Nach dem Abwägen der Vor- und Nachteile der beiden Schnittstellen fällt die Wahl auf die nRQL-Schnittstelle, die zwar eine schlechte Performanz, dafür aber eine korrekte Konsistenzaussage bietet.

7.4 OWL-Modul

Die Analyse der Zugriffspfade (siehe Abschnitt 6.2) hat ergeben, dass die implizite Validierung in der Klasse `Collection` und die explizite Validierung in der Klasse `Validator` ausgelöst wird. Damit das Datenbanksystem neben der bereits vorhandenen syntaktischen Validierung auch eine semantische Validierung (Konsistenzprüfung)

durchführen kann, müssen diese beiden Klassen erweitert werden. Da der Eingriff in das Datenbanksystem möglichst gering gehalten werden soll, wird die neue Funktionalität in ein zusätzliches Modul namens `owlconsistency` gekapselt. Dadurch ist die Änderung an dem Datenbanksystem auf einen Aufruf des Moduls `owlconsistency` in den zu verändernden Klassen beschränkt.

Im Folgenden werden zwei Entwürfe für das Modul `owlconsistency` vorgestellt. Der erste Entwurf, im Weiteren als Basisentwurf bezeichnet, unterstützt nur Dokumente, die eine T-Box und soweit vorhanden eine A-Box enthalten. Im Gegensatz dazu unterstützt der zweite Entwurf, im Weiteren als erweiterter Entwurf bezeichnet, die Aufteilung von T-Box und A-Box auf unterschiedliche Dokumente.

7.4.1 Basisentwurf

Der Basisentwurf des `owlconsistency` Moduls ermöglicht es, OWL-Dokumente an RacerPro zu übergeben und sie auf ihre Konsistenz prüfen zu lassen. Anschließend werden die Ergebnisse dieser Konsistenzprüfung ausgewertet und an die aufrufende Instanz weitergeleitet. Die OWL-Dokumente, die für diesen Entwurf betrachtet werden, enthalten eine T-Box und falls vorhanden eine A-Box.

Die Klasse `OWLChecker` stellt die Außenschnittstelle des `owlconsistency` Moduls dar. Die Abbildung 7.7 stellt dar, wie die Konsistenzprüfung intern in dem Modul `owlconsistency` abläuft.

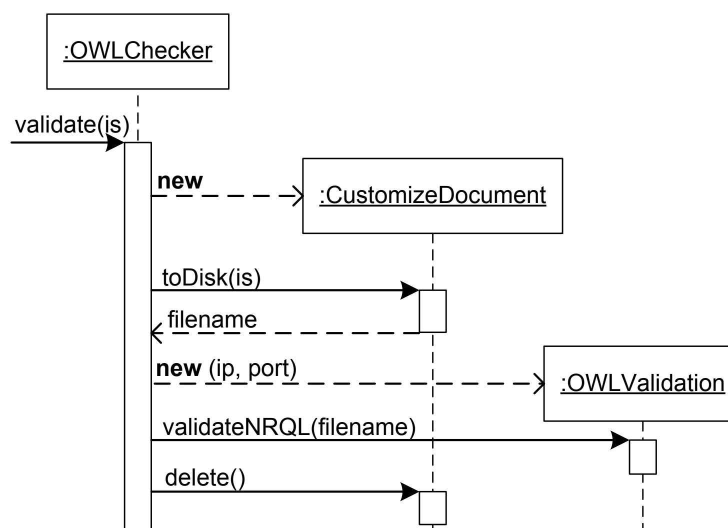


Abbildung 7.7: Einfache Darstellung der Konsistenzprüfung

Zu Beginn wird die Methode `validate` der Klasse `OWLChecker` aufgerufen und dabei das zu prüfende Dokument in Form einer `InputSource`-Instanz übergeben. Diese Instanz der Klasse `InputSource` kann einen ByteStream oder einen CharacterStream, über den auf das Dokument zugegriffen werden kann, enthalten. Alternativ kann in der Membervarible `systemId` der `InputSource`-Instanz der Pfad zu einer Datei im Dateisystem enthalten sein. In dieser Datei ist das Dokument enthalten. In der Methode `validate` wird anschließend die Konsistenzprüfung dieses Dokumentes veranlasst.

Bevor das Dokument auf seine Konsistenz geprüft wird, wird es an die Methode `toDisk` der Klasse `CustomizeDocument` übergeben. Dieses Dokument wird in dieser Klasse in eine temporäre Datei auf der Festplatte gespeichert. Der Pfad der temporären Datei wird der aufrufenden Klasse `OWLChecker` mitgeteilt.

Nachdem die Klasse `OWLChecker` den Pfad der temporären Datei bekommen hat, übergibt sie diesen an die Methode `validateNRQL` der Klasse `OWLValidation`. Außer dem Pfad werden die Verbindungsdaten von RacerPro in Form von IP-Adresse und Port an den Konstruktor der Klasse `OWLValidation` übergeben. Mit den Verbindungsdaten wird, wie in Abschnitt 7.3.1 beschrieben, über das `jracer` Modul eine Verbindung zur Managementkonsole von RacerPro aufgebaut. Über diese Verbindung wird das Einlesen und die Konsistenzprüfung des OWL-Dokumentes veranlasst. Das Ergebnis dieser Prüfung wird der Klasse `OWLValidation` von dem OWL-Reasoner mitgeteilt. Die Klasse `OWLValidation` analysiert die Ergebnisse der Konsistenzprüfung. Der aufrufenden Klasse wird das Ergebnis der Prüfung nicht explizit mitgeteilt, da die Klasse `OWLValidation` nur bei einem inkonsistenten Dokument oder einem Verbindungsfehler durch eine Fehlermeldung reagiert. Die Fehlermeldung wird direkt an die Klasse, durch welche die Konsistenzprüfung veranlasst wurde, weitergeleitet. Abschließend wird die temporäre Kopie des OWL-Dokumentes durch den Aufruf der Methode `delete` der Klasse `CustomizeDocument` von der Festplatte entfernt.

7.4.2 Erweiterter Entwurf

Der erweiterte Entwurf des Moduls `owlconsistency` bietet weiterhin die Funktionalität des Basisentwurfes. Zusätzlich ermöglicht dieser Entwurf die in Abschnitt 4.2 beschriebene Aufteilung von OWL-Dokumenten in T-Box und A-Box. Die T-/A-Box kann auf mehrere Dokumente in dem Datenbanksystem abgebildet werden.

Die Klasse `OWLChecker` stellt weiterhin die Außenschnittstelle des `owlconsistency` Moduls dar. Die Abbildung 7.8 zeigt, was bei der Konsistenzprüfung eines Dokumentes mit dem erweiterten Entwurf des Moduls `owlconsistency` passiert. Die Konsistenzprüfung wird genau wie im Basisentwurf mit dem Aufruf der Methode `validate` der

Klasse `OWLChecker` gestartet. Dieser Methode wird das OWL-Dokument, welches auf seine Konsistenz geprüft werden soll, übergeben.

Im ersten Schritt muss untersucht werden, was in dem Dokument enthalten ist. Dazu wird das Dokument an die Methode `evaluate` der Klasse `DocumentContent` übergeben. Als Ergebnis teilt diese Klasse der Klasse `OWLChecker` mit, ob das Dokument eine T-Box, eine A-Box oder beides enthält. Wenn das Dokument eine T-Box und eine A-Box enthält, läuft die weitere Konsistenzprüfung exakt so ab, wie im Basisentwuf in Abschnitt 7.4.1 beschrieben.

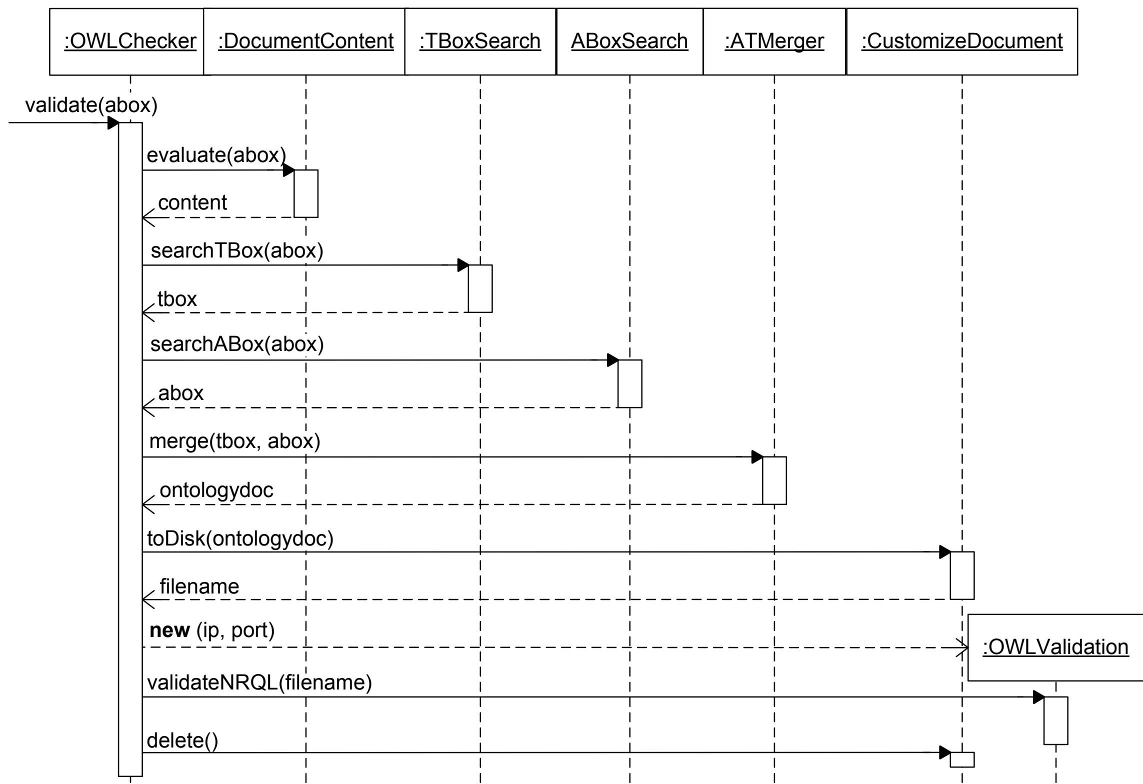


Abbildung 7.8: Erweitertes OWL-Modul

Da die Unterschiede zwischen dem Basisentwurf und dem erweiterten Entwurf gezeigt werden sollen, wird im Weiteren davon ausgegangen, dass das Dokument eine A-Box enthält. Ist dies der Fall, muss im nächsten Schritt die T-Box, zu der die A-Box gehört, gesucht werden. Dazu wird die Methode `searchTBox` der Klasse `TBoxSearch` aufgerufen. Diese Methode greift auf das Datenbanksystem zu und sucht die T-Box, die für die Collection, in der das Dokument liegt oder eingefügt werden soll, als Standard-T-Box angegeben ist. Wenn die T-Box in dem Datenbanksystem, wie in Abschnitt 4.2 beschrieben, auf mehrere Dokumente aufgespaltet ist, werden alle T-Box-Fragmente eingelesen und mit der Methode `merge` der Klasse `TBFMerger` in ein

Dokument zusammengefügt. Liegt die T-Box in einem einzigen Dokument vor, wird sie eingelesen. Das Dokument, das die T-Box enthält wird anschließend an die Klasse `OWLChecker` zurückgegeben.

Daraufhin wird das A-Box-Dokument an die Methode `searchABox` der Klasse `ABoxSearch` übergeben. Diese Methode untersucht, ob das übergebene A-Box-Dokument eine komplette A-Box oder das Fragment einer A-Box enthält. Wenn ein Fragment enthalten ist, werden alle Fragmente der A-Box geladen und anschließend mit der Methode `merge` der Klasse `ABFMerger` zu einem Dokument zusammengefügt. Das A-Box-Dokument mit der kompletten A-Box wird schließlich der aufrufenden Klasse als Rückgabewert übergeben.

Wenn statt eines A-Box-Dokumentes ein T-Box-Dokument übergeben wird, wird dies auch den Klassen `TBoxSearch` und `ABoxSearch` übergeben. Die Methode `searchTBox` der Klasse `TBoxSearch` untersucht, ob das T-Box-Dokument die gesamte T-Box oder nur ein Fragment enthält. Wenn nur ein Fragment enthalten ist, werden alle anderen Fragmente eingelesen und an die Methode `merge` der Klasse `TBFMerger` übergeben. In dieser Methode werden alle Fragmente zu einem T-Box-Dokument zusammengefasst. Das T-Box-Dokument wird an die Klasse `OWLChecker` als Rückgabewert übergeben.

Anschließend wird das T-Box-Dokument der Methode `searchABox` der Klasse `ABoxSearch` übergeben. Mit der Methode kann zu einem T-Box-Dokument die A-Box gesucht werden. Wenn die A-Box, die gesucht wird, auf mehrere Teile aufgeteilt ist, liest die Klasse `ABoxSearch` alle Teile ein und fügt sie mit der Methode `merge` der Klasse `ABFMerger` zu einem Dokument zusammen. Das A-Box-Dokument, wird anschließend als Rückgabewert an die Klasse `OWLChecker` übergeben.

Um die beiden separaten Dokumente auf ihre Konsistenz prüfen zu können, müssen sie wieder in ein Dokument zusammengefasst werden. Dazu werden das T-Box-Dokument und das A-Box-Dokument an die Methode `merge` der Klasse `ATMerger` übergeben. In dieser Klasse werden die beiden separaten OWL-Dokumente untersucht und anschließend so zu einem Dokument zusammengefasst, dass dieses neue Dokument ein wohlgeformtes XML-Dokument ist. Dieses zusammengefasste Dokument wird anschließend als Rückgabewert an die Klasse `OWLChecker` übergeben.

Nachdem die Methode `validate` der Klasse `OWLChecker` das zusammengefasste Dokument erhalten hat, läuft die übrige Konsistenzprüfung analog zu der Konsistenzprüfung im Basisentwurf ab.

7.4.3 Übersicht

Mit der Funktionalität des Basisentwurfes ist es möglich OWL-Dokumente auf ihre Konsistenz zu überprüfen. Dazu ist es allerdings notwendig, dass sowohl die T-Box

als auch die A-Box in demselben Dokument enthalten sind. Der erweiterte Entwurf bietet zusätzlich die Möglichkeit OWL-Dokumente, bei denen die T-Box und die A-Box in unterschiedlichen Dokumenten gespeichert wurden, auf ihre Konsistenz zu überprüfen. Da bereits mit der im Basisentwurf beschriebenen Funktionalität die Konsistenzprüfung von OWL-Dokumenten möglich ist und in der Aufgabenstellung nicht mehr verlangt wird, beschränkt sich die vorliegnde Arbeit auf die Weiterführung des Basisentwurfes und sieht den erweiterten Entwurf als Ausblick in die Zukunft.

7.5 Zusammenfassung

Zu Beginn wurde erläutert, wie dem Datenbanksystem eine Unterscheidung zwischen XML-Dokumenten und OWL-Dokumenten möglich ist. Dies ist nötig, da die beiden Dokumenttypen unterschiedliche Prüfungsmethoden benötigen. Im Anschluss wurden kurz verschiedene Fälle, in denen eine Konsistenzprüfung notwendig ist gezeigt und erläutert, wie diese Konsistenzprüfung abläuft. Im Folgenden wurde die Entscheidung getroffen, welcher Ansatz zur Übergabe von Dokumenten vom Datenbanksystem zum OWL-Reasoner im Rahmen der vorliegenden Arbeit verwendet werden soll. Die Wahl fiel dabei auf die nRQL-Schnittstelle. Abschließend wurden zwei Entwürfe für das OWL-Modul, in welchem die Konsistenzprüfung von OWL-Dokumenten gekapselt ist, vorgestellt.

8 Systemanalyse

Im folgenden Kapitel werden zunächst die Ergebnisse der Analyse der impliziten und expliziten Validierung von XML-Dokumenten im Datenbanksystem vorgestellt. Anschließend wird die Konfigurierbarkeit der Validierung im Datenbanksystem erklärt. Daraufhin wird erklärt, wie das Lesen eines Dokumentes aus dem Datenbanksystem intern erfolgt und wie die Triggermechanismen des Datenbanksystems funktionieren.

8.1 Implizite Validierung

Implizite Validierung bedeutet, dass XML-Dokumente auf ihre Gültigkeit geprüft werden, ohne dass dies vom Benutzer explizit aufgerufen wird. Diese Art der Validierung ist immer dann gewünscht, wenn neue Dokumente in das Datenbanksystem eingefügt werden, oder wenn Änderungen an vorhandenen Dokumenten vorgenommen werden. Die Analyse des Datenbanksystems ergab, dass bei dem eXist Datenbanksystem, nach einer Änderung eines Dokuments durch einen XQuery-Update-Extension Ausdruck oder einen XUpdate Ausdruck keine automatische Validierung durchgeführt wird.

In Abschnitt 6.2 wurde festgestellt, dass die implizite Validierung, unabhängig davon über welche Datenbankschnittstelle ein Dokument eingefügt wird, in der Klasse `Collection` veranlasst wird. Im Weiteren wird genauer auf die Pfade der REST-Schnittstelle, der XML-RPC-Schnittstelle und der OWL:DB API eingegangen.

REST-Schnittstelle

Die Abbildung 8.1 zeigt den Ablauf des Einfügevorgangs über die REST-Schnittstelle von der Methode `doPut` der Klasse `RESTServer`, bis zur Validierung des Dokumentes. Die Methode erhält als Übergabeparameter eine Instanz der Klasse `DBBroker`, sowie eine Instanz der Klasse `File`, die den Zugriff auf das einzufügende Dokument ermöglicht und den Pfad an dem das Dokument in das Datenbanksystem eingefügt werden soll, als `XmldbURI`. Zusätzlich wird eine `HttpServletRequest`- und eine `HttpServletResponse`-Instanz übergeben. Als erstes wird eine neue Transaktion für den Einfügevorgang gestartet. Anschließend wird der Pfad des Dokumentes in Collection-

Pfad und Dokumentname zerlegt. Danach wird eine Instanz der Collection, in die das Dokument eingefügt werden soll, mit der getCollection-Methode der DBBroker-Instanz angefordert. Zur Validierung des Dokumentes wird die Methode validateXML-Resource der Collection-Instanz aufgerufen. Der Methode wird die Transaktion, die DBBroker-Instanz, der Dokumentname als XmldbURI und eine InputSource-Instanz, die den Pfad der temporären Datei enthält, übergeben. Der Ablauf der validateXML-Resource-Methode wird im Zusammenhang mit Abbildung 8.4 für alle Zugriffspfade beschrieben. Nach der Validierung kann das Datenbanksystem mit dem Einfügen des Dokumentes weitermachen.

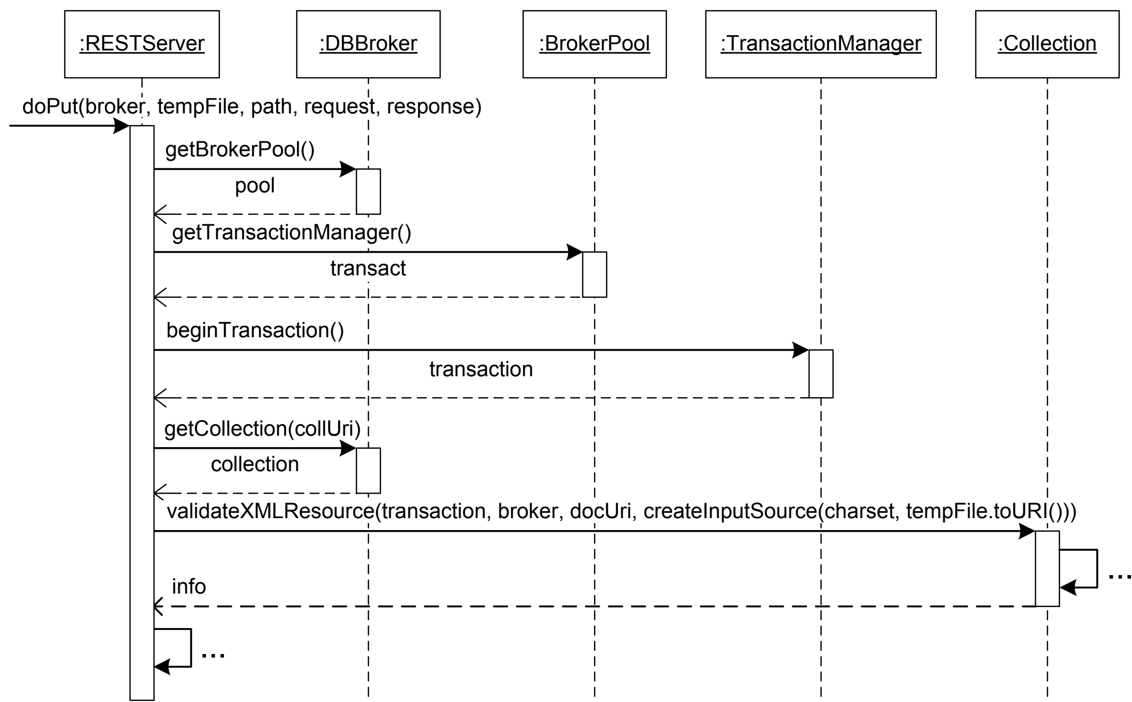


Abbildung 8.1: Einfügen über die REST-Schnittstelle

XML-RPC-Schnittstelle

Die Abbildung 8.2 zeigt den Ablauf einer Validierung, die bei der XML-RPC-Schnittstelle beim Einfügen eines Dokumentes veranlasst wird. Die Methode parse der Klasse RpcServer bekommt eine User-Instanz, den Dokumentinhalt in einem Bytearray, den Dokumentnamen als String-Instanz und eine Integer, der anzeigt ob das Dokument ein anderes überschreiben soll, übergeben. Als erstes wird vom ConnectionPool eine RpcConnection-Instanz angefordert. Von der RpcConnection-Instanz wird die Methode parse aufgerufen.

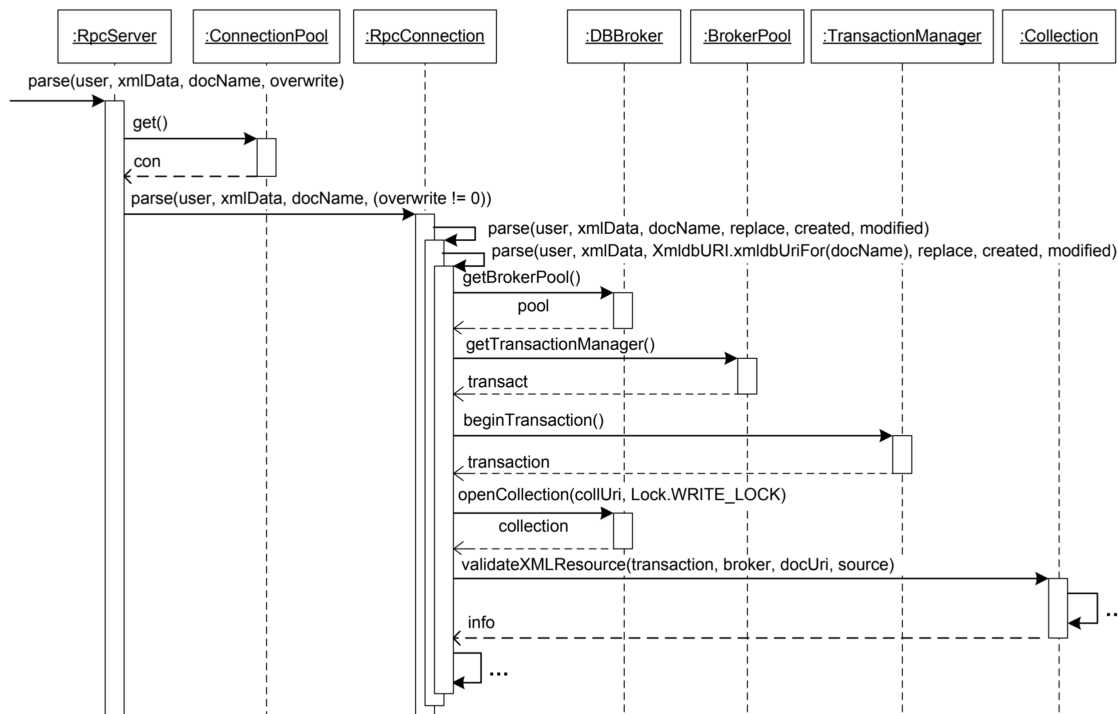


Abbildung 8.2: Einfügen über die XML-RPC-Schnittstelle

In der Klasse `RpcConnection` wird der Pfad, an dem das Dokument gespeichert werden soll, in eine `XmldbURI` umgewandelt. Als nächstes wird eine neue Transaktion für den Einfügevorgang gestartet. Danach wird eine Instanz der `Collection`, in die das Dokument eingefügt werden soll, mit der `getCollection`-Methode der `DBBroker`-Instanz angefordert. Gleichzeitig mit der Anforderung wird auf die `Collection` eine Schreibsperre (WRITE-LOCK) gesetzt. Mit dem Dokumentinhalt wird zunächst eine `InputStream`-Instanz erzeugt, die unmittelbar darauf zu einer `InputSource`-Instanz umgewandelt wird. In diesem Zustand wird der Dokumentinhalt, die Transaktion, die `DBBroker`-Instanz und der Dokumentname als `XmldbURI`, an die Methode `validateXMLResource` der `Collection`-Instanz übergeben. Der Ablauf der `validateXMLResource`-Methode wird im Zusammenhang mit Abbildung 8.4 für alle Zugriffspfade beschrieben. Nach der Validierung kann das Datenbanksystem mit dem Einfügen des Dokumentes weitermachen.

XML:DB API

Die Abbildung 8.3 stellt dar, wie die implizite Validierung beim Einfügen eines Dokumentes über die XML:DB API abläuft. Der Ablauf wird von dem `InteractiveClient` (put /pfad/dateiname.xml) aus gezeigt, der als Client mit eXist ausgeliefert

wird. Mit dem übergebenen Dateinamen wird eine `File`-Instanz erzeugt, die anschließend in ein `File`-Array eingefügt wird. Aus der `String`-Instanz mit dem Pfad des einzufügenden Dokumentes und dem MIME-Typ des Dokumentes wird mit der Methode `createResource` der `LocalCollection`-Instanz eine `Ressource`-Instanz erzeugt. Dieser `Resource`-Instanz wird mit `setContent` die zugehörige `File`-Instanz zugewiesen. Anschließend wird die `Resource`-Instanz an die Methode `storeResource` der `LocalCollection`-Instanz übergeben.

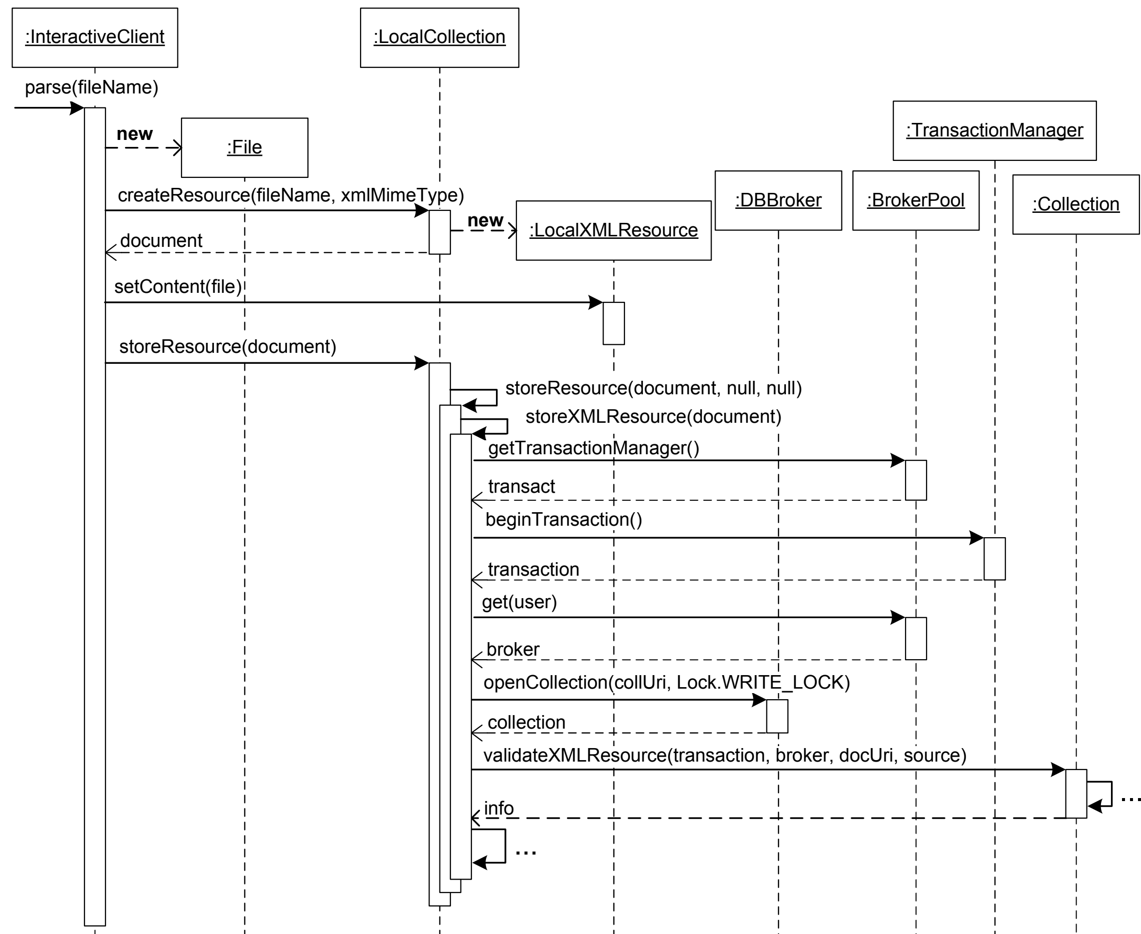


Abbildung 8.3: Einfügen über die XML:DB API

In der Methode `storeResource` wird die Methode `storeXMLResource` aufgerufen. Dort wird eine neue Transaktion für das Einfügen des Dokumentes gestartet. Als nächstes wird mit der `BrokerPool`-Instanz eine `DBBroker`-Instanz angefordert. Danach wird eine Instanz der `Collection`, in die das Dokument eingefügt werden soll, mit der `getCollection`-Methode der `DBBroker`-Instanz angefordert und eine Schreibsperre (WRITE-LOCK) darauf gesetzt. Anschließend wird aus der einzufügenden Datei eine `InputSource`-Instanz erzeugt, die zusammen mit der Transaktion, der

DBBroker-Instanz und dem Dateinamen an die validateXMLResource-Methode der Collection-Instanz übergeben wird. Der Ablauf der validateXMLResource-Methode wird im Zusammenhang mit Abbildung 8.4 für alle Zugriffspfade beschrieben. Nach der Validierung kann das Datenbanksystem mit dem Einfügen des Dokumentes weitermachen.

Validierung in der Klasse Collection

Zur Validierung der Dokumente wird pfadunabhängig, wie aus dem Abschnitt 6.2 bekannt, die Klasse Collection verwendet. Der genau Ablauf in der Klasse Collection wird in der Abbildung 8.4 graphisch veranschaulicht.

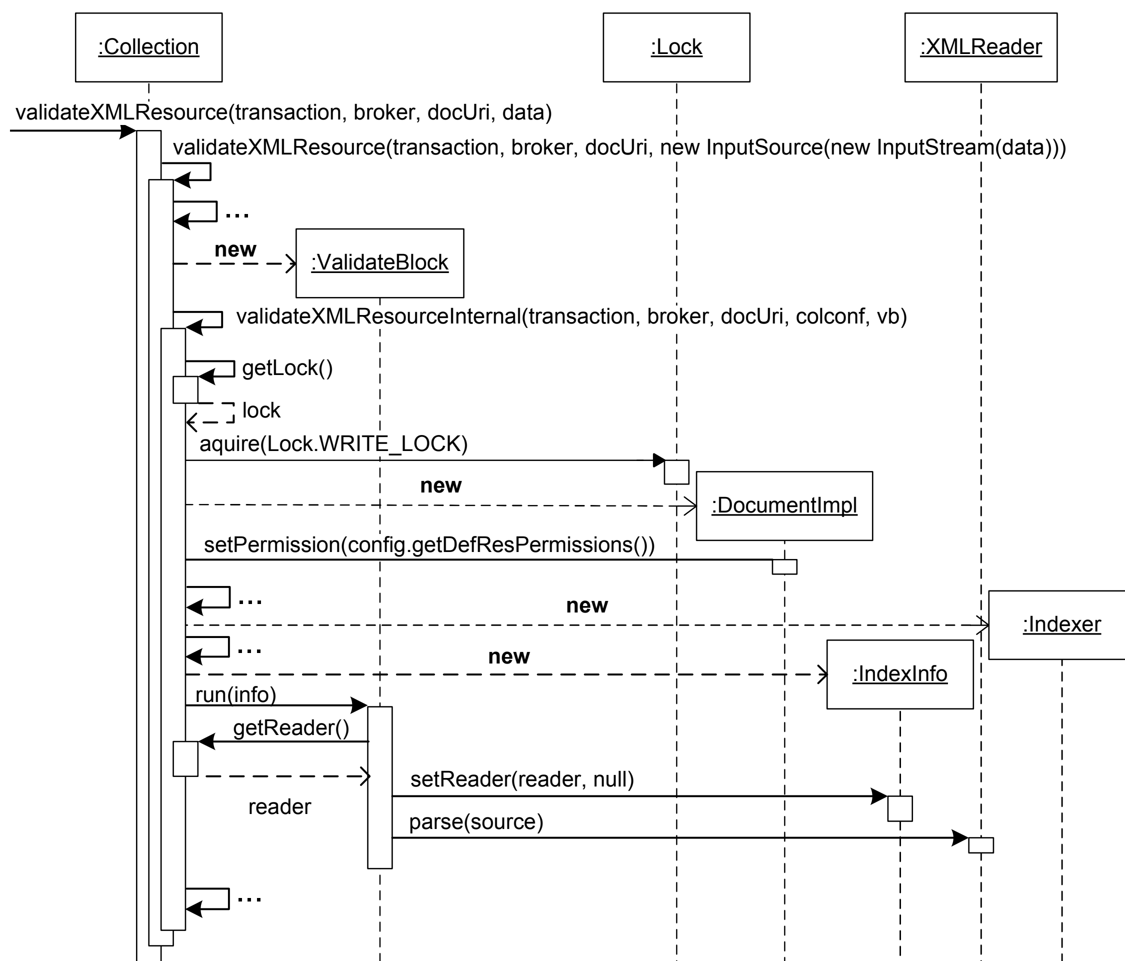


Abbildung 8.4: Implizite Validierung in der Klasse Collection

Zur Validierung wird die Methode `validateXMLResource` aufgerufen, die intern wiederum die Methode `validateXMLResourceInternal` aufruft. Innerhalb der `validateXMLResourceInternal`-Methode wird eine Schreibsperre (WRITE-LOCK) gesetzt, die verhindert, dass andere Transaktionen auf dasselbe Dokument schreibend zugreifen. Im weiteren Verlauf ruft diese Methode, die Methode `run` einer `ValidateBlock`-Instanz auf. Diese Methode löst die eigentliche Validierung aus, nachdem sie eine XMLReader-Instanz aus dem `XMLReaderPool` angefordert und entsprechend der Konfigurationsdatei (siehe Abschnitt 8.3) konfiguriert hat. Der `XMLReader`-Instanz wird das Dokument in der Methode `parse` in einer `InputSource`-Instanz übergeben. Die Validierung des XML-Dokumentes war genau dann erfolgreich, wenn während der Ausführung der `parse`-Methode keine Exceptions (Fehler) auftreten.

8.2 Explizite Validierung

Mit der expliziten Validierung lassen sich Dokumente, die im Datenbanksystem vorhanden sind, jederzeit explizit auf ihre Validität prüfen. Diese Prüfung wird in der Klasse `Validator` veranlasst. Bei XML-Dokumenten, die gegen eine Schemasprache (DTD/XML-Schema) geprüft werden sollen, muss entweder in der Datei die zugehörige Grammatik angegeben sein, oder ihr muss als zusätzlicher Parameter die Grammatik übergeben werden. Wenn keine Grammatik gefunden wird, kann das Dokument nicht auf seine Validität geprüft werden und ist somit nicht valide.

In Abschnitt 6.2 wurde festgestellt, dass die explizite Validierung auf jedem der drei Pfade durch die Klasse `Validator` veranlasst wird. Bei zweien dieser Pfade, nämlich der über die XML:RPC-Schnittstelle und der über die grafische Oberfläche im Embedded-Modus (XML:DB API), wird die Validierung eines Objektes vom Benutzer veranlasst. Über die anderen Datenbankschnittstellen ist diese manuelle Validierung nicht möglich, sondern es muss der XQuery-Pfad verwendet werden. Der dritte Pfad bietet eine Erweiterung der XQuery Syntax um zwei Funktionen zur Validierung von Dokumenten. Die Validierungsfunktionen unterscheiden sich nur in ihrem Rückgabewert. Die Funktion `validate` gibt das Ergebnis als boolschen Wert zurück, während die Funktion `validate-report` ein XML-Dokument zurückgibt, in dem steht, ob das Dokument valide ist und falls nicht, welche Fehler bei der Validierung aufgetreten sind.

XML-RPC-Schnittstelle

Die Abbildung 8.5 zeigt den Ablauf einer Validierung, die über die XML-RPC-Schnittstelle veranlasst wird. Dazu wird die Methode `isValid` in der Klasse `RpcServer` mit dem Namen des Benutzers und dem Namen des zu überprüfenden XML-

Dokumentes aufgerufen. Anschließend fordert die Funktion eine `RpcConnection`-Instanz aus dem `ConnectionPool` an. Dieser `RpcConnection` werden die Übergabeparameter zur weiteren Validierung übergeben.

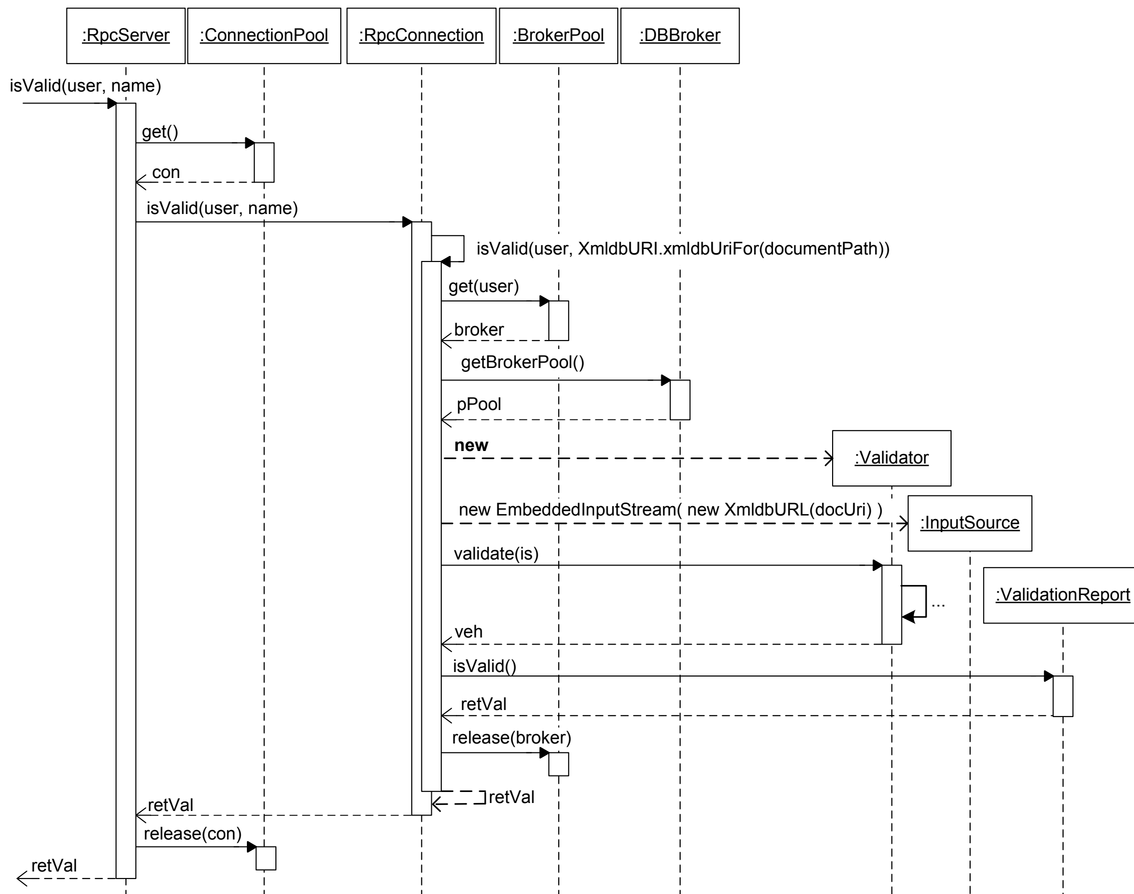


Abbildung 8.5: Sequenzdiagramm Validierung über die XML-RPC-Schnittstelle

Zur weiteren Verarbeitung wird der Pfad, der bisher in einer `String`-Instanz steht in eine `XmldbURI` umgewandelt. Anschließend wird mit einer `DBBroker`-Instanz eine weitere `BrokerPool`-Instanz angefordert. Mit dieser `BrokerPool`-Instanz wird eine Instanz der Klasse `Validator` erstellt. Die `XmldbUri`, die den Pfad des Dokumentes enthält, wird dazu verwendet, um eine `InputStream`-Instanz, die das Lesen des Dokumentes ermöglicht zu erzeugen. Diese `InputStream`-Instanz wird der Methode `validate` der `Validator`-Instanz übergeben. Der innere Ablauf der Klasse `Validator` wird im Zusammenhang mit Abbildung 8.8 erläutert. Das Ergebnis der Validierung wird bei der `ValidationReport`-Instanz, die als Rückgabewert von der `validate`-Methode kam, angefragt. Dieses Ergebnis wird anschließend an die aufrufenden Klassen übermittelt und schließlich dem Benutzer mitgeteilt.

Bei der Analyse der Validierung über die XML-RPC-Schnittstelle zeigte sich, dass dabei kein Parameter, der den Pfad zur Grammatik (Schemadatei) enthält, angegeben werden kann. Dadurch können Dokumente, deren Schemadateien (XML/DTD) explizit im Datenbanksystem liegen nicht ohne weiteres korrekt validiert werden.

XML:DB API

Im Embedded-Modus wird die XML:DB API verwendet um auf das Datenbanksystem zuzugreifen. Hier wird beispielhaft der Ablauf bei der Verwendung des `InteractiveClients` beschrieben (siehe Abbildung 8.6), der bei eXist mitgeliefert wird. Wenn die explizite Validierung eines Dokumentes angeordnet wird, fordert die Anwendung, mit dem `Collection`-Interface der XML:DB API eine `LocalValidationService`-Instanz an. Dieser Instanz werden die Übergabeparameter des Validierungsbefehles in der Methode `validateResource` übergeben.

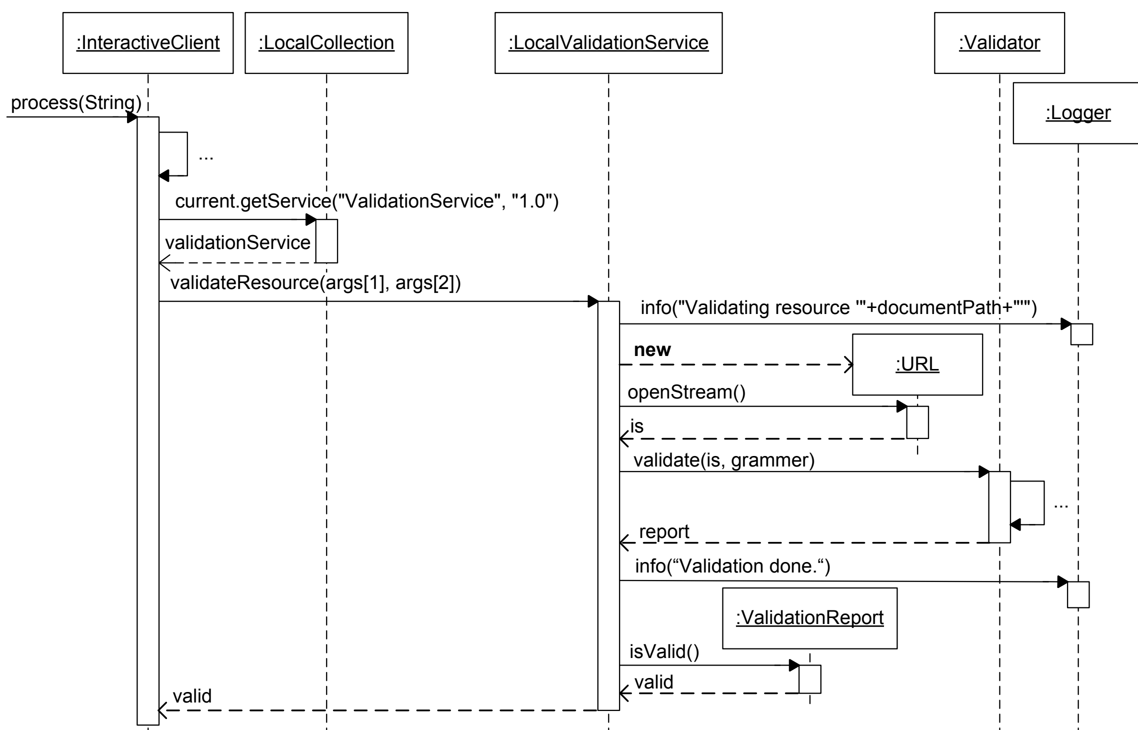


Abbildung 8.6: Sequenzdiagramm Validierung im Embedded-Modus

Die `String`-Instanz die den Pfad enthält, wird in eine `InputStream`-Instanz umgewandelt, aus der anschließend gelesen werden kann. Diese `InputStream`-Instanz wird zusammen mit dem Pfad der Grammatik (Schemadatei), falls vorhanden, an die Methode `validate`, der `Validator`-Instanz, die im Konstruktor der Klasse erstellt

wurde, übergeben. Der innere Ablauf der Klasse `Validator` wird im Zusammenhang mit Abbildung 8.8 erläutert. Das Ergebnis der Validierung wird anschließend aus der ValidationReport Instanz ausgelesen und als boolscher Wert an den Client des Benutzers zurückgegeben und dort angezeigt.

XQuery

Die Abbildung 8.7 zeigt den Ablauf der Validierung, die über einen XQuery Befehl ausgelöst wird. Dieser Ablauf bezieht sich auf einen Aufruf des Befehles in der Form `validation:validate("/db/example.xml", "/db/example.xsd")`.

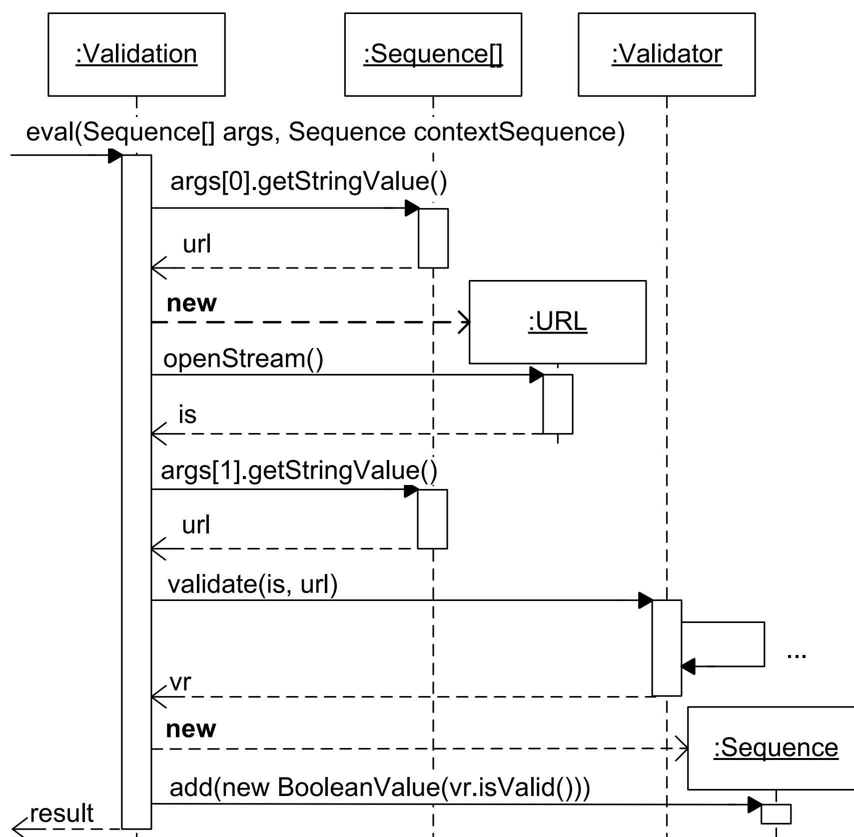


Abbildung 8.7: Sequenzdiagramm Validierung bei XQuery

Zunächst wird aus dem Pfad des zu validierenden Dokumentes eine `InputStream`-Instanz erzeugt. Im nächsten Schritt wird der absolute Pfad der Grammatik in eine `String`-Instanz umgewandelt. Die `InputStream`-Instanz wird zusammen mit der `String`-Instanz an die `validate`-Methode, der im Konstruktor erstellten `Validator`-Instanz übergeben. Der innere Ablauf der Klasse `Validator` wird im Zusammenhang

mit Abbildung 8.8 erläutert. Anschließend wird das Ergebnis erstellt. In diesem Fall enthält das Ergebnis nur den boolschen Wert, den die `isValid`-Methode der Klasse `ValidationReport` liefert.

Wenn die Validierung mit dem Befehl `validation:validate-report($PARAMETERS)` veranlasst wird, wird aus der `ValidationReport`-Instanz ein ausführlicher Ergebnisbericht erzeugt.

Validierung in der Klasse Validator

Die Abbildung 8.8 zeigt, was intern in der Klasse `Validator` passiert. Dies wird separat gezeigt, damit es in den anderen Erklärungen nicht redundant enthalten ist. Der beschriebene Fall ist der Aufruf einer Validierung, ohne dabei den Pfad zu einer Grammatik mitzugeben.

Der Klasse `Validator` wird beim Aufruf der Methode `validate`, das Dokument in Form einer `InputStream`-Instanz, welche validiert werden soll, übergeben. Zusätzlich kann eine Grammatik übergeben werden. Aus der `InputStream`-Instanz wird eine `InputStreamReader`-Instanz erzeugt, die zusammen mit einer leeren Grammatik, einer anderen `validate`-Methode übergeben wird. In dieser Methode wird eine Instanz der Klasse `SAXParserFactory` erzeugt. Mit dieser `SAXParserFactory`-Instanz wird eine `SAXParser`-Instanz und daraus schließlich eine `XmlReader`-Instanz erstellt. Aus der `InputStreamReader`-Instanz wird eine `InputSource`-Instanz generiert. Anschließend wird die `XmlReader`-Instanz mit den Methoden `setProperty` und `setFeature` für die Validierung konfiguriert. Der letzte `setProperty` Aufruf konfiguriert, wo die Grammatik zu suchen ist, in diesem Fall also im Systemkatalog. Sowohl der `grammerPool`, als auch der `systemCatalogResolver`, die in den `setProperty` Aufrufen verwendet werden, wurden im Konstruktor der Klasse `Validator`, mit der dort übergebenen `BrokerPool`-Instanz angefordert. Anschließend wird der `XmlReader`-Instanz als ErrorHandler eine neu erzeugte `ValidationReport`-Instanz übergeben. Analog wird der `XmlReader`-Instanz als ContentHandler eine `ValidationContentHandler`-Instanz übergeben. Daraufhin wird die Startzeit der Validierung in den `ValidationReport` eingetragen und das Dokument von Methode `parse` der `XmlReader`-Instanz eingelesen und gegebenenfalls gegen die angegebene Grammatik geprüft. Nachdem die `XmlReader`-Instanz mit der Validierung fertig ist, wird die Endzeit im `ValidationReport` festgehalten. Abschließend wird der Namensraum der `ValidationReport`-Instanz gesetzt. Das Resultat der Validierung ist in der `ValidationReport`-Instanz verfügbar. Dieses Objekt enthält die letzte aufgetretene `Exception` und eine Liste aller `SAXParseExceptions`, die während des Validierungsvorganges aufgetreten sind. Wenn keine `Exception` aufgetreten ist, wurde ein gültiges Dokument geprüft. Diese Instanz wird an die aufrufende Klasse zurückgegeben.

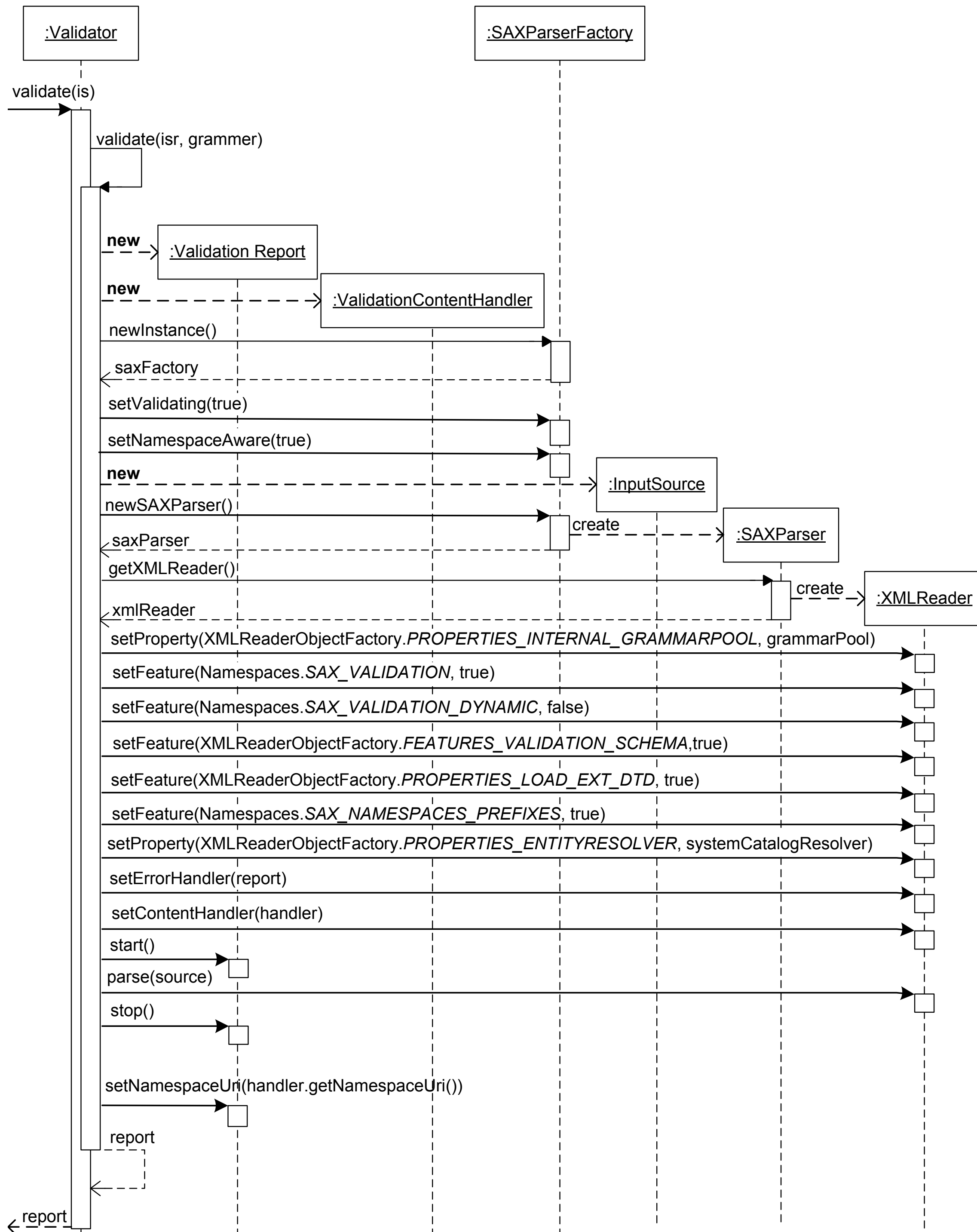


Abbildung 8.8: Sequenzdiagramm der Klasse Validator

Validierung (ohne Grammatik)

Im gerade erläuterten Fall wurde keine Grammatik zur Validierung übergeben. Dies wird noch einmal durch den Ausschnitt aus dem Gesamtdiagramm in Abbildung 8.9 gezeigt. Dabei wird der Systemkatalog als Quelle für Grammatiken angegeben und bei der eigentliche Validierung später durchsucht, um eine Grammatik zu finden.

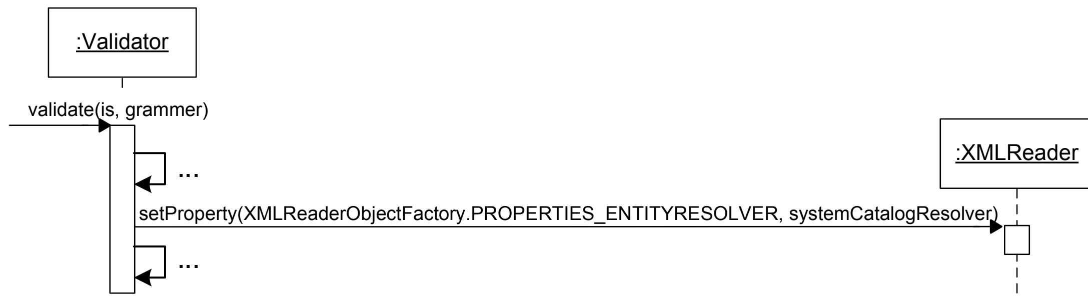


Abbildung 8.9: Sequenzdiagramme Validator, Grammatik Fall 1

Validierung (mit Katalogdatei)

Im Fall, der in Abbildung 8.10 gezeigt wird, wird als Pfad zur Grammatik, der Pfad zu einer Katalogdatei (catalog.xml) angegeben. Um diese Grammatik in der `XmlReader`-Instanz zu konfigurieren, wird eine Instanz der Klasse `eXistXMLCatalogResolver` erzeugt. Anschließend wird dieser Instanz, in der Methode `setCatalogList`, der absolute Pfad der Katalogdatei in einer `Stringarray`-Instanz übergeben. Der Methode `setProperty` der `XmlReader`-Instanz wird die resultierende Instanz im letzten Aufruf übergeben.

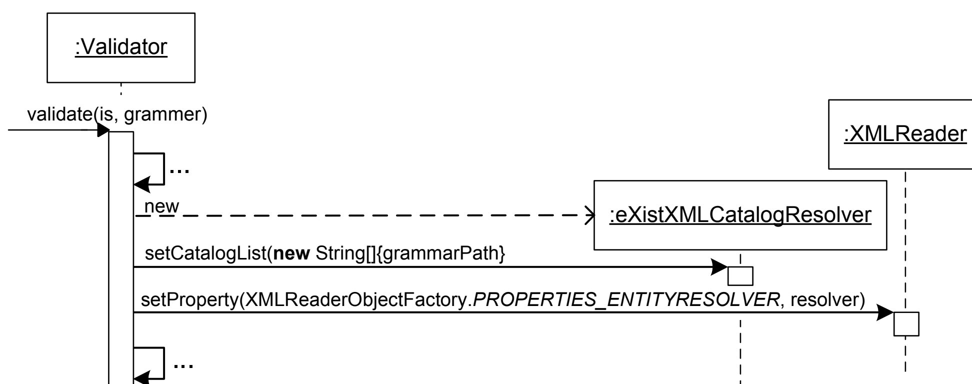


Abbildung 8.10: Sequenzdiagramme Validator, Grammatik Fall 2

Validierung (mit Collection-Pfad)

Die Abbildung 8.11 zeigt den Fall, dass als Grammatik keine Datei angegeben ist, sondern der Pfad einer Collection, die nach der Grammatik durchsucht werden soll. Für diesen Zweck wird eine Instanz der Klasse `SearchResourceResolver` erzeugt. Im Konstruktor wird der absolute Pfad der zu durchsuchenden Collection und eine Instanz der Klasse `BrokerPool` übergeben. Die erzeugte Instanz der Klasse `SearchResource-Resolver` wird anschließend beim letzten `setProperty` Aufruf der `XmlReader`-Instanz übergeben.

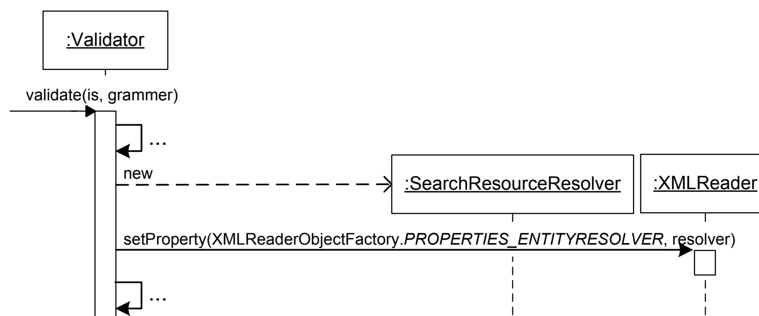


Abbildung 8.11: Sequenzdiagramme Validator, Grammatik Fall 3

Validierung (mit Schemadatei)

Im letzten Fall, der in Abbildung 8.12 dargestellt ist, wird der Pfad zu einem Dokument, das ein XML-Schema oder eine Dokumenttypdefinition (DTD) enthält angegeben. Um diese in der `XmlReader`-Instanz konfigurieren zu können, wird aus dem absoluten Pfad des Dokumentes eine Instanz der Klasse `AnyUriResolver` erzeugt, die dem letzten `setProperty` Aufruf der `XmlReader`-Instanz übergeben wird.

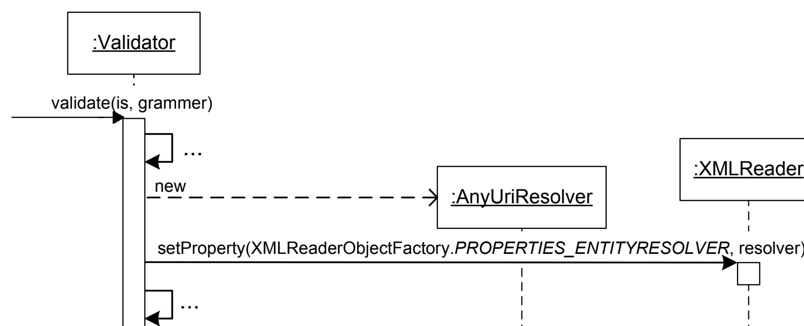


Abbildung 8.12: Sequenzdiagramme Validator, Grammatik Fall 4

8.3 Konfiguration

Im Folgenden wird gezeigt, wie die Standardkonfiguration der Validierung im Datenbanksystem vorgenommen werden kann und welche Auswirkungen die verschiedenen Konfigurationsmöglichkeiten haben. Im Anschluss daran wird erläutert, wie die Standardkonfiguration der Validierung in den einzelnen Collections überschrieben werden kann.

8.3.1 Konfiguration des Datenbanksystems

Die Konfiguration von eXist wird mit dem XML-Dokument `conf.xml`, welches unter `$EXIST_HOME/conf.xml` zu finden ist, vorgenommen. Dieses Dokument muss wohlgeformtes XML enthalten. Die Konfiguration für die Validierung ist im Element `validation` enthalten.

```
1 <validation mode="yes">
2    ...
3 </validation>
```

Listing 8.1: Ausschnitt Konfigurationsdatei

Das Attribut `mode` kann drei verschiedene Werte annehmen:

- `auto`:
  In dieser Einstellung werden alle Dokumente, die wohlgeformt sind, aber keine Angaben über das Schema des Dokumentes enthalten in das Datenbanksystem aufgenommen. Wenn das Dokument Angaben zum Schema des Dokumentes enthält, wird ähnlich wie im `yes`-Fall vorgegangen. Alle Dokumente werden auf ihre Validität geprüft und nur zum Datenbanksystem hinzugefügt, wenn sie valide sind. Wenn die Angabe zum Schema ungültig ist oder nicht interpretiert werden kann, wird das Dokument ebenfalls abgelehnt.

- `yes`:
  Bei der Einstellung `yes` ist die Validierung eingeschaltet und alle Dokumente müssen valide sein, wenn sie in das Datenbanksystem eingefügt werden. Wenn die Validierung als Ergebnis ein `ungültig` liefert, wird das Dokument abgelehnt. Wenn die Datei keine Angaben über das ihr zugrunde liegende Schema (z.B. DTD, XML-Schema) enthält, ist die Validierung der Datei nicht möglich und sie wird ebenfalls abgelehnt. Wenn die Datei eine Angabe über das zugrunde liegende Schema (z.B. DTD, XML-Schema) enthält, dieses an der angegebenen Stelle jedoch nicht vorhanden ist, wird das Dokument abgelehnt.

- `no`:
  Wenn die Einstellung auf `no` steht, ist die Validierung des Datenbanksystems abgeschaltet. Alle wohlgeformten Dokumente werden in diesem Fall in das Datenbanksystem aufgenommen.

Dabei ist darauf zu achten, dass sich die Einstellung für den Validationsmodus des Datenbanksystems nur auf die implizite Validierung von Dokumenten bezieht und nicht auf die explizite Validierung. Die explizite Validierung von Dokumenten ist unabhängig von dieser Konfiguration möglich.

8.3.2 Konfiguration von Collections

Neben der gerade beschriebenen globalen Konfiguration des Datenbanksystems, kann intern jede einzelne Collection konfiguriert werden. Diese Konfigurationsdatei heißt `collection.xconf` und liegt in dem Datenbanksystem unter `/db/system/config/$COLLECTION_PATH`, im entsprechenden Unterverzeichnis. Mit dieser Konfigurationsdatei kann für jede Collection ein eigener Validationsmodus festgelegt werden. Eine beispielhafte Konfigurationsdatei, bei der die Validation von Dokumenten eingeschaltet ist, sieht folgendermaßen aus:

```
1 <collection xmlns="http://exist-db.org/collection-config/1.0">
2     <validation mode="yes"/>
3 </collection>
```

Listing 8.2: Configuration Konfigurationsdatei

Das Attribut `mode` kann dieselben Werte wie bei der Konfiguration des Datenbanksystems annehmen. Die Werte haben dieselben Auswirkungen wie im Fall der globalen Konfiguration, mit dem Unterschied, dass die Konfiguration nicht für das gesamte Datenbanksystem, sondern nur für die entsprechende Collection und ihre Kinder gilt. Besitzt eine Kind-Collection eine eigene Konfiguration, dann überdeckt diese die Konfiguration der Eltern-Collection.

8.4 Lesen eines Dokumentes aus dem Datenbanksystem

Das Lesen eines Dokumentes aus dem Datenbanksystem wird in diesem Abschnitt, wie in Abbildung 8.13 dargestellt, aus der Sicht der XML-RPC-Schnittstelle beschrieben. Die Methode `getDocument` der Klasse `RpcServer` erhält eine `User`-Instanz und den Pfad des Dokumentes in einer `String`-Instanz. Zusätzlich werden zwei Konfigurationsparameter übergeben. Aus dem `ConnectionPool` wird eine `RpcConnection`-Instanz

angefordert. Die Konfigurationsparameter werden in eine HashMap eingefügt. Diese HashMap und die anderen beiden Parameter werden an die Methode `getDocument` der `RpcConnection`-Instanz übergeben.

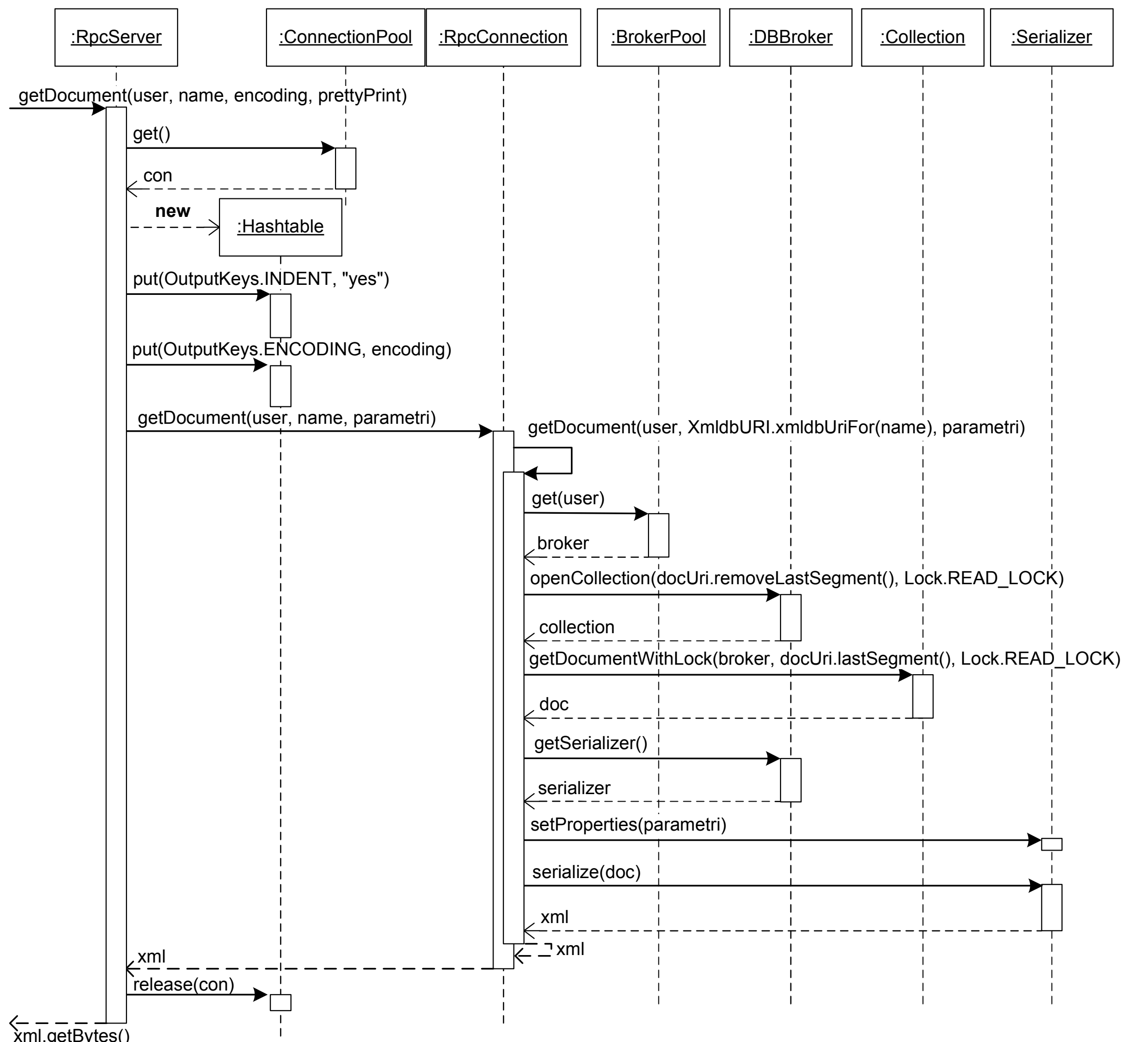


Abbildung 8.13: Datei aus dem Datenbanksystem lesen

In der Klasse `RpcConnection` wandelt die Methode die `String`-Instanz mit dem Pfad zu dem Dokument in eine `XmldbURI` um und fordert eine Instanz der Klasse `DBBroker` von der Klasse `BrokerPool` an. Mit dieser `DBBroker`-Instanz fordert die Methode eine Instanz der `Collection` an, in der das gewünschte Dokumente liegt und setzt auf diese ein Lesesperre (ReadLock). Mit der Methode `getDocumentWithLock` der Klasse `Collection` wird das Dokumente als Instanz der Klasse `DocumentImpl` angefordert. Während des Vorganges ist ein Lesesperre (ReadLock) auf dem Dokument. Um aus der `DocumentImpl`-Instanz die XML-Daten zu extrahieren wird mit der `DBBroker`-

Instanz eine `Serializer`-Instanz angefordert. Die `Serializer`-Instanz wird mit den übergebenen Parametern konfiguriert. Anschließend wird die `DocumentImpl`-Instanz an die Methode `serialize` der Klasse `Serializer` übergeben. Der Ausschnitt aus dem Klassendiagramm der Klasse `Serializer` zeigt, dass eine Methode `serialize` auch für die Objekttypen `NodeValue` und `NodeProxy` vorhanden ist. Als Ergebnis liefert diese Methode die enthaltenen XML-Daten in Form einer `String`-Instanz. Diese `String`-Instanz wird an die aufrufende Methode zurückgegeben.

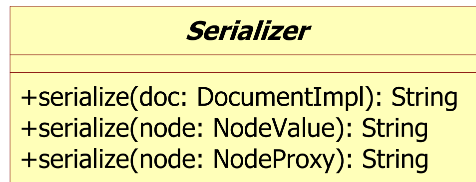


Abbildung 8.14: Klassendiagramm von Serializer

Zurück in der Klasse `RpcServer` wird die `RpcConnection`-Instanz wieder freigegeben. Danach wird das Dokument in ein Bytearray mit gewünschtem Zeichensatz umgewandelt und an den Userclient übermittelt.

8.5 Trigger

In Abschnitt 6.3 wurden die Grundzüge der Funktionsweise von Triggern im eXist Datenbanksystem erläutert. Zusätzlich wurde erklärt, wie ein Trigger für bestimmte Aktionen konfiguriert werden kann. Im Folgenden wird die Funktionsweise eines Java Triggers erläutert. Da, wie bereits in 6.3 erklärt, zwischen den Ereignissen bei Collections und Dokumenten unterschieden wird, wird im Weiteren die Funktionsweise eines DocumentTriggers beschrieben.

Wenn ein Trigger für eine Collection verwendet werden soll, muss er wie in Abschnitt 6.3 in die Konfigurationsdatei der Collection eingetragen werden. In der Konfigurationsdatei wird eingestellt, bei welchen der dokumentbezogenen Ereignissen (insert, update, delete) der Trigger ausgeführt werden soll. Das Ereignis „insert“ tritt auf, wenn ein Dokument in das Datenbanksystem eingefügt wird und dabei kein bestehendes Dokument überschreibt. Ein „update“-Ereignis tritt in drei Fällen auf. Im ersten Fall wird ein Dokument in das Datenbanksystem eingefügt und überschreibt dabei ein anderes Dokument, im nächsten Fall wird ein Dokument in dem Datenbanksystem durch einen XQuery-Update-Extension Ausdruck verändert und im letzten Fall wird ein Dokument durch einen XUpdate Ausdruck verändert. Das „delete“-Ereignis tritt genau dann auf, wenn ein Dokument aus dem Datenbanksystem entfernt wird. Dabei ist eine beliebige Kombination dieser Ereignisse zulässig.

Die Abbildung 8.15 zeigt den groben Ablauf der einzelnen Phasen des `ExampleTriggers`. Aus dem Sequenzdiagramm geht hervor, dass die Konfigurations- und Vorbereitungsphase vor der Änderung ausgeführt werden. Die Endphase wird ausgeführt, nachdem die Änderungen durchgeführt wurden, aber bevor sie persistiert wurden. Der Trigger wird letztendlich immer von der Klasse aufgerufen, die eine Änderung vornehmen will. Beim Einfügen eines Dokumentes wird die Ausführung des Triggers beispielsweise in der Klasse `Collection` veranlasst.

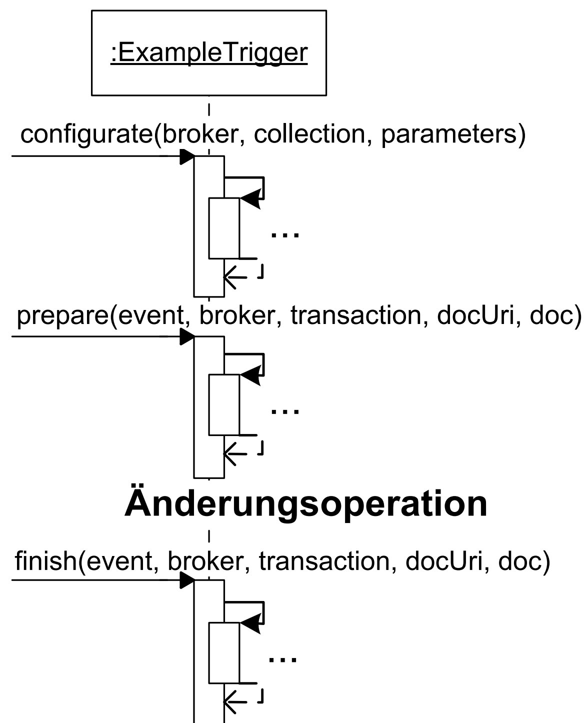


Abbildung 8.15: Sequenzdiegramm eines DokumentTriggers

8.5.1 Konfigurationsphase

Wenn ein Ereignis, bei dem der Trigger angewendet werden soll auftritt, dann wird zuerst die Methode `configure` aufgerufen. Als Parameter erhält diese Methode eine Instanz der Klasse `DBBroker`, eine Instanz der `Collection`, in der sich das Dokument befindet und eine Instanz der Klasse `Map`, welche die optionalen Parameter, die in der Konfigrationsdatei der Collection für den Trigger übergeben werden können, enthält. Die hier übergebenen Parameter können für die spätere Benutzung als Membervariable abgelegt werden, oder es können mit ihnen bereits andere Aktionen ausgeführt werden, wie beispielsweise ein Log-Eintrag.

8.5.2 Vorbereitungsphase

Nach der Konfigurationsphase folgt die Vorbereitungsphase. Dazu wird die Methode `prepare` des Triggers aufgerufen. Dieser Aufruf erfolgt, bevor eine Änderung an dem Dokument vorgenommen wurde. Die Methode bekommt als Übergabeparameter das Ereignis als `Integer` codiert, eine Instanz der Klasse `DBBroker`, eine Instanz der Klasse `Tnx`, welche die aktuelle Transaktion verwaltet, eine `XmldbURI`, die den Pfad des Dokumentes in dem Datenbanksystem enthält und eine Instanz der Klasse `DocumentImpl`, welche die aktuelle Version des zu verändernden Dokumentes enthält. In der `prepare`-Methode können beliebige Aktionen auf dem Datenbanksystem ausgeführt werden. Der `HistoryTrigger` legt beispielsweise vor jeder Änderung an einem Dokument eine Kopie des Dokumentes an.

8.5.3 Endphase

Nachdem die Vorbereitungsphase des Triggers abgeschlossen ist, führt das Datenbanksystem die gewünschten Änderungen an dem Dokument aus. Wenn die Änderungen durchgeführt sind, aber bevor die Transaktion abgeschlossen wird, wird die Endphase des Triggers ausgeführt, indem die Methode `finish` aufgerufen wird. Die Methode `finish` bekommt mit Ausnahme der `DocumentImpl`-Instanz dieselben Übergabeparameter. Die Instanz der Klasse `DocumentImpl` unterscheidet sich dadurch, dass sie das Dokument mit den Änderungen enthält. In dieser Methode können, genau wie in der Methode `prepare`, die gewünschten Aktionen ausgeführt werden. Das veränderte Dokument kann in dieser Methode beispielsweise auf seine Validität geprüft werden.

8.6 Zusammenfassung

In diesem Abschnitt wurden zunächst die Zugriffspfade zu der Klasse `Collection` für die REST- und XML-RPC-Schnittstelle sowie für die XML:DB API analysiert. Daraufhin wurden die Zugriffspfade zu der Klasse `Validator` beschrieben. Die Analyse der Zugriffspfade auf die Validierungseinheit hat gezeigt, dass Transaktionen in den Datenbankschnittstellen beginnen und enden. Außerdem wird bereits in den Schnittstellen veranlasst, dass Dokumente und Collections, auf die zugegriffen wird mit den entsprechenden Sperren (Locks) versehen werden. Im Anschluss wurde erklärt, welche Konfigurationsmöglichkeiten das Datenbanksystem in Bezug auf die Validierung bietet. Die Untersuchung, wie Dokumente, die sich im Datenbanksystem befinden ausgelesen werden können, hat gezeigt, dass die Verwendung der Klasse `Serializer` sehr nützlich ist. Diese Klasse bietet eine Methode `serialize`, der ein `DocumentenImpl`-Instanz, zur Umwandlung des Dokumentinhaltes in eine `String`-Instanz, übergeben wird.

Des Weiteren wurden die einzelnen Phasen eines `DocumentTriggers` erklärt. Die Systemanalyse als Gesamtes hat gezeigt, dass der Zugriff auf die physische Datenbank über die abstrakte Klasse `DBBroker` läuft. Unterhalb von dieser Klasse ist die gesamte Abbildung von Dokumenten bis auf den physischen Speicher implementiert.

9 Systementwurf

Im Folgenden wird der verfeinerte Entwurf für ein Modul, welches die Verarbeitung eines OWL-Dokumentes und die Kommunikation mit dem OWL-Reasoner übernimmt, vorgestellt. Dieser Entwurf wird gefolgt von den Änderungen, die am Datenbanksystem vorgenommen werden müssen um den Entwurf zu integrieren. Daraufhin wird ein Entwurf für die Erweiterung der Konfigurierbarkeit der Validierung im Datenbanksystem erklärt. Danach wird der Entwurf für den `ValidationTrigger` vorgestellt.

9.1 OWL-Modul

Im Abschnitt 7.4 wurden der Basisentwurf und der erweiterte Entwurf für das Modul `owlconsistency` vorgestellt. Im Folgenden wird genauer auf den Basisentwurf eingegangen, da bereits im Abschnitt 7.4 festgelegt wurde, dass dieser und nicht der erweiterte Entwurf umgesetzt werden soll.

Der Basisentwurf des `owlconsistency` Moduls ermöglicht es, OWL-Dokumente an RacerPro zu übergeben und sie auf ihre Konsistenz prüfen zu lassen. Anschließend werden die Ergebnisse dieser Konsistenzprüfung ausgewertet und an die aufrufende Instanz weitergeleitet. Die OWL-Dokumente, die für diesen Entwurf betrachtet werden, enthalten eine T-Box und falls vorhanden eine A-Box.

Die Klasse `OWLChecker` stellt die Außenschnittstelle des Moduls `owlconsistency` dar. Die Abbildung 9.1 zeigt, wie durch den Aufruf der Methode `validate` dieser Klasse, die Konsistenzprüfung eines Dokumentes veranlasst wird. Beim Aufruf der Methode `validate` wird ihr das zu überprüfende Dokument übergeben.

Da das Dokument per Dateiübergabe an den OWL-Reasoner übergeben wird, muss das Dokument, bevor es auf seine Konsistenz geprüft wird, in eine Datei auf der Festplatte gespeichert werden. Hierfür wird eine Instanz der Klasse `CustomizeDocument` erzeugt. Anschließend wird das Dokument der Methode `toDisk` der Klasse `CustomizeDocument` übergeben, welche die erforderliche Datei auf der Festplatte anlegt. Als Rückgabewert übergibt die Methode `toDisk` den Pfad der temporären Datei an die aufrufende Klasse `OWLChecker`.

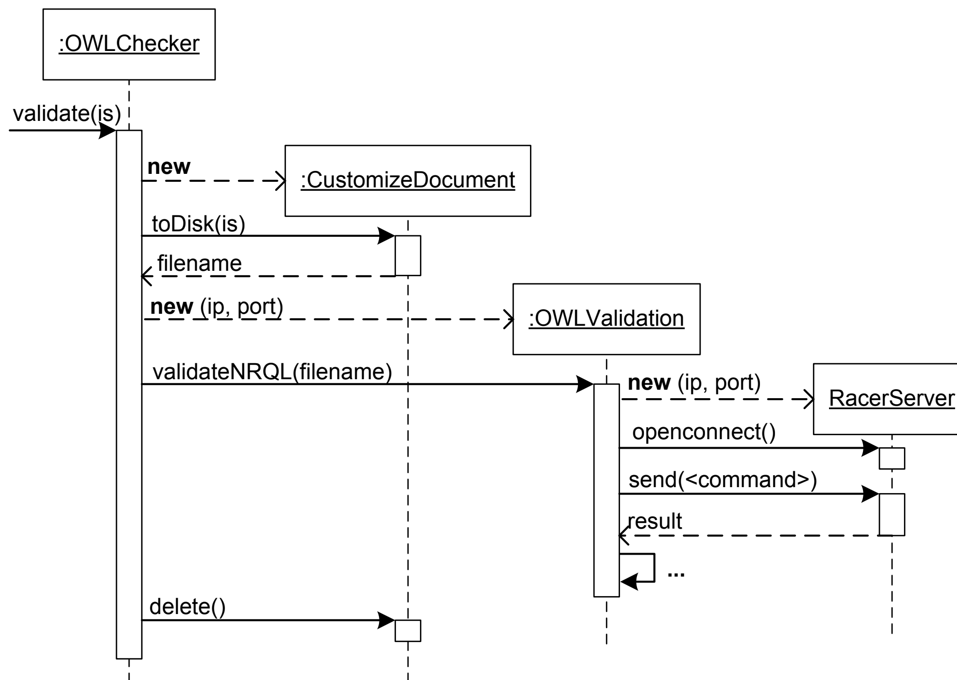


Abbildung 9.1: Einfache Darstellung der Konsistenzprüfung

Nachdem die Klasse `OWLChecker` den Pfad der temporären Datei bekommen hat, erzeugt sie eine Instanz der Klasse `OWLValidation`. Beim Erzeugen der Instanz wird die IP und der Port des OWL-Reasoners im Konstruktor übergeben. Anschließend wird die Methode `validateNRQL` der Klasse `OWLValidation` aufgerufen. Der Methode `validateNRQL` wird der Pfad, den die Methode `toDisk` als Ergebnis geliefert hat, übergeben. Die Methode `validateNRQL` baut eine Netzwerkverbindung mit dem OWL-Reasoner auf. Dazu wird das Modul `jracer` verwendet. Dieses Modul enthält die Klasse `RacerServer`, von der eine Instanz erzeugt wird. Mit dieser Instanz wird mit der Methode `connect` eine Verbindung zu RacerPro aufgebaut. Für den Verbindungsaufbau werden die IP und der Port, die der Klasse `RacerServer` im Konstruktor übergeben wurden, verwendet. Mit der Methode `send` der Klasse `RacerServer` wird das Kommando zum Einlesen des Dokumentes von der Festplatte an die Managementschnittstelle von RacerPro übergeben. Nachdem die Datei eingelesen ist, werden die Kommandos zum Überprüfen der Konsistenz der T-Box und der A-Box übertragen. Die Ergebnisse der Konsistenzprüfung, welche die Methode `send` als Rückgabewerte liefert, werden in der Methode `OWLValidation` ausgewertet. Das Ergebnis der Auswertung wird nur dann an die Klasse `OWLChecker` übermittelt, wenn das Dokument nicht konsistent ist, oder ein Fehler bei der Konsistenzprüfung aufgetreten ist.

Bevor die Methode `validate` der Klasse `OWLChecker` abgeschlossen wird, muss die temporäre Datei, in der das Dokument gespeichert wurde wieder gelöscht werden. Dazu wird die Methode `delete` der Klasse `CustomizeDocument` aufgerufen.

9.2 Änderungen an dem Datenbanksystem

Damit das entworfene `owlconsistency`-Modul auch vom Datenbanksystem für die Konsistenzprüfung von OWL-Dokumenten verwendet werden kann, muss es sowohl im Fall einer impliziten, als auch einer expliziten Validierung verwendet werden. In Abschnitt 8 wurde gezeigt, dass die implizite Validierung in der Klasse `Collection` und die explizite Validierung in der Klasse `Validator` ausgelöst wird.

Die Abbildung 9.2 zeigt, wie in der Klasse `Validator` das `owlconsistency` Modul aufgerufen wird. Dabei wird die Schemavalidierung von XML-Dokumenten durch die Konsistenzprüfung von OWL-Dokumenten ersetzt.

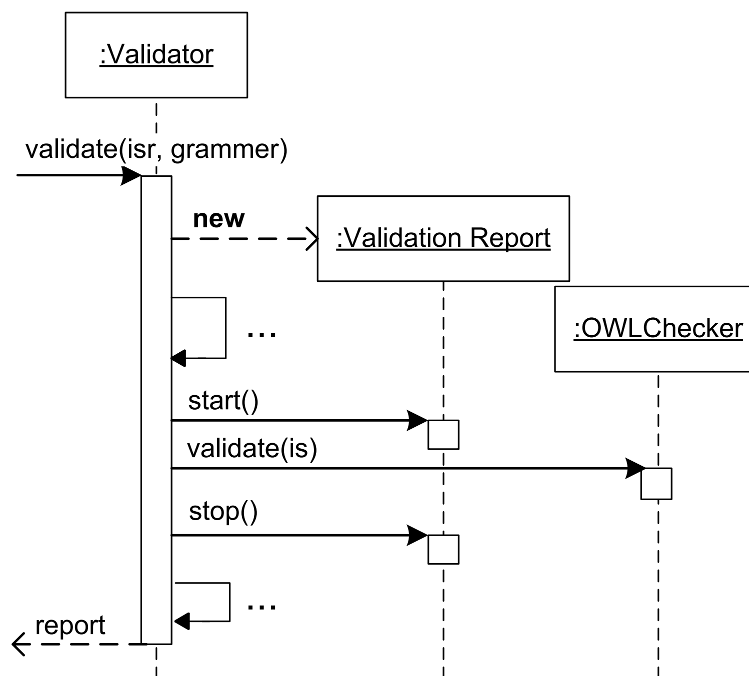


Abbildung 9.2: Validator mit OWL-Modul

Analog zeigt Abbildung 9.3, wie die Klasse `Collection` das `owlconsistency`-Modul aufruft. Diese Konsistenzprüfung wird durchgeführt, bevor das Dokument mit dem XML-Parser in das datenbankinterne Format umgewandelt wird, damit möglichst wenig Aufwand auf inkonsistente Dokumente verwendet wird.

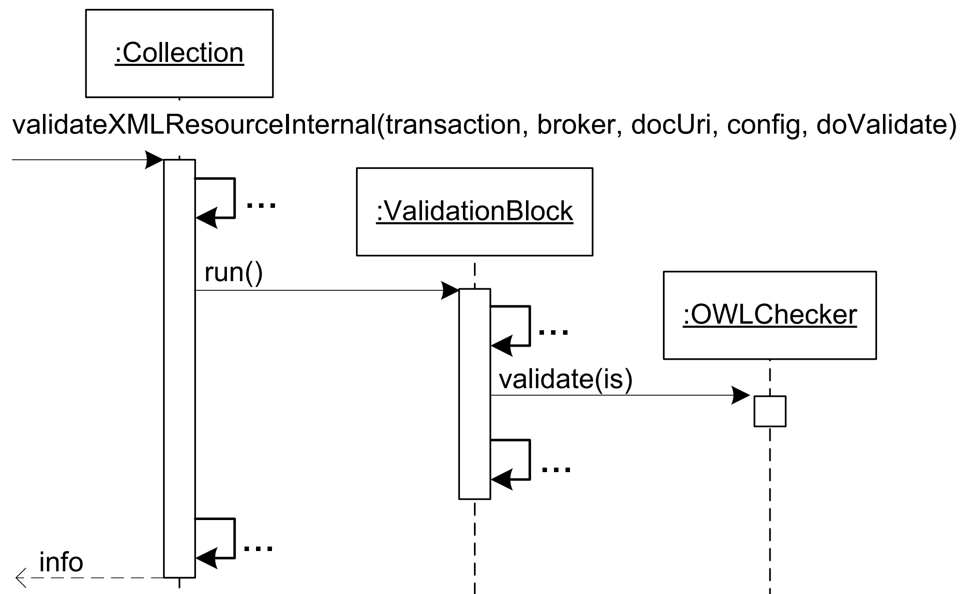


Abbildung 9.3: Collection mit OWL-Modul

Daraus folgt, dass die Methode `validate` der Klasse `Validator` und die Methode `run` der Klasse `ValidationBlock`, die ein Teil der Klasse `Collection` ist, entsprechend des Sequenzdiagramms in Abbildung 9.2 bzw. 9.3 geändert werden müssen.

9.3 Erweiterung der Konfiguration für OWL-Validierung

Die Idee ist, die im letzten Kapitel vorgestellten, bereits vorhandenen Konfigurationsmöglichkeiten, mit der derzeitigen Funktionalität beizubehalten und einen neuen Validationsmodus einzuführen. Auf diese Weise kann das Datenbanksystem weiterhin als natives XML-Datenbanksystem mit Schemavalidierung betrieben werden, alternativ kann das Datenbanksystem auch in ihrer neuen Rolle als Datenbanksystem, welches OWL-Dokumente validieren und speichern kann, verwendet werden. Zu diesem Zweck wird ein vierter Modus mit dem Namen `owl` eingeführt. Dieser Modus funktioniert so, dass jedes XML-Dokument, welches in das Datenbanksystem eingefügt werden soll, zuerst mit Hilfe des OWL-Reasoners auf seine Konsistenz untersucht wird. Das Dokument wird nur aufgenommen, wenn es ein konsistentes OWL-Dokument ist. Zusätzlich zum Einfügen des neuen Validationsmodus, werden im Validierungskontext der Konfiguration zwei neue Attribute eingefügt.

```
1 <validation mode="no" port="1234" ip="127.0.0.1">
2     ...
3 </validation>
```

Listing 9.1: Ausschnitt aus der Konfigurationsdatei

Eines der Attribute ist eine IP-Adresse, das andere ein Port. Diese beiden neuen Attribute werden eingeführt, da das Datenbanksystem über eine Netzwerkschnittstelle mit dem OWL-Reasoner kommuniziert. Da die beiden neuen Attribute in der globalen Konfigurationsdatei des Datenbanksystems enthalten sind, ist es nicht notwendig, sie statisch in den Quelltext des Datenbanksystems oder in einer eigenen Konfigurationsdatei abzulegen. Diese Tatsache trägt zur Vereinfachung der Gesamtkonfiguration des Datenbanksystems bei.

9.4 ValidationTrigger für XQuery und XUpdate

Das eXist Datenbanksystem bietet, in der verwendeten Version, keine Möglichkeit Dokumente, die durch die Ausführung eines XQuery-Update-Extension Ausdrucks oder XUpdate Ausdrucks verändert wurden, automatisch auf ihre Validität/Konsistenz prüfen zu lassen. Da von einem tiefen Eingriff in das Datenbanksystem abgesehen werden soll, wird versucht diese Funktionalität durch die Entwicklung eines entsprechenden Triggers zu ergänzen.

9.4.1 ValidationTrigger auf Basis des HistoryTriggers

Bei der Untersuchung der Triggermechanismen wurde festgestellt, dass ein `HistoryTrigger` existiert. Dieser `HistoryTrigger` eignet sich bei genauerer Untersuchung als Vorlage für den geplanten `ValidationTrigger`. Der HistoryTrigger legt, in der unveränderten Version, vor jeder Änderung an einem Dokument in dem Datenbanksystem eine Kopie dieses Dokumentes an, so dass zu jedem späteren Zeitpunkt auf eine beliebige alte Version des Dokumentes zugegriffen werden kann. Diese vorhandene Funktionalität bietet einen guten Ausgangspunkt, um diesen Trigger vom `HistoryTrigger` zu einem `ValidationTrigger` umzubauen.

Zu diesem Zweck wird der vorhandene Trigger so erweitert, dass, die in Abbildung 9.4 dargestellten Operationen in der Vorbereitungs- bzw. Endphase durchgeführt werden.

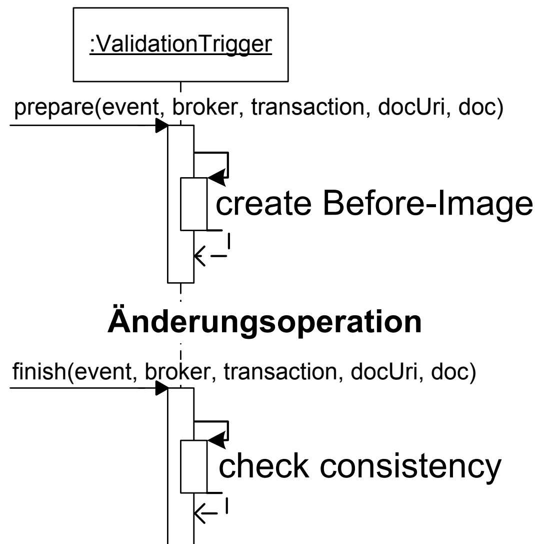


Abbildung 9.4: Phasenübersicht ValidationTrigger

In der Abbildung 9.5 ist zu sehen, dass in der Vorbereitungsphase (`prepare`-Methode) ein Before-Image[1] in Form einer Kopie des Dokumentes, welches anschließend verändert werden soll, im Datenbanksystem erstellt wird. Nach der Erstellung des Before-Images des Dokumentes ist diese Phase des Triggers beendet und das Datenbanksystem fährt mit den Änderungen an den Dokumenten fort.

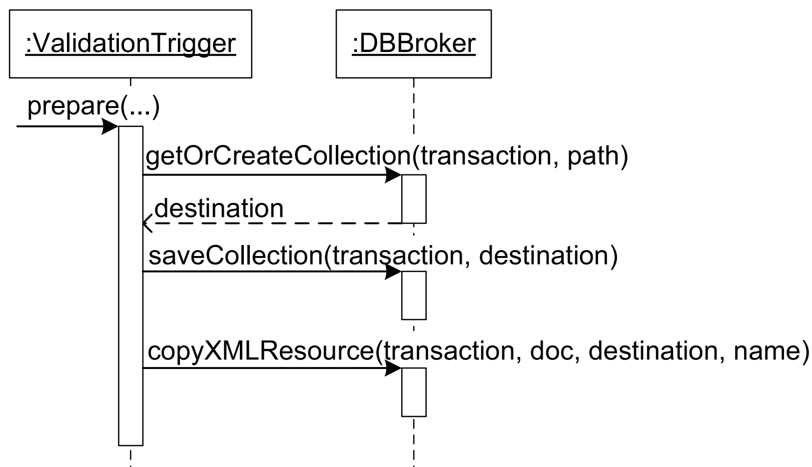


Abbildung 9.5: Sequenzdiagramm: Vorbereitungsphase des ValidationTriggers

[1] Unter einem Before-Image wird die Kopie eines Objektes, das anschließend geändert werden soll verstanden. Das Before-Image wird angelegt, damit im Fehlerfall die Ursprungsdaten wieder hergestellt werden können.

Nachdem die Änderungen am Dokument vorgenommen wurden, beginnt die, in Abbildung 9.6 dargestellte Endphase (`finish`-Methode) des Triggers. In der Endphase wird das veränderte Dokument untersucht, ob es weiterhin valide/konsistent ist.

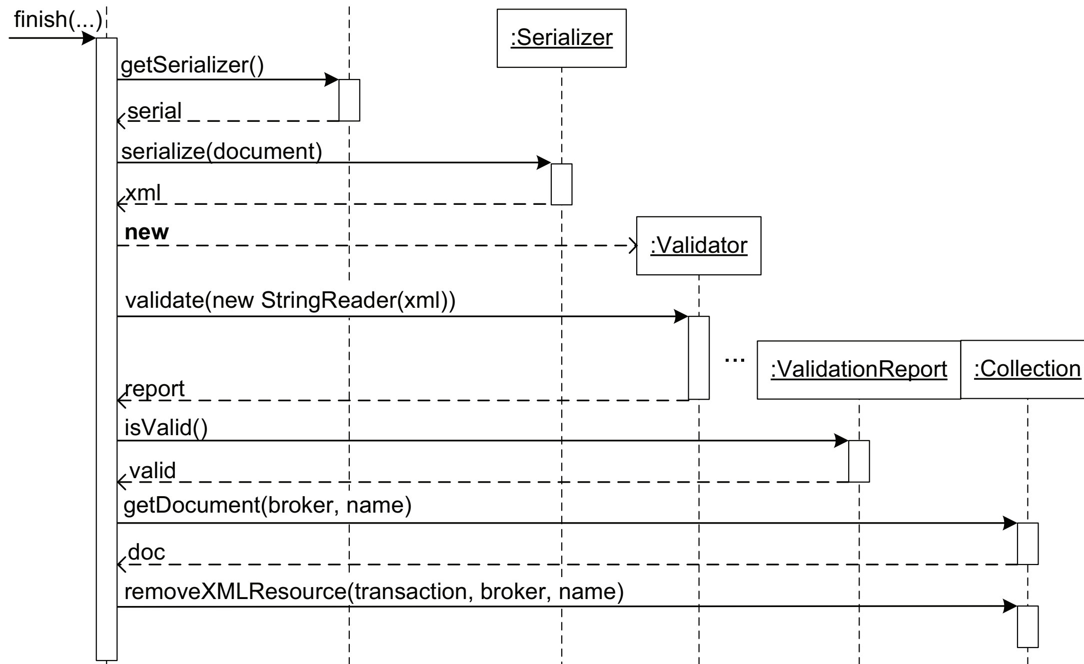


Abbildung 9.6: Sequenzdiagramm: Endphase des ValidationTriggers

Zugriff auf das Dokument

Um die Konsistenzprüfung des modifizierten Dokumentes durchführen zu können, muss die Version verwendet werden, die dem `ValidationTrigger` in der Methode `finish` übergeben wurde. Ein Zugriff auf das Dokument, welches im Datenbanksystem abgelegt wurde ist nicht möglich, da dieses im Fall einer Änderung über die XQuery-Update-Extension noch gesperrt (Schreibsperre) ist! Ein Versuch trotzdem lesend auf dieses Dokument zuzugreifen führt zu einem Deadlock, da der `ValidationTrigger` eine Lesesperre für das Dokument anfordert. Diese kann aber erst dann erteilt werden, wenn die Schreibsperre gelöst wird. Die Schreibsperre wird erst gelöst, wenn die Endphase des `ValidationTriggers` beendet ist. Der Ausgang der Konsistenzprüfung gibt an, wie weiter verfahren wird:

- Wenn die Prüfung ergibt, dass das modifizierte Dokument nicht valide/konsistent ist, werden alle vorgenommenen Änderungen ungeschehen gemacht, indem das neue Dokument durch das vorher erstellte Before-Image ersetzt wird.
- Ergibt die Prüfung, dass es sich um ein valides/konsistentes Dokument handelt, wird das Before-Image gelöscht.

9.4.2 Konfiguration des ValidationTriggers

Damit der ValidationTrigger vom Datenbanksystem verwendet wird, muss er konfiguriert werden. In Abschnitt 6.3 ist eine Beispielkonfiguration für einen Trigger angegeben. Wenn diese für den `ValidationTrigger` angepasst wird, sieht das wie folgt aus:

```
1 <collection xmlns="http://exist−db.org/collection−config/1.0">
2    <triggers>
3       <trigger event="update" class="org.exist.collections.triggers.ValidationTrigger">
4          <parameter ... />
5       </trigger>
6    </triggers>
7 </collection>
```

Listing 9.2: Konfiguration für den ValidationTrigger

Der `ValidationTrigger` wird bei dieser Konfiguration jedes Mal ausgeführt, wenn das Ereignis „update“ auftritt. Damit der `ValidationTrigger` in einer Collection ausgeführt wird muss er entweder in der Konfigurationsdatei dieser Collection, oder zumindest in der Konfigurationsdatei einer Collection, oberhalb in dem baumartigen Dateisystem liegen. Wenn der Trigger in einer anderen Collection konfiguriert ist, darf keine andere Collection auf dem Weg zwischen den beiden eine eigene Konfigurationsdatei der Collection besitzen.

9.4.3 Funktionsumfang des Triggers

Der Funktionsumfang des `ValidationTriggers` ist davon abhängig, bei welchen Ereignissen er ausgeführt wird. In den Kapiteln 8.5 und 6.3 wird erklärt, dass es für `DokumentTrigger` drei Ereignisse gibt, bei denen sie ausgeführt werden können. Die Ereignisse sind „insert“, „update“ und „delete“. Für den `ValidationTrigger` sind nur die ersten beiden Ereignisse relevant.

Konfiguration für das Ereignis „update“

Im ersten Szenario wird der `ValidationTrigger` so konfiguriert, dass er nur ausgeführt wird, wenn das Ereignis „update“ auftritt. Die Tabelle 9.1 zeigt, dass der neue `ValidationTrigger` in drei Fällen eine Validierung durchführen kann und das korrekte Ergebnis dieser Prüfung liefert. Der erste Fall ist, wenn ein Dokument in das Datenbanksystem eingefügt werden soll, welches ein anderes, bereits vorhandenes Dokument überschreibt. Die beiden anderen Fälle sind die Veränderung eines Dokumentes im Datenbanksystem durch einen XQuery-Update-Extension Ausdruck oder durch einen XUpdate Ausdruck. Der `ValidationTrigger` wurde speziell für die letzten beiden Fälle entwickelt, da in diesen bisher keine Prüfung der veränderten Dokumente stattfindet.

Im ersten Fall kommt es zu einer Überschneidung, da das Einfügen von Dokumenten von der impliziten Validierung abgedeckt wird. Dadurch kommt es in dem mit (*) markierten Fall zu einer doppelten Konsistenzprüfung. Einmal bevor es in das Datenbanksystem eingefügt wird und ein zweites Mal nachdem es eingefügt wurde, durch den ValidationTrigger. Zur Erkennung der mit (**) markierten Fälle ist der ValidationTrigger nicht notwendig, da schon vor der Ausführung der XQuery-Update-Extension/XUpdate Anfragen untersucht wird, ob sie Verstöße gegen die Wohlgeformtheit verursachen.

	Insert (mit Überschreiben)	XQuery	XUpdate
Konsistentes Dokument	Ja*	Ja	Ja
Syntaktisch falsch	Ja	Ja**	Ja**
Semantisch falsch	Ja	Ja	Ja

Tabelle 9.1: Funktionsumfang des ValidationTriggers

Konfiguration für die Ereignisse „insert“ und „update“

Wenn der `ValidationTrigger` so eingestellt ist, dass er bei den Ereignissen „insert“ und „update“ ausgeführt wird, wird der erste Fall in Tabelle 9.1 erweitert. Der `ValidationTrigger` wird dann jedes mal, wenn ein Dokument in das Datenbanksystem eingefügt wird, ausgeführt. Bei genauerer Betrachtung wird dabei die Feststellung gemacht, dass der `ValidationTrigger` in dieser Konfiguration die gesamte implizite Validierung von eXist ein zweites Mal durchführt. Zusammen mit der Tatsache, dass der `ValidationTrigger` die Prüfung in den Fällen, die vom Datenbanksystem nicht abgedeckt werden übernimmt, folgt daraus, dass der `ValidationTrigger` die gesamte implizite Validierung von eXist ersetzen kann.

Überblick

Bei der Konfiguration für das Ereignis „update“ tritt eine doppelte Prüfung auf die Konsistenz auf. Dies ist der Leistung des Datenbanksystems abträglich. Der andere Ansatz, bei dem die gesamte implizite Validierung durch den `ValidationTrigger` übernommen wird, hat den großen Nachteil, dass dadurch die Prüfung der Konsistenz erst dann erfolgt, wenn das Dokument bereits in das Datenbanksystem eingefügt ist. Bei konsistenten Dokumenten macht dies nichts aus. Wenn ein inkonsistentes Dokument in das Datenbanksystem eingefügt werden soll führt es dazu, dass ein Dokument zuerst in das Datenbanksystem eingefügt wird und nach der Konsistenzprüfung wieder entfernt werden muss. Aufgrund dieses Nachteils wird in Kauf genommen, dass eine doppelte Konsistenzprüfung auftreten kann.

9.5 Zusammenfassung

Zunächst wurde der Basisentwurf des `owlconsistency`-Moduls verfeinert dargestellt. Anschließend wurde der Entwurf für die Integration des `owlconsistency`-Moduls in die bestehende Architektur des Datenbanksystems erklärt. Gefolgt wurde dies vom Entwurf für die Erweiterung der Konfigurierbarkeit um einen OWL-Zustand. Zuletzt folgte der Entwurf eines Triggers, der die im Datenbanksystem fehlende Validierung bzw. Konsistenzprüfung bei einer Änderung durch ein XQuery-Update-Extension Ausdruck oder XUpdate Ausdruck ergänzt.

10 Technische Umsetzung

Im folgenden Abschnitt wird zunächst erklärt, welche Änderungen am Datenbanksystem vorgenommen wurden, um den neuen Validierungsmodus für OWL-Dokumente zu realisieren. Danach wird die Umsetzung des Systementwurfes für das `owlconsistency`-Modul vorgestellt. Anschließend wird gezeigt, wie das `owlconsistency`-Modul in das Datenbanksystem integriert wird, um alternativ XML-Validierung oder OWL-Konsistenzprüfung anzubieten. Daraufhin wird erklärt wie der `ValidationTrigger` umgesetzt wird. Abschließend werden die drei Validierungsmechanismen gegenübergestellt.

10.1 Erweiterung der Konfigurierbarkeit

Um die Konfigurierbarkeit zu erweitern, müssen Eingriffe an vier verschiedenen Stellen vorgenommen werden. Zunächst muss die Konfigurationsdatei (`conf.xml`) um die neuen Attribute erweitert werden. Im nächsten Schritt wird das XML-Schema der Konfigurationsdatei erweitert. Zuletzt müssen die Klassen `Configuration` und `XMLReaderObjectFactory` erweitert werden, da über sie intern auf die Konfigurationsparameter zugegriffen werden kann.

10.1.1 Konfigurationsdatei conf.xml

Die ersten Änderungen müssen an der in Abschnitt 8.3 erklärten Konfigurationsdatei `conf.xml`[1] vorgenommen werden. Hier müssen die beiden neuen Attribute `ip` und `port` zu dem Element `validation` hinzugefügt werden. Das Ergebnis sieht wie folgt aus:

```
1 <validation mode="owl" ip="127.0.0.1" port="1234">
2    ...
3 </validation>
```

Listing 10.1: Ausschnitt aus der Konfigurationsdatei

[1] Zu finden unter $EXIST_HOME/conf.xml

10.1.2 XML-Schema der Konfigurationsdatei

Anschließend muss das XML-Schema[2] der Konfigurationsdatei um die neuen Attribute erweitert werden. Diese neuen Attribute werden innerhalb der Definition des `validation`-Elementes definiert. Für das Attribut `mode` wurde ein eigener Datentyp, der auf dem `xs:string`-Datentyp basiert, definiert. Dieser muss um das Element `owl` erweitert werden (Zeile 10). Die Bedeutung der anderen drei Werte wird in Abschnitt 8.3.1 erklärt. Beide Attribute (Zeile 15 und 16) werden vom Datentyp `xs:string` definiert. Außerdem wird beiden Attributen ein Standardwert zugewiesen, damit auch bei einer alten Konfigurationsdatei Parameter für den Versuch eines Verbindungsaufbaus zum OWL-Reasoner vorhanden sind. Nach vollzogenen Änderungen sieht die Definition des `validation`-Elementes wie folgt aus:

```
1  <xs:element name="validation">
2    <xs:complexType>
3      ...
4      <xs:attribute name="mode" default="auto">
5        <xs:simpleType>
6          <xs:restriction base="xs:string">
7            <xs:enumeration value="auto"/>
8            <xs:enumeration value="no"/>
9            <xs:enumeration value="yes"/>
10           <xs:enumeration value="owl"/> <!-- OXDBS-spezifisch -->
11         </xs:restriction>
12       </xs:simpleType>
13     </xs:attribute>
14     <!-- Anfang OXDBS-spezifisch -->
15     <xs:attribute name="ip" type="xs:string" default="127.0.0.1"/>
16     <xs:attribute name="port" type="xs:string" default="1234"/>
17     <!-- Ende OXDBS-spezifisch -->
18   </xs:complexType>
19 </xs:element>
```

Listing 10.2: Ausschnitt aus dem XML-Schema der Konfigurationsdatei

10.1.3 Erweiterung der Klasse XMLReaderObjectFactory

Die Klasse `XMLReaderObjectFactory` muss um fünf Konstanten für den Zugriff auf die Attributswerte erweitert werden. Für den Validationsmodus, in dem sich das Datenbanksystem befindet, wird eine Konstante benötigt, da die Modi mit Integerzahlen kodiert sind. Die Werte 0 bis 2 entsprechen den bereits vorhandenen

[2] Zu finden unter $EXIST_HOME/schema/conf.xsd

Modi und -1 einem unbekannten Zustand. Daher wird der Modus `VALIDATION_OWL` mit der Zahl drei kodiert:

```
1 public final static int VALIDATION_UNKNOWN = −1;
2 public final static int VALIDATION_ENABLED = 0;
3 public final static int VALIDATION_AUTO = 1;
4 public final static int VALIDATION_DISABLED = 2;
5 public final static int VALIDATION_OWL = 3; // OXDBS−spezifisch
```

Listing 10.3: Ausschnitt aus XMLReaderObjectFactory.java

Für die Attribute `ip` und `port` werden je zwei Konstanten gebraucht. Die erste Variable dient dazu das Attribut im Elementkontext zu identifizieren, die zweite Variable dazu, um auf die Attributwerte über eine `HashMap`-Instanz zuzugreifen.

```
1 public final static String VALIDATION_IP_ATTRIBUTE = "ip";
2 public final static String PROPERTY_VALIDATION_IP = "validation.ip";
```

Listing 10.4: Ausschnitt aus XMLReaderObjectFactory.java

Außer den Konstanten müssen noch zwei Methoden der Klasse verändert werden. Die erste ist die Methode `convertValidationMode`, die eine `String`-Instanz übergeben bekommt, diese mit den möglichen Validationsmodi vergleicht und den entsprechenden Integerwert zurückgibt. In der Methode ist eine weitere Bedingung einzufügen, welche die übergebene `String`-Instanz mit dem Wert „`owl`“ vergleicht und im Fall der Übereinstimmung den Rückgabewert auf drei setzt.

Die andere Methode heißt `setReaderValidationMode` und dient dazu, die dem Validierungsmodus entsprechenden Einstellungen am verwendeten XML-Parser (Apache Xerces 2) vorzunehmen. Da das Datenbanksystem XML-Dokumente zur Konfiguration verwendet, wird der XML-Parser auch verwendet, wenn der Modus `owl` eingestellt ist. Daher muss auch in diesem Fall eine Konfiguration des XML-Parsers vorgenommen werden. Die Wahl fiel darauf, den XML-Parser im `owl`-Modus genau wie im Modus `auto` zu konfigurieren. Das ist der Fall, da im Modus `auto` wie in Abschnitt 8.3 beschrieben, eine syntaktische Validierung möglich ist, aber nicht erzwungen wird.

10.1.4 Erweiterung der Klasse Configuration

In der Klasse `Configuration` (org.exist.util) muss die Methode `configureValidation` erweitert werden, damit die beiden neuen Parameter zur Laufzeit verfügbar sind und nicht nach der Initialisierung verworfen werden. Der Methode wird ein Objekt vom Typ `Element` übergeben, das alle Daten (z.B. Attribute), die in diesem Element enthalten sind, enthält.

```
1 String ip = validation.getAttribute(
2 XMLReaderObjectFactory.VALIDATION_IP_ATTRIBUTE);
3 if (ip != null){
4     config.put(XMLReaderObjectFactory.PROPERTY_VALIDATION_IP, ip);
5     LOG.debug(XMLReaderObjectFactory.PROPERTY_VALIDATION_IP + ": " + config
          .get(XMLReaderObjectFactory.PROPERTY_VALIDATION_IP));
6 }
```

Listing 10.5: Veränderungen der Klasse Configuration

Die neuen Attribute müssen, genau wie das Attribut für den Modus der Validierung, aus dem Objekt ausgelesen werden. Dazu wird im Fall des Attributes `ip` die Variable `VALIDATION_IP_ATTRIBUTE` aus der Klasse `XMLReaderObjectFactory` verwendet. Wenn das Attribut ausgelesen ist, wird es zusammen mit der Variable `PROPERTY_VALIDATION_IP` in die HashMap `config` eingefügt. Nach dem Einfügen des Attributes in die HashMap, wird ein Eintrag in das Logbuch des Datenbanksystems gemacht, der sich aus denselben Werten wie der HashMap-Eintrag zusammensetzt. Das Einfügen des Attributes für den `port` funktioniert analog zu `ip`, nur dass die in Kapitel 10.1.3 beschriebenen Variablen von `port` verwendet werden müssen. Über die HashMap kann während der Ausführung des Datenbanksystems jede Klasse, die über eine Instanz der Klasse `Configuration` verfügt, mit der entsprechenden Variable aus der `XMLReaderObjectFactory`, auf die Attribute aus der Konfigurationsdatei zugreifen.

10.2 Implementierung des OWL-Moduls

Das `owlconsistency`-Modul übernimmt die gesamte Konsistenzprüfung von OWL-Dokumenten. Implementiert ist der Basisentwurf aus Kapitel 9. Zusätzlich sind die Klassen, die für die Umsetzung des erweiterten Entwurfes nötig sind, angelegt. Die Klassen für die Umsetzung des erweiterten Entwurfes sind bereits eingebunden, besitzen aber keine Funktionalität.

Die Klasse `OWLChecker` stellt die Außenschnittstelle des Moduls `owlconsistency` dar und sorgt dafür, dass das Dokument, welches ihr übergeben wird auf seine Konsistenz geprüft wird. Dazu delegiert sie die Arbeit an die Klassen `CustomizeDocument` und `OWLValidation`. In der Klasse `OWLChecker` werden alle Fehlermeldungen, die in einer der aufgerufenen Klassen auftreten, in eine Fehlermeldung des Datenbanksystems (`EXistException`) umgewandelt und weitergeleitet.

Die Klasse `CustomizeDocument` hat die Aufgabe das Dokument, das geprüft werden soll in einer temporären Datei auf der Festplatte abzulegen, damit es vom OWL-

Reasoner von dort gelesen werden kann. In den Aufgabenbereich dieser Klasse fällt auch das abschließende Löschen der Datei.

Die Klasse `OWLValidation` ist für die Durchführung der Konsistenzprüfung verantwortlich. Sie verwendet dabei das Modul `jracer` um die Kommandos zum Einlesen des Dokumentes und zu dessen Konsistenzprüfung an RacerPro zu übermitteln. Wenn bei der Konsistenzprüfung ein Fehler auftritt oder ein inkonsistentes Dokument geprüft wird, wird dies mittels einer Fehlermeldung an die Klasse `OWLChecker` mitgeteilt.

10.2.1 Die Klasse OWLChecker

Um eine Konsistenzprüfung durchzuführen wird eine Instanz der Klasse `OWLChecker` angelegt. Die Abbildung 10.1 zeigt, dass die Klasse zwei Konstruktoren besitzt. Der erste bekommt keine Übergabeparameter und setzt die Variable für die IP, mit der die Verbindung zum OWL-Reasoner aufgebaut wird, auf `127.0.0.1` und den Port auf `1234`. Der zweite Konstruktor bekommt als Übergabeparameter eine `String`-Instanz, in der die IP des OWL-Reasoners steht und einen `Integer`, der die Nummer des Ports enthält. Nach dem Erzeugen der Instanz der Klasse wird die Methode `validate` aufgerufen. Dieser Methode wird das zu validierende Dokument in der Form einer `InputSource`-Instanz übergeben.

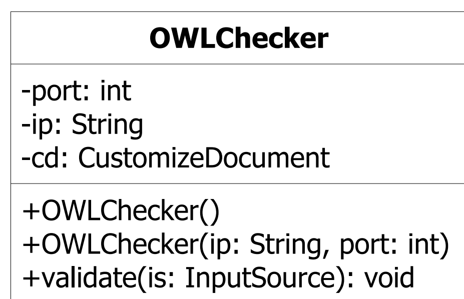


Abbildung 10.1: Klassendiagramm der Klasse OWLChecker

In der Methode `validate` wird eine Instanz der Klasse `CustomizeDocument` erzeugt, die genauer in Abschnitt 10.2.2 erklärt wird. Anschließend wird die Methode `toDisk(InputSource)` aufgerufen. Als Ergebnis liefert die Methode den Pfad der temporären Datei, in der die zu überprüfenden Daten enthalten sind. Anschließend wird eine Instanz der Klasse `OWLValidation` erzeugt, welche als Parameter die IP und den Port des OWL-Reasoners übergeben bekommt. Nach der Erstellung der Instanz wird die Methode `validateNRQL` aufgerufen, die als Parameter den Pfad zu der temporären Datei, der von der `CustomizeDocument`-Instanz geliefert wurde, bekommt. Wenn dabei keine Fehler auftreten, wurde ein konsistentes Dokument geprüft. Tritt

bei der Ausführung der Konsistenzprüfung durch die Klasse `OWLValidation` eine `OWLException` auf, wird diese gefangen. Mit der Nachricht der `OWLException` wird eine neue `EXistException` generiert um mitzuteilen, dass das Dokument inkonsistent ist. Nachdem das Ergebnis der Konsistenzprüfung feststeht, wird die temporäre Datei, wenn sie durch die `CustomizeDocument`-Instanz erstellt wurde mit deren Methode `delete` wieder gelöscht.

10.2.2 Die Klasse CustomizeDocument

Die Aufgabe der Klasse `CustomizeDocument` ist es, das OWL-Dokument in die korrekte Form zu bringen, damit es an den OWL-Reasoner übergeben werden kann. Die Abbildung 10.2 zeigt das Klassendiagramm der Klasse `Customizedocument`, das zeigt welche Methoden und Klassenvariablen zur Verfügung stehen.

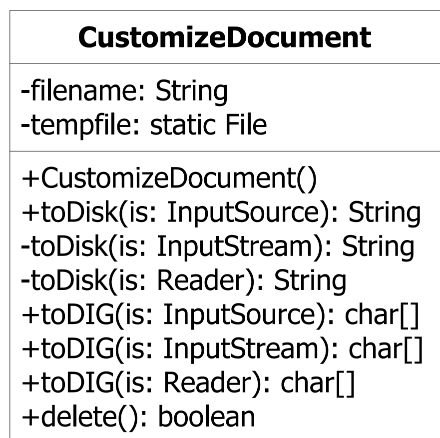


Abbildung 10.2: Klassendiagramm der Klasse CustomizeDocument

Die Klasse bietet eine Methode `toDisk(InputSource)` an, welche das Dokument in eine temporäre Datei auf der Festplatte schreibt. Der Methode wird eine `Input-Source`-Instanz, mit der auf das Dokument zugegriffen wird, übergeben. Als nächstes wird versucht, den ByteStream (`InputStream`), der in der `InputSource`-Instanz enthalten ist, zu extrahieren. Ist kein ByteStream enthalten, wird versucht einen CharacterStream (`Reader`) zu extrahieren. Wenn weder ein ByteStream noch eine CharakterStream vorhanden sind, wird untersucht, ob die `systemId` der `InputSource`-Instanz den Pfad zu einer bereits angelegten Datei enthält. Ist dies nicht der Fall, wird eine Fehlermeldung erzeugt, da keine Daten vorhanden sind, die ausgegeben werden können. Wenn einer der beiden Streams extrahiert wird, kann mit ihm wieder die Methode `toDisk(Reader/InputStream)` aufgerufen werden. Diese Methode erzeugt

eine temporäre Datei auf der Festplatte. Die Dateinamen beginnen mit `validation` und enden mit `.owl`, dazwischen sind zufällige Zahlen. In diese Datei wird der Inhalt des OWL-Dokumentes gespeichert. Das Speichern wird in einer Schleife erledigt, die immer 8 KB große Stücke an das Ende der Datei anfügt, bis das Dokument vollständig in der temporären Datei enthalten ist. Tritt bei diesem Vorgang ein Fehler auf, so wird der Vorgang mit einer Fehlermeldung abgebrochen. Als Ergebnis gibt die Methode den Pfad der neu erstellten temporären Datei zurück oder den in der `systemId` gespeicherten Pfad.

Zusätzlich zu den Methoden die ein Dokument auf die Festplatte speichern, sind Methoden angelegt, die das Dokument in das DIG Format umwandeln sollen, damit es über das DIG-Interface an den OWL-Reasoner übergeben werden kann. Diese Methoden sind bisher nicht implementiert, sondern nur als Rumpf vorhanden.

Für den Fall, dass das Dokument auf die Festplatte geschrieben wird, wird eine Methode angeboten, mit der das Dokument nach der Prüfung durch den OWL-Reasoner wieder von der Festplatte gelöscht werden kann.

10.2.3 Die Klasse OWLValidation

Die Abbildung 10.3 zeigt das Klassendiagramm für die Klasse `OWLValidation` und damit ihre verfügbaren Methoden und Klassenvariablen. Im Konstruktor werden die IP und der Port für die Verbindung zum OWL-Reasoner übergeben und in dem Objekt gespeichert. Die Klasse stellt zwei Methoden für die Kommunikation mit dem OWL-Reasoner zur Verfügung. Die eine Methode heißt `validateDIG` und ist für die Kommunikation mit dem OWL-Reasoner über das DIG Protokoll gedacht. Diese Methode ist noch nicht implementiert, da die Version 1.1 der DIG-Schnittstelle, wie in Kapitel 3.8 beschrieben, noch Mängel aufweist.

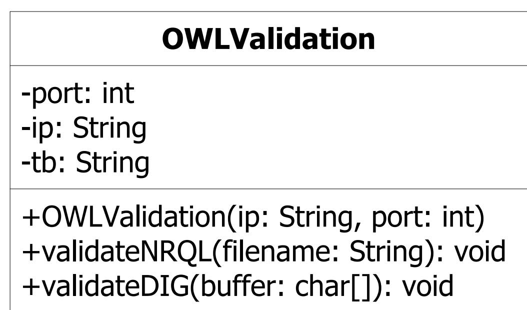


Abbildung 10.3: Klassendiagramm der Klasse OWLValidation

Stattdessen ist die Methode `validateNRQL` implementiert. Die Abbildung 10.4 zeigt, was aus Sicht der Klasse `OWLValidation` nötig ist, um eine Konsistenzprüfung durch-

zuführen. Zuerst muss eine Verbindung zu RacerPro aufgebaut werden, damit Befehle an ihn übermittelt werden können. Anschließend muss der Befehl übermittelt werden, der dafür sorgt, dass das Dokument von RacerPro geladen wird. Nach dem erfolgreichen Laden wird der Befehl übermittelt, der RacerPro dazu veranlasst die T-Box des übermittelten Dokumentes auf ihre Konsistenz zu prüfen. Wenn die T-Box konsistent ist, wird als letztes der Befehl übermittelt, der veranlasst, dass RacerPro die A-Box des Dokumentes gegen dessen T-Box prüft. Ergibt diese Prüfung, dass die A-Box in Bezug auf die T-Box konsistent ist, handelt es sich um ein konsistentes Dokument. Sollte die Konsistenzprüfung nicht den hier skizzierten Verlauf nehmen, werden die in der Abbildung angegebenen Fehlermeldungen erzeugt.

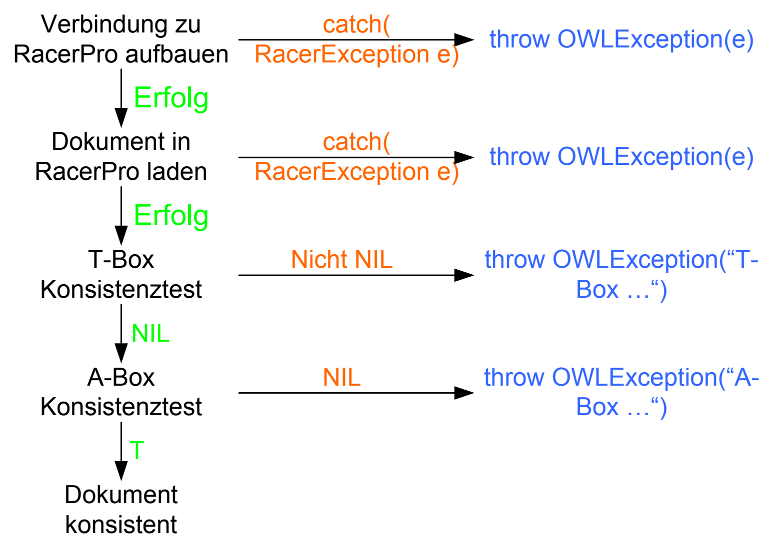


Abbildung 10.4: Ablauf der Konsistenzprüfung

Für die Kommunikation mit RacerPro wird das Paket `jracer` verwendet, das von Racer-Systems zur Verfügung gestellt wird. Um eine Verbindung aufbauen zu können wird eine Instanz der Klasse `RacerServer` erstellt, wobei im Konstruktor die IP und der Port, unter dem RacerPro zu erreichen ist übergeben werden. Der Aufbau der Verbindung erfolgt durch den Aufruf der Methode `openConnection` der Klasse `RacerServer`. Ist dieser Verbindungsaufbau erfolgreich, so können beliebige nRQL-Befehle an RacerPro übermittelt werden. Zu diesem Zweck bietet die Klasse `RacerServer` die Methode `send` an. Die Methode `send` hat als Rückgabewert eine `String`-Instanz, die das Ergebnis des Befehls enthält. Um einen Konsistenztest eines Dokumentes durchzuführen sind die Befehle notwendig, welche die in Abbildung 10.4 genannten Aktionen auslösen. Diese Befehle sind:

1. (full-reset)
2. (owl-read-file “filename“)
3. (check-tbox-coherence)
4. (abox-consistent-p)

Als erstes wird der Befehl `(full-reset)` an die Methode `send` der Klasse `RacerServer` übergeben. Dieser Befehl ist für die Konsistenzprüfung selbst nicht notwendig, sondern er dient dazu RacerPro in seinen Anfangszustand zurückzusetzen, was bedeutet, dass alle Daten, die bereits geladen wurden, verworfen werden. Der Befehl wird verwendet, damit das Ergebnis der Konsistenzprüfung unter keinen Umständen durch bereits in RacerPro vorhandene Daten verändert wird.

Nachdem RacerPro in seinen Ausgangszustand zurückversetzt wurde, wird der Befehl `(owl-read-file "filename")` an die Methode `send` übergeben. Dieser Befehl hat mit der eigentlichen Konsistenzprüfung noch nichts zu tun. Er sorgt lediglich dafür, dass RacerPro das Dokument, welches in einer temporären Datei auf der Festplatte liegt, einliest.

Für die ersten beiden Befehle ist es nicht notwendig den Rückgabewert der Methode `send` zu betrachten, da im Fehlerfall eine `RacerException` auftritt, die gefangen wird und in einer `OWLException` an die aufrufende Klasse übergeben wird. Nachdem das Dokument von RacerPro eingelesen ist, wird mit der tatsächlichen Konsistenzprüfung begonnen. Dazu wird der Befehl `(check-tbox-coherence)` an die Methode `send` der Klasse `RacerServer` übergeben. Der Befehl veranlasst RacerPro dazu die T-Box, die in dem geladenen Dokument enthalten ist, auf ihre Konsistenz zu prüfen. Der Rückgabewert der Methode `send` enthält eine Liste aller nichtentscheidbarer Konzepte. Die T-Box ist nur dann konsistent, wenn keine nichtentscheidbaren Konzepte vorhanden sind. Daher muss untersucht werden, ob der Rückgabewert eine leere Liste in Form des Wertes `NIL`, enthält. Enthält der Rückgabewert etwas anderes als die leere Liste, wird die Konsistenzprüfung wie in Abbildung 10.4 dargestellt, mit einer Fehlermeldung in Form einer `OWLException` abgebrochen.

Wenn eine konsistente T-Box vorliegt, wird der Befehl `(abox-consistent-p)` an die Methode `send` übergeben. Dieser Befehl veranlasst die Überprüfung, ob die A-Box konsistent in Bezug auf die T-Box ist. Der Rückgabewert der Methode `send` kann in diesem Fall den Wert `T` oder den Wert `NIL` annehmen. Wenn der Rückgabewert den Wert `NIL` hat, bedeutet dies, dass die A-Box nicht konsistent zu der T-Box ist. Das hat zur Folge, dass die Konsistenzprüfung mit einer Fehlermeldung in Form einer `OWLException` abgebrochen wird. Der Rückgabewert `T` sagt aus, dass die A-Box konsistent in Bezug auf die T-Box ist.

Die Prüfung, ob die A-Box konsistent in Bezug auf die T-Box ist, stellt den letzten Schritt der Konsistenzprüfung dar. Ein Dokument ist genau dann konsistent, wenn beim hier beschriebenen Ablauf der Konsistenzprüfung keine Fehlermeldung an die aufrufende Klasse weitergegeben wird.

10.2.4 Die Klasse OWLException

Die OWLExceptions sind von der Exception-Klasse abgeleitet und bieten die Möglichkeit Fehlermeldungen zu erzeugen, die lediglich während des Konsistenztests in der Klasse `OWLValidation` auftreten. Die Abbildung 10.5 zeigt, welche Konstruktoren für die Klasse `OWLException` verfügbar sind.

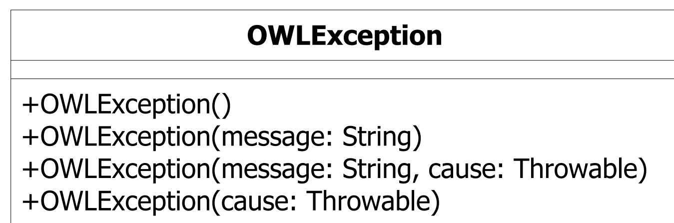


Abbildung 10.5: Klassendiagramm der Klasse OWLException

10.2.5 Klassen für den erweiterten Entwurf

Für eine Implementierung des erweiterten Entwurfes aus Abschnitt 7.4.2 sind weitere Klassen nötig, die Methoden für die erweiterte Funktionalität zur Verfügung stellen. Der erweiterte Entwurf wird nicht umgesetzt, die dafür benötigten Klassen und die Rümpfe der Methoden werden vorsorglich angelegt. Dadurch wird es für ein auf dieser Arbeit aufbauendes Projekt vereinfacht den erweiterten Entwurf zu implementieren. Um den erweiterten Entwurf zu implementieren werden folgende Klassen benötigt:

- DocumentContent
- TBFMerger
- TBoxSearch
- ABFMerger
- ABoxSearch
- TAMerger

Die Abbildung 10.6 zeigt die Klassendiagramme für die zusätzlichen Klassen des erweiterten Entwurfes. Für jede Klasse ist angegeben, welche Methode sie besitzen soll. Im Folgenden wird beschrieben welche Funktionsweise für die einzelnen Klassen vorgesehen ist.

<<ABoxSearch>>	<<ABFMerger>>
+searchABox(is: InputSource): String	+merge(boxfrag: String[])
<<TBoxSearch>>	<<TBFMerger>>
+searchTBox(is: InputSource): String	+merge(boxfrag: String[])
<<TAMerger>>	<<DocumentContent>>
+merge(tbox: String, abox: String): String	+evaluate(is: InputSource): int

Abbildung 10.6: Zusätzliche Klassen für erweiterten Entwurf

Die Klasse `DocumentContent` besitzt eine Methode `evaluate`. Dieser Methode wird die `InputSource`-Instanz übergeben, die der Methode `validate` der Klasse `OWLChecker` übergeben wird. Die Methode `evaluate` untersucht das Dokument, das in der `InputSource`-Instanz enthalten ist, um zu entscheiden, ob es eine T-Box, eine A-Box oder beides enthält. Das Ergebnis dieser Untersuchung wird im Rückgabewert der aufrufenden Klasse mitgeteilt.

Die Klasse `TBFMerger` besitzt eine Methode `merge`, die als Übergabeparameter eine `Stringarray`-Instanz erhält. Jede im Array enthaltene `String`-Instanz enthält ein T-Box-Fragment. Die Methode `merge` nimmt diese T-Box-Fragmente und fügt sie in einer `String`-Instanz zu einer T-Box zusammen. Diese `String`-Instanz wird als Rückgabewert an die aufrufende Klasse übergeben.

Neben der Klasse `TBFMerger` existiert die Klasse `ABFMerger`, welche eine Methode `merge` bietet, die genauso arbeitet wie die Methode `merge` der Klasse `TBFMerger`. Der Unterschied besteht darin, dass in der Klasse `ABFMerger` mit A-Box-Fragmenten gearbeitet wird.

Die Klasse `TBoxSearch` hat eine Methode `searchTBox`. Dieser Methode wird die `InputSource`-Instanz übergeben, die der Methode `validate` der Klasse `OWLChecker` übergeben wird. In der Methode `searchTBox` werden alle T-Box-Fragmente, die zu

dem Dokument, welches in der `InputSource`-Instanz enthalten ist, aus dem Datenbanksystem geladen. Die T-Box-Fragmente liegen nach dem Lesen als Instanz der Klasse `DocumentImpl` vor. Mit der Methode `serialize` der Klasse `Serializer` wird der Dokumentinhalt aus der `DocumentImpl`-Instanz in eine `String`-Instanz extrahiert. Dies geschieht für alle T-Box-Fragmente. Anschließend wird eine `Stringarray`-Instanz erzeugt, welches alle T-Box-Fragmente enthält. Alle T-Box-Fragmente sollen zu einem T-Box-Dokument zusammengefasst werden. Dazu wird eine Instanz der Klasse `TBFMerger` erzeugt und die Methode `merge` dieser Klasse mit der `Stringarray`-Instanz aufgerufen. Die Methode `merge` liefert eine `String`-Instanz als Rückgabewert, die das T-Box-Dokument enthält.

Die Klasse `ABoxSearch` stellt auf A-Box Seite das Gegenstück zu der Klasse `TBoxSearch` dar. Die Klasse `ABoxSearch` bietet identische Methoden zu der Klasse `TBoxSearch`. Der Unterschied liegt darin, dass Methode `searchABox` die gesamte A-Box statt der T-Box liefert.

Die Klasse `TAMerger` bietet eine Methode `merge`. Dieser Methode werden zwei `String`-Instanzen übergeben. Von diesen `String`-Instanzen enthält eine das T-Box-Dokument und die andere das A-Box-Dokument. Die Methode nimmt diese beiden Dokumente und erzeugt daraus ein wohlgeformtes XML-Dokument, welches sowohl die T-Box als auch die A-Box enthält. Die `String`-Instanz, in der sich das Dokument befindet wird als Rückgabewert an die aufrufende Klasse übergeben.

10.2.6 Probleme

Bei der Implementierung der Klasse `CustomizeDocument` trat das Probleme auf, dass aus der übergebenen `InputSource`-Instanz nicht direkt die Dokumentendaten gelesen werden konnten. Dabei war es notwendig zu untersuchen, ob die Instanz einen `ByteStream` oder einen `CharacterStream`, der auf die Festplatte geschrieben werden muss, enthält. Als dritte Möglichkeit kann die `InputSource`-Instanz eine `String`-Instanz enthalten, der einen Pfad im Dateisystem enthält, an dem der Dokumentinhalt in einer temporären Datei liegt. Diese Untersuchung findet in der Klasse `CustomizeDocument` statt.

Wenn das Dokument mit einer `Reader`-Instanz eingelesen wurde, wurde genau wie im Fall einer `InputStream`-Instanz versucht, immer volle 8KB (8192 Byte) zu lesen. Die `Reader`-Instanz liefert aber nicht die exakten 8KB, sondern meistens 1-5 Byte weniger. Der Grund dafür, weshalb nicht volle 8192 Byte gelesen werden, konnte nicht ermittelt werden. Dadurch musste in diesem Fall dafür gesorgt werden, dass immer genau die eingelesene Anzahl an Zeichen in die temporäre Datei geschrieben wurde, da sonst alte Daten an falsche Stellen des Dokumentes eingefügt wurden.

10.3 Integration des OWL-Moduls in eXist

In diesem Abschnitt wird beschrieben, wie das `owlconsistency`-Modul, das die eigentliche Konsistenzprüfung der Dokumente durchführt, in das eXist Datenbanksystem integriert wird. Damit die Konsistenzprüfung von Dokumenten genauso funktioniert wie die Validierung von XML-Dokumenten, muss das `owlconsistency`-Modul in die Klassen des Datenbanksystems eingefügt werden, in denen die Validierungseinheit aufgerufen wird. In Abschnitt 9.2 wurde erklärt, dass die Änderungen an der Klasse `Validator` und der Klasse `Collection` vorgenommen werden müssen. Im Folgenden wird beschrieben, welche Änderungen an den Klassen vorgenommen werden.

10.3.1 Collection

In der Klasse `Collection` im Paket `org.exist.collections` muss die Methode `validateXMLResource(Txn, DBBroker, XmldbURI, InputSource)` erweitert werden, damit eine Konsistenzprüfung von OWL-Dokumenten möglich ist. Dazu wird in dieser Methode eine Variable `mode` eingeführt. Dieser Variable soll der konfigurierte Validierungsmodus zugewiesen werden. Dazu wird die Methode `getProperty` der Klasse `CollectionConfiguration` mit der Konstante `PROPERTY_VALIDATION_MODE` der Klasse `XMLReaderObjectFactory` aufgerufen.

In der Klasse `Collection` wird eine innere Schnittstelle (Interface) namens `ValidationBlock` definiert. Die Schnittstelle besitzt die Methode `run`, der eine Instanz der Klasse `IndexInfo` übergeben wird. In der Methode `validateXMLResource` wird eine Instanz der Schnittstelle `ValidationBlock` erstellt, dabei wird die Methode `run` implementiert. Die weiteren Änderungen an der Klasse `Collection` finden in der Implementierung dieser Methode statt. Wie in Abschnitt 8.1 beschrieben, fordert diese Methode eine `XMLReader`-Instanz an, der anschließend das Dokument zum Einlesen in der Methode `parse` übergeben wird. Aus diesem Grund ist in der Methode eine Fallunterscheidung notwendig, da mit einer `XMLReader`-Instanz keine OWL-Dokumente auf ihre Konsistenz geprüfen werden können. Bevor mit der Validierung/Konsistenzprüfung begonnen werden kann, wird geprüft, ob der OWL-Modus eingestellt ist. Ist dies der Fall, wird eine Instanz der `OWLChecker`-Klasse aus dem Modul `owlconsistency` erstellt. Im Konstruktor müssen die IP und der Port, unter dem der OWL-Reasoner erreichbar ist, übergeben werden. Diese beiden Werte werden analog zum Validierungsmodus (`mode`) mit der Methode `getProperty` der Klasse `CollectionConfiguration` erfragt.

```
...
info.setReader(reader, null);
/* consistency check start (OXDBS specific) */
if (mode.toUpperCase() == "OWL"){
    OWLChecker owl = new OWLChecker(ip, Integer.getInteger(port));
    owl.validate(source);
    final InputStream is = source.getByteStream();
    if (is != null){
        is.reset();
    } else {
        final Reader cs = source.getCharacterStream();
        if (cs != null)
            cs.reset();
    }
}
/* consistency check end */
reader.parse(source);
...
```

Listing 10.6: Ausführung der OWL-Konsistenzprüfung (Klasse Collection)

Nach der Erzeugung der `OWLChecker`-Instanz wird deren Methode `validate` mit einer `InputSource`-Instanz namens `source` aufgerufen. Tritt während dieser Ausführung keine Exception auf, war die Konsistenzprüfung des OWL-Dokumentes erfolgreich. Wenn die Konsistenzprüfung ergibt, dass ein konsistentes Dokument in die Datenbank eingefügt werden soll, dann muss `source` wieder in den Ursprungszustand versetzt werden, damit die `XmlReader`-Instanz das Dokument daraus lesen kann. Um `source` in seinen Ausgangszustand zurückzusetzen, wird zuerst versucht mit der Methode `getByteStream` eine `InputStream`-Instanz zu extrahieren. Ist die Extrahierung erfolgreich, dann wird die `InputStream`-Instanz mit der Methode `reset` zurückgesetzt. Kann keine `InputStream`-Instanz extrahiert werden, dann wird mit der Methode `getCharacterStream` versucht, eine `Reader`-Instanz zu entnehmen. Wenn keine `Reader`-Instanz herausgenommen werden kann, dann ist keine Zurücksetzung notwendig, da der Pfad des zu prüfenden Dokumentes in der Membervariable `systemId` übergeben wurde. Wurde eine `Reader`-Instanz extrahiert, dann wird dieser mit der Methode `reset` zurückgesetzt. Schlägt die Methode `reset` fehl, dann tritt eine `IOException` auf, die vom Datenbanksystem gefangen und ausgewertet wird.

Befindet sich das Datenbanksystem nicht im OWL-Modus, werden XML-Dokumente mit dem XML-Parser validiert, so als existierte keine Erweiterung des Datenbanksystems.

10.3.2 Validator

Die Klasse `Validator` im Paket `org.exist.validation` muss erweitert werden. In die Klasse werden drei neue Instanzvariablen `ip`, `port` und `mode` eingefügt. Den neuen Variablen werden beim Aufruf des Konstruktors die Werte zugewiesen, die in der Konfigurationsdatei des Datenbanksystems angegeben sind. Um auf die Konfigurationsparameter zugreifen zu können, wird die Methode `getProperty` der Klasse `Configuration` aufgerufen, als Parameter wird die entsprechende Konstante aus der Klasse `XMLReaderObjectFactory` übergeben.

```
1 ip = (String) config.getProperty(XMLReaderObjectFactory.
      PROPERTY_VALIDATION_IP);
```

Listing 10.7: Beispiel aus Validator.java

Die weiteren Änderungen werden in der Methode `validate(Reader, String)` vorgenommen. Wie in Kapitel 8.2 beschrieben, ist diese Methode für die Konfiguration des XML-Parsers und das Anstoßen der Validierung der XML-Dokumente zuständig. Aus diesem Grund ist in der Methode eine Fallunterscheidung notwendig. Bevor mit der Validierung/Konsistenzprüfung begonnen werden kann, wird geprüft, ob der OWL-Modus eingestellt ist. Ist dies der Fall, wird eine Instanz der Klasse `OWLChecker` aus dem Modul `owlconsistency` erstellt. Im Konstruktor wird der Klasse die `ip` als `String`-Instanz, und der `port` als `Integer` übergeben.

```
1  ...
2  report.start(); // eXist code
3  if (mode.toUpperCase() == "OWL"){
4     OWLChecker owl = new OWLChecker(ip, Integer.getInteger(port));
5     owl.validate(source);
6  } else {
7     reader.parse(source); // eXist code
8  }
9  report.stop(); // eXist code
10 ...
```

Listing 10.8: Ausführung der OWL-Konsistenzprüfung (Klasse Validator)

Nach der Erzeugung der `OWLChecker`-Instanz wird deren Methode `validate` aufgerufen. Als Übergabeparameter erhält sie die `InputSource`-Instanz `source`. Tritt während dieser Ausführung keine Exception auf, war die Konsistenzprüfung des OWL-Dokumentes erfolgreich.

Ist das Datenbanksystem so konfiguriert, dass keine Konsistenzprüfung durchgeführt werden soll, dann wird das OWL-Modul nicht ausgeführt. Stattdessen werden XML-Dokumente mit dem XML-Parser validiert.

10.3.3 Aufgetretene Probleme

Bei der Erweiterung der Klasse `Collection` traten zu Beginn Probleme auf, da versucht wurde die Validierungseinheit durch die Konsistenzprüfungseinheit zu ersetzen. Dies hatte den Effekt, dass die Konsistenzprüfung durchgeführt wurde. Im weiteren Verlauf des Einfügeprozesses kam es aber zu Fehlern, da die Validierungseinheit auch für das Umwandeln der OWL-/XML-Dokumente in die datenbankinterne `DocumentImpl`-Instanz zuständig ist. Dadurch dass die `DocumentImpl`-Instanz nicht vorhanden war, wurde eine Exception ausgelöst und der Einfügevorgang wurde abgebrochen, auch wenn das Dokument konsistent war. Aus diesem Grund wurde die Validierungseinheit nicht ersetzt, sondern erweitert. Wenn das Datenbanksystem im Modus `owl` ausgeführt wird, dann wird vor der Ausführung der Validierungseinheit die Konsistenzprüfungseinheit ausgeführt.

Ist das Datenbanksystem für die Verwaltung von OWL-Dokumenten konfiguriert, dann wird ein Dokument, das eingefügt wird, zuerst auf seine Konsistenz geprüft. Dafür wird das Dokument aus der `InputSource`-Instanz gelesen. Damit es eingefügt werden kann, muss es anschließend der Methode `parse` der Klasse `XmlReader` übergeben werden. Wenn die `InputSource`-Instanz einen ByteStream (`InputStream`) oder einen CharakterStream (`Reader`) enthält, dann führt dies zu einer Fehlermeldung (`IOException`). Dies liegt daran, dass das Dokument bereits komplett eingelesen wurde und der „Stream" am Ende des Dokumentes angekommen ist. Um das Dokument erneut lesen zu können muss die `InputSource`-Instanz in den Ursprungszustand zurückgesetzt werden. Dies wird getan indem der vorhandene „Stream" aus der `InputSource`-Instanz extrahiert und mit der Methode `reset` zurückgesetzt wird.

10.4 ValidationTrigger

Wie bereits in Abschnitt 9.4 erläutert, wird als Grundlage des `ValidationTriggers`, der vom Datenbanksystem mitgelieferte `HistoryTrigger` verwendet. Dieser erstellt in der Vorbereitungsphase des Triggers bei jeder Änderung eine Kopie des Dokumentes, damit jederzeit auf eine frühere Version des Dokumentes zugegriffen werden kann.

10.4.1 Vorbereitungsphase des ValidationTriggers

In der modifizierten Vorbereitungsphase (`prepare`-Methode) des Triggers, dargestellt in Abbildung 9.5, wird zuerst mit der `getOrCreateCollection`-Methode der `DBBroker`-Klasse die Ziel-Collection (default: /history/$dbpath) für die Kopie des Dokumentes (Before-Image) angefordert. Diese `Collection`-Instanz wird sofort mit

der `saveCollection`-Methode der Klasse `DBBroker` in das Datenbanksystem gespeichert, da es sein kann, dass sie im vorhergehenden Schritt erst erzeugt wurde. Im Anschluss daran wird mit der Methode `copyXMLResource` der Klasse `DBBroker` eine Kopie des zu verändernden Dokumentes in die Ziel-Collection kopiert. Nachdem dieses Before-Image erzeugt wurde ist die Vorbereitungsphase des Triggers beendet.

10.4.2 Endphase des ValidationTriggers

Bevor die nach der Vorbereitungsphase vorgenommenen Änderungen endgültig in das Datenbanksystem übernommen werden, wird die in Abschnitt 9.4 entwickelte Abschlussphase des ValidationTriggers ausgeführt. Die Abschlussphase (`finish`-Methode) des Triggers wird in Abbildung 9.6 dargestellt und bekommt das modifizierte Objekt im Speicher als `DocumentImpl`-Objekt übergeben. Dieses `DocumentImpl`-Objekt enthält zusätzlich zu dem Inhalt des Dokumentes alle, für das Datenbanksystem interessanten Metadaten zu dem Dokument. Um aus einem Objekt vom Typ `DocumentImpl` die wohlgeformten XML-Daten zu extrahieren, wird eine `Serializer`-Instanz benötigt, die vom `DBBroker` angefordert werden kann. Die `Serializer`-Instanz ermöglicht es, die XML-Daten eines Dokumentes in eine `String`-Instanz zu extrahieren. Mit dieser `String`-Instanz wird eine Instanz der Klasse `StringReader` erzeugt. Diese `StringReader`-Instanz wird anschließend der Methode `validate` einer neu erzeugten `Validator`-Instanz übergeben. Ab diesem Punkt wird die Konsistenzprüfung von der `Validator`-Klasse übernommen und wie in Abschnitt 10.3.2 erläutert, durchgeführt. Das Ergebnis der Konsistenzprüfung wird in einer Variable vom Typ `boolean` abgelegt. Tritt in diesem Bereich des Triggers ein Fehler auf, wird automatisch die Ergebnisvariable auf `false` gesetzt.

Wenn die Ergebnisvarible auf `true` steht, wird die in der Vorbereitungsphase erstellte Kopie des Dokumentes mit der Methode `removeXMLResource` der Klasse `Collection` aus dem Datenbanksystem entfernt. Wenn die Konsistenzprüfung als Ergebnis `false` liefert, wird die Kopie, die in der Vorbereitungsphase erstellt wurde, mit der Methode `moveXMLResource`, der Klasse `DBBroker`, zurück an die ursprüngliche Position im Datenbanksystem verschoben. Tritt in diesem zweiten Teil, in dem abhängig vom Ergebnis der Konsistenzprüfung weiter vorgegangen wird, ein Fehler auf, wird dieser ausgegeben, da die Weitergabe der Fehlermeldung an den Benutzer noch nicht implementiert ist.

10.4.3 Aufgetretene Probleme

Zu Beginn war der Plan, dass die modifizierten Dokumente in der Abschlussphase des Triggers aus dem Datenbanksystem gelesen werden. Dieser Plan scheiterte daran, dass

beispielsweise bei einer Validierung nach einem XQuery, das veränderte Dokument noch mit einer Schreibsperre (WRITE-LOCK) versehen ist. Dadurch führte der Versuch dieses Dokument zu lesen zu einem Verklemmung (DeadLock), da der Trigger darauf wartete, dass die Schreibsperre (WRITE-LOCK) gelöst wird, die Schreibsperre (WRITE-LOCK) erst nach Beendigung des Triggers gelöst wird.

Der nächste Versuch war, die Daten des Dokumentes aus der `DocumentImpl`-Instanz zu extrahieren. Dazu wurde die `Node`-Instanz, welche die Dokumentendaten enthält, aus der `DocumentImpl`-Instanz extrahiert. Anschließend wurden die Daten mit der `toString` Methode umgewandelt. Das Ergebnis dieser `toString`-Methode war syntaktisch leider nicht mit dem Ausgangsdokument identisch. Letztlich war die Verwendung der `Serializer`-Klasse zur Umwandlung der `DocumentImpl`-Instanz ein Erfolg.

10.5 Gegenüberstellung der Validierungsmechanismen

Die Tabelle 10.1 zeigt eine Aufstellung der Validierungsmechanismen. Dabei wurde untersucht, wann eine Validierung/Konsistenzprüfung durch den Validierungsmechanismus durchgeführt werden kann. Im Weiteren wird davon ausgegangen, dass das Datenbanksystem so konfiguriert ist, dass die Validierungsmechanismen in jedem ihnen möglichen Fall ausgeführt werden. Es existieren drei Validierungsmechanismen, die implizite Validierung, die explizite Validierung und der neu entwickelte `ValidationTrigger`. Eine Validierung oder Konsistenzprüfung kann stattfinden, wenn ein Dokument in das Datenbanksystem eingefügt wird (Insert), wenn ein Dokument in dem Datenbanksystem mit einem XQuery-Update-Extension Ausdruck (XQuery) oder einem XUpdate Ausdruck (XUpdate) verändert wird, oder wenn ein im Datenbanksystem vorhandenes Dokument geprüft werden soll (Benutzer).

	Insert	XQuery	XUpdate	Benutzer
Implizite Validierung	Ja	Nein	Nein	Nein
Explizite Validierung	Nein	Nein	Nein	Ja
ValidationTrigger	Ja	Ja	Ja	Nein

Tabelle 10.1: Vergleich der Validierungsmechanismen

In Abschnitt 9.4 wurde bereits erklärt, dass der `ValidationTrigger` nur deshalb entwickelt wurde, da das Datenbanksystem nur einen Teil der Fälle, in denen validiert werden muss, abdeckt. Abschließend wurde dort der Funktionsumfang des `ValidationTrigger` mit der expliziten Validierung verglichen. Dabei wurde festgestellt, dass die implizite Validierung vollständig durch den `ValidationTrigger` ersetzt werden kann. Dies liegt daran, dass die implizite Validierung nur bei dem Trigger-Ereignis „insert“ und teilweise bei dem Ereignis „update“ ausgeführt wird. Beim

Ereignis „update“ wird die implizite Validierung nur ausgeführt, wenn ein Dokument in das Datenbanksystem eingefügt wird und dabei ein Dokument im Datenbanksystem überschreiben soll. Die implizite Validierung hat den Vorteil, dass die Dokumente vor dem Einfügen in das Datenbanksystem auf ihre Validität/Konsistenz geprüft werden. Wenn die implizite Validierung durch den `ValidationTrigger` ersetzt wird, wird die Konsistenzprüfung in der Endphase ausgeführt. Dies hat zur Folge, dass jedes Dokument zuerst in das Datenbanksystem eingefügt wird und erst anschließend geprüft wird. Daraus ergibt sich der Nachteil, dass ein inkonsistentes Dokument zuerst in das Datenbanksystem eingefügt wird und nach der Prüfung wieder entfernt werden muss. Aus diesem Grund wird die implizite Validierung weiterhin verwendet und der `ValidationTrigger` vorerst nur bei dem Ereignis „update“ verwendet.

Die explizite Validierung ist sowohl von der impliziten Validierung als auch vom `ValidationTrigger` unabhängig. Dies liegt daran, dass die explizite Validierung, wie der Name schon sagt explizit aufgerufen werden muss und nicht durch ein bestimmtes Ereignis im Datenbanksystem automatisch ausgelöst wird.

10.6 Zusammenfassung

Zunächst wurde beschrieben, welche Änderungen am Datenbanksystem vorgenommen wurden, damit die in Abschnitt 9.3 entworfene Erweiterung der Konfigurierbarkeit umgesetzt werden konnte. In nächsten Schritt wurde die Implementierung des `owlconsistency`-Moduls, nach dem in Abschnitt 7.4 vorgestellten Entwurfes, erläutert. Dabei wurde, wie bereits angemerkt, nur der `Basisentwurf` implementiert, der erweiterte Entwurf wurde jedoch beim Anlegen der Klassenstruktur des Moduls bereits berücksichtigt, damit eine spätere Implementierung möglichst einfach ist. Anschließend wurde die Integration des `owlconsistency`-Moduls in die vorhandene Datenbankarchitektur beschrieben. Danach wurde die Implementierung des `ValidationTriggers` dargestellt, der die fehlende Konsistenzprüfung bei Änderungen durch XQuery und XUpdate Anfragen ergänzt. Abschließend wurde der Funktionsumfang der impliziten Validierung, expliziten Validierung und des `ValidationTriggers` gegenübergestellt, um zu untersuchen, wie sich deren Funktionsumfang überschneidet.

11 Evaluation der Umsetzung

Im ersten Teil der vorliegenden Arbeit bestand die Aufgabe darin, ein geeignetes natives XML-Datenbanksystem (Open Source) zu finden, welches die Möglichkeit bietet, XML-Dokumente zu validieren und anschließend die Architektur dieses Datenbanksystems daraufhin zu untersuchen, ob sie eine möglichst gekapselte Validierungseinheit besitzt. Zu diesem Zweck wurden vier native XML-Datenbanksysteme ausgewählt, Xindice von Apache, eXist, XTC der Technischen Universität Kaiserslautern und BaseX der Universität Konstanz. Bei genauerer Betrachtung (siehe Abschnitt 5) stellte sich heraus, dass von diesen XML-Datenbanksystemen einzig eXist für die Lösung im Rahmen der vorliegenden Arbeit verwendbar ist, da die übrigen Datenbanksysteme entweder keine Validierung von XML-Dokumenten anbieten, oder sich in einem zu frühen Entwicklungsstadium befinden. Das eXist Datenbanksystem unterstützt die Validierung von XML-Dokumenten sowohl implizit, also automatisch beim Einfügen eines neuen Dokumentes, als auch explizit, wenn vom Benutzer oder in einem XQuery ein bestimmtes, schon im Datenbanksystem vorhandenes, Dokument validiert wird. Bei der genaueren Betrachtung des Datenbanksystems stellte sich heraus, dass eXist in den Konfigurationsdateien bereits einen Schalter enthält, mit dem die Ausführung einer impliziten Validierung von Dokumenten beeinflusst werden kann.

Bei der anschließenden Analyse der Systemarchitektur zeigte sich, dass die Ausführung der impliziten/expliziten Validierung in unterschiedlichen Klassen stattfindet. Explizite Validierungen werden in der Klasse `Validator` und implizite Validierungen in der Klasse `Collection` ausgelöst. Dies Trennung ist darauf zurückzuführen, dass die explizite Validierung immer, wenn sie ausgeführt wird, eine Validierung durchführen soll. Die implizite Validierung führt nur eine Validierung durch, wenn der Schalter in der Konfiguration entsprechend gesetzt ist. Für die eigentliche Validierung wird der Apache Xerces 2 XML-Parser verwendet. Neben den Vorteilen einer impliziten und expliziten Validierung besitzt eXist aber den Nachteil, dass bei einer Änderungsoperation durch eine der Anfragesprachen (XUpdate oder XQuery-Update-Extension) keine Validierung durchgeführt wird, sondern einfach die Änderungen im Datenbanksystem übernommen werden.

Im zweiten Teil der Arbeit sollte das ausgewählte native XML-Datenbanksystem so erweitert werden, dass OWL-DL-Ontologien damit auf ihre Gültigkeit (Konsistenz) geprüft werden können. Zu diesem Zweck wurden die OWL-Reasoner RacerPro,

FaCT++ und Pellet, auf ihre Tauglichkeit für die Konsistenzprüfung von OWL-Dokumenten, betrachtet. Die Entscheidung zwischen Pellet und RacerPro fiel am Ende auf RacerPro, da die Universität Erlangen-Nürnberg Kontakte zu den Entwicklern hat, wodurch ein besonders guter Support erwartet wurde. Anschließend musste eXist mit RacerPro verbunden werden, so dass OWL-Dokumente vom Datenbanksystem aus auf ihre Konsistenz geprüft werden können. Der Schalter in der Konfiguration von eXist wurde dabei so erweitert, dass die implizite Validierung auch so eingestellt werden kann, dass OWL-Dokumente mit Hilfe des OWL-Reasoners auf ihre Konsistenz untersucht werden können. Für die Kommunikation zwischen eXist und RacerPro wurde das Modul `owlconsistency` entwickelt, welches in eXist integriert wurde. Dieses Modul verwendet das Modul `jracer` um eine TCP-Verbindung zur nRQL-Schnittstelle von RacerPro aufzubauen, über die Befehle an RacerPro übermittelt werden können. Der Nachteil an dieser Verbindung ist die Einschränkung, dass die Dokumente nicht über die Netzwerkverbindung übertragen werden können, sondern als Datei auf der Festplatte abgelegt werden müssen, um anschließend von RacerPro eingelesen zu werden. Diese zusätzlichen Zugriffe auf die Festplatte, die bei jeder Konsistenzprüfung nötig sind, stellten in Hinsicht auf die Performanz ein Problem dar.

Bisher wurde nach einer Änderung immer das gesamte Dokument auf seine Konsistenz geprüft. Dabei stellt sich die Frage, ob es nötig ist, immer das gesamte Dokument auf seine Konsistenz zu prüfen, oder ob es möglich ist, nur den neuen Teil zu prüfen, da für das alte Dokument bekannt ist, dass es konsistent ist. Bei XML-Dokumenten, die schemavalidiert werden, ist es schließlich ausreichend, wenn die neuen/veränderten Teile gegen die Schemadatei geprüft werden. Dieser Ansatzpunkt wird auch in Abschnitt 13 angesprochen.

Das Modul `owlconsistency` wurde so in das Datenbanksystem eingefügt, dass es eine Konsistenzprüfung genau dann ausführt, wenn der Schalter in der Konfiguration auf dem OWL-Modus steht. Obwohl nun eine implizite als auch explizite Validierung (Konsistenzprüfung) von OWL-Dokumenten mit Hilfe des Datenbanksystems möglich ist, stellt sich immer noch das Problem, dass Änderungen an Dokumenten durch XUpdate Ausdrücke oder XQuery-Update-Extension Ausdrücke ohne Überprüfung übernommen werden. Zu diesem Zweck wurde der `ValidationTrigger` entwickelt, der vor jeder Änderung eine Kopie des Dokumentes im Datenbanksystem erstellt und nach der Änderung des Dokumentes dessen Konsistenz prüft und im inkonsistenten Fall das modifizierte Dokument wieder durch die vorher gesicherte Fassung ersetzt. Um sicherzustellen, dass der `ValidationTrigger` in jeder Situation das korrekte Ergebnis liefert wurden alle Fälle getestet. Die Beschreibung dieser Testfälle ist im Anhang im Abschnitt A.2 zu finden.

Abschließend wird betrachtet, ob die Ziele, die in Abschnitt 1.2 für die Erweiterung des Datenbanksystems gesteckt wurden, erreicht werden konnten. Das erste Ziel war die

Konsistenzprüfung von Dokumenten beim Einfügen in das Datenbanksystem. Dieses Ziel wurde durch die Entwicklung des `owlconsistency`-Moduls erfüllt. Um Dokumente, die mit einem XQuery-Update-Extension Ausdruck oder einem XUpdate Ausdruck verändert wurden, auf ihre Konsistenz zu prüfen reicht die vorhandene Funktionalität des Datenbanksystems nicht aus. Daher wurde der `ValidationTrigger` entwickelt. In Abschnitt 9.4 wurde sein Funktionsumfang untersucht, was zeigte, dass mit ihm sogar die implizite Validierung ersetzt werden könnte. Aufgrund der des dort beschriebenen Nachteils wird dies jedoch nicht gemacht. Bei der Entwicklung des `owlconsistency`-Moduls und seiner Integration in das Datenbanksystem wurde stets darauf geachtet, dass die Konsistenzprüfung sowohl über die REST- und XML-RPC-Schnittstelle als auch über die XML:DB API möglich ist. Dadurch kann das Datenbanksystem sowohl zur zentralen Datenhaltung verwendet werden, als auch in andere Anwendungen integriert werden. Neben der REST- und XML-RPC-Schnittstelle besitzt eXist wie Abbildung 6.1 zeigt noch weitere Schnittstellen, nämlich: WebDAV, AtomServices und SOAP. Es ist nicht geprüft, ob die Konsistenzprüfung beim Einfügen über diese Schnittstellen funktioniert. Die explizite Validierung oder die Validierung mit Hilfe des `ValidationTriggers` ist sichergestellt, da diese beiden Mechanismen schnittstellenunabhängig sind. Die letzte Anforderung an das erweiterte Datenbanksystem war, dass es weiterhin als natives XML-Datenbanksystem eingesetzt werden kann, welches eine syntaktische Validierung von XML-Dokumenten durchführen kann. Dies ist dadurch gewährleistet, dass das Datenbanksystem in der Konfigurationsdatei einen Schalter enthält, mit dem zwischen XML- und OWL-Betrieb umgeschaltet werden kann.

12 Zusammenfassung der Arbeit

Ontologien verbreiten sich immer stärker zur Speicherung von Wissen. Zur Repräsentation dieser Ontologien wird häufig OWL-DL verwendet. Die Konsistenzprüfung dieser OWL-DL-Ontologien übernehmen so genannte OWL-Reasoner. Im Rahmen der vorliegenden Arbeit wurde die Kombination von Konsistenzprüfung der Dokumente mit der Speicherung der Dokumente in einem XML-Datenbanksystem erläutert. Als Resultat soll ein Datenbanksystem, das beim Einfügen automatisch OWL-Dokumente auf ihre Konsistenz prüft und dadurch sicherstellt, dass sie nur konsistente Dokumente enthält, entstehen.

Um das gewünschte Resultat zu erhalten wurde zunächst analysiert, wie die Speicherung und Konsistenzprüfung von OWL-Dokumenten bisher abläuft und schließlich ein Modell entwickelt, wie es aus Endnutzersicht nach der Erweiterung funktionieren soll. Im nächsten Schritt wurde untersucht, in welcher Form OWL-DL-Ontologien im Datenbanksystem gespeichert werden sollen. Abschließend wurden die Anforderungen an die Systembausteine, das native XML-Datenbanksystem und der OWL-Reasoner, die verwendet werden sollen, gestellt.

Nachdem die Anforderungen an die Systembausteine festgelegt waren, wurden vier XML-Datenbanksysteme sowie drei OWL-Reasoner ausgewählt. Die gewählten Komponenten wurden daraufhin untersucht, ob die ausgearbeiteten Anforderungen von den einzelnen Systembausteinen erfüllt werden. Anhand der Untersuchungsergebnisse fiel die Wahl des XML-Datenbanksystems auf eXist und auf den OWL-Reasoner RacerPro.

Nach der Festlegung der zu verwendenden Komponenten wurde mit der Planung der Grobarchitektur begonnen. Zuerst wurde festgelegt, dass das Datenbanksystem eine Möglichkeit besitzen muss, um zu unterscheiden, ob ihr ein normales XML-Dokument, das schemavalidiert werden soll, oder ein OWL-Dokument, das auf seine Konsistenz geprüft werden soll, übergeben wird. Daraufhin wurde untersucht in welchen Fällen eine Konsistenzprüfung vom Datenbanksystem vorgenommen wird und wie diese abläuft. Dabei stellte sich heraus, dass eXist bei Änderungen durch XQuery-Update-Extension Ausdrücke und XUpdate Ausdrücke keine Konsistenzprüfung durchführt. Anschließend wurde untersucht, welche Möglichkeiten für die Anbindung des OWL-Reasoners an das XML-Datenbanksystem existieren. Nach der Abwägung aller Vor- und Nachteile fiel die Wahl auf die nRQL-Schnittstelle von RacerPro.

Im nächsten Schritt wurde die Architektur des Datenbanksystems untersucht. Durch die Untersuchung wurde ermittelt, wo die Validierung durchgeführt wird. Damit wurden mögliche Ansatzpunkte für die Erweiterung des Datenbanksystems eruiert. Die Untersuchung ergab, dass die Validierung in einen impliziten und einen expliziten Teil aufgeteilt ist. Anschließend wurde untersucht, welche Konfigurationsmöglichkeiten es für die implizite Validierung über Konfigurationsdateien gibt und wie das Auslesen eines Dokumentes aus dem Datenbanksystem abläuft. Zuletzt wurden die von dem Datenbanksystem angebotenen Triggermechanismen untersucht, da ein Trigger entwickelt werden sollte, der Lücken, die in Abschnitt 6.3 erläutert werden, in der Funktionalität des Datenbanksystems schließt.

Anschließend wurden die Ergebnisse aus der Systemanalyse mit dem Grobentwurf kombiniert. Dies führte zum Entwurf für das `owlconsistency`-Modul, welches die Anbindung an den OWL-Reasoner und die Konsistenzprüfung enthält. Daraufhin wurde erarbeitet, welche Änderungen am Datenbanksystem vorgenommen werden müssen, um das `owlconsistency`-Modul einzubinden. Auf diesen Entwurf folgte der Plan für die Erweiterung der Konfigurierbarkeit, damit ein eigener Validierungsmodus für OWL-Dokumente eingeführt wird. Abschließend wurde ein Trigger entworfen, der die von eXist nicht angebotene Validierung im Fall einer Veränderung durch eine XQuery-Update-Extension/XUpdate Anfrage liefert.

Im nächsten Schritt wurden die erstellten Entwürfe umgesetzt. Dabei wurde beschrieben, wie die Umsetzung der Entwürfe erfolgte. Außerdem wurden Probleme, die bei der Umsetzung auftraten, dargestellt.

Die einzelnen Schritte wurden abschließend evaluiert, um festzustellen, ob alle geforderten Aufgaben umgesetzt wurden und um festzustellen, ob die realisierte Lösung noch Verbesserungspotential besitzt.

Für die Endnutzer bietet das Datenbanksystem eine Reihe von Vorteilen gegenüber der lokalen Speicherung. Durch die Speicherung im Datenbanksystem wird ein Dokument, wenn es eingefügt wird, automatisch auf seine Konsistenz geprüft. Dadurch ist sichergestellt, dass jedes Dokument, das in dem Datenbanksystem gespeichert ist, konsistent ist. Wenn das Datenbanksystem im Server-Modus gestartet wird, dann kann es zur zentralen Speicherung der OWL-Dokumente verwendet werden. Die zentrale Speicherung in einem Datenbanksystem hat den zusätzlichen Vorteil, dass über Sperrverfahren (Locking) sichergestellt wird, dass immer nur ein Benutzer Änderungen an einem Dokument vornehmen kann. Wenn das Datenbanksystem nicht im Server-Modus betrieben werden soll, ist es alternativ möglich, es in eine Anwendung einzubetten. Das Datenbanksystem kann so beispielsweise in einen OWL-Editor integriert werden, der ein Dokument nicht im lokalen Dateisystem speichert, sondern in das Datenbanksystem einfügt.

13 Ausblick

Auf den ersten Blick erfüllt die im Rahmen der vorliegenden Arbeit entwickelte Erweiterung des eXist Datenbanksystems alle in der Aufgabenstellung geforderten Punkte. Trotzdem besteht in allen Teilen noch weiteres Verbesserungspotential.

Ein großer Fortschritt wäre, wenn das eXist Datenbanksystem in einer zukünftigen Version, von den Entwicklern so erweitert würde, dass es selbstständig eine Validierung durchführt, sobald eine Veränderung an einem oder mehreren Dokumenten im Datenbanksystem, durch eine Veränderungsoperation (XQuery-Update-Extension oder XUpdate) ausgelöst wird. Durch diese Verbesserung des Datenbanksystems wäre es nicht länger nötig den Umweg über den `ValidationTrigger` zu machen, um die Validierung durchzuführen. Das positive Resultat wäre ein Performanzgewinn, da es nicht mehr nötig wäre, vor jeder Veränderungsoperation eine Kopie des zu verändernden Dokumentes zu erstellen.

Die Leistung des Datenbanksystems bei der Konsistenzprüfung könnte deutlich gesteigert werden, indem ein anderes Kommunikationsprotokoll zum OWL-Reasoner, im owlconsistency-Modul verwendet wird. Eine alternative Schnittstelle wird mit DIG (siehe Abschnitt 3.8) vorgestellt. In der aktuellen Version von RacerPro wird aber, wie bereits beschrieben, lediglich die Version 1.1 der DIG-Schnittstelle unterstützt. Sofern die Entwickler von RacerPro zu einem zukünftigen Zeitpunkt die Spezifikation der neuen DIG-Schnittstelle (DIG 2.0) in RacerPro umsetzen, würde es sich anbieten, das Modul owlconsistency auf DIG 2.0 umzustellen, da diese neue Version spezifiziert wurde, um die Unzulänglichkeiten der Vorgängerversion zu beheben. Durch diese Umstellung würde das Datenbanksystem einerseits auf lange Sicht unabhängig von RacerPro, da die nRQL-Schnittstelle nur von RacerPro angeboten wird. Andererseits könnten die Dokumente aus dem Datenbanksystem direkt über das Netzwerk an den OWL-Reasoner übergeben werden. Für die Übertragung mit DIG 2.0 wäre es immer noch notwendig, die Dokumente in das DIG-Schema zu serialisieren, was aber erwartungsgemäß einfacher und schneller ist, als das Dokument bei jedem Konsistenztest in eine temporäre Datei auf die Festplatte zu serialisieren und zu persistieren.

Neben diesen beiden Punkten, die weit reichende Änderungen in den im Rahmen dieser Arbeit verwendeten Softwarebausteinen voraussetzen, kann als erster einfacher Schritt der, in Kapitel 7.4.2 vorgestellte, erweiterte Systementwurf umgesetzt werden. Der erweiterte Entwurf bringt den Vorteil mit sich, dass die T-Box und A-Box separiert

werden können. Durch die Aufteilung kann erreicht werden, dass die Bearbeiter nur auf die Teile der Ontologie Zugriff bekommen, der für sie relevant ist. Die Boxen (T-Box/A-Box) an sich können wiederum auf mehrere Dokumente im Datenbanksystem verteilt werden. Dadurch kann einerseits eine redundante Speicherung der T-Box verhindert werden, andererseits kann durch die Aufspaltung einer T-Box das Laden von nicht benötigten Teilen verhindert werden, was auf lange Sicht einen Geschwindigkeitsvorteil bringen kann.

In Abschnitt 11 wurde die Frage gestellt, ob es eine Möglichkeit gibt, bei der Änderung eines Dokumentes nur den Teil erneut auf seine Konsistenz zu prüfen, der von der Änderung betroffen ist. Die Umsetzung dieses Ansatzes hat großes Potential, da dadurch nur ein Teil des Dokumentes an den OWL-Reasoner übertragen werden muss.

Anhang A

Dokumentation

A.1 eXist Validierung und Anfrageausführung

Die manuelle Validierung ist möglich, indem der mitgelieferte Client gestartet wird, um auf das Datenbanksystem zuzugreifen. In der Kommandozeile des Clients gibt es das Kommando „validate“. Diesem Befehl können wahlweise ein oder zwei Argumente übergeben werden. Das erste Argument ist der absolute Pfad, im Datenbanksystem, des Dokumentes das validiert werden soll.

```
1 validate /db/beispiel.xml
```

Listing A.1: Explizite Validierung ohne Grammatik

Als zweites Argument kann noch das zur Datei gehörige XML-Schema oder die Document Type Definition (DTD) angegeben werden.

```
1 validate /db/beispiel.xml /db/schema/beispiel.xsd
2 validate /db/beispiel.xml /db/schema/beispiel.dtd
```

Listing A.2: Explizite Validierung mit Grammatik

XPath Ausdrücke können mit dem Kommando „find“ direkt auf der Kommandozeile des Clients eingegeben werden. Für die Ausführung von XQuery Anfragen kann auch der Client verwendet werden, allerdings gibt es dafür ein spezielles Fenster, (Zu erreichen über das Menü „Edit“, Menüpunkt „Find“) in dem die Anfragen wahlweise eingetippt werden können oder vorgefertigte Anfragen aus einer Datei geladen werden können. Es ist selbstverständlich nicht nötig für eine einfache Anfrage extra den graphischen Client (GUI) zu starten, sondern es kann die vorgefertigte Anfrage auch direkt über die Kommandozeile des Betriebsystems ausgeführt werden.

```
1 bin/client.sh -F xquery.xq
```

Listing A.3: Ausführung einer XQuery Anfrage über die Kommandozeile

Zusätzlich gibt es die Variante den Client nur im XQuery Anfragemodus zu starten und ohne graphische Oberfläche Anfragen in eine Kommandozeile einzugeben.

```
1 bin/client.sh −x
```

Listing A.4: Java Client im Textmodus für XQuery Anfragen starten

Weiterhin wird für die Veränderung von XML-Dokumenten XUpdate zur Verfügung gestellt. Dabei ist zu beachten, dass XUpdates nur über die Kommandozeile des Betriebsystems ausgeführt werden können und nicht über die graphische Benutzeroberfläche (GUI).

```
1 bin/client.sh −c /db −f example.xml −X xupdate.xml
```

Listing A.5: Ausführung eines XUpdates von der Kommandozeile aus

Die Optionen -c und -f sind optional und stellen lediglich eine Einschränkung der Collection/Datei dar, auf die das Update bezogen ist.

A.2 Testbeschreibung

In diesem Abschnitt wird erklärt, wie geprüft wurde, ob die Konsistenzprüfung im Datenbanksystem, wenn sie ausgeführt wird, das korrekte Ergebnis liefert. Bei der Konsistenzprüfung existiert ein positives Ergebnis und zwei negative. Das positive Ergebnis bedeutet, dass das untersuchte Dokument konsistent ist. Sofern das Dokument nicht konsistent ist, wird zwischen einem semantischen und einem syntaktischen Problem unterschieden. Ein syntaktisches Problem tritt auf, wenn das untersuchte Dokument kein wohlgeformtes XML enthält. Das semantische Problem bedeutet, dass im Dokument Widersprüche enthalten sind. Nach der Erweiterung des Datenbanksystems existieren drei Mechanismen, die eine Konsistenzprüfung durchführen. Diese Mechanismen sind die implizite Validierung, die explizite Validierung und der `ValidationTrigger` und werden im Anschluss untersucht.

A.2.1 Implizite Validierung

Die Konsistenzprüfung bei der impliziten Validierung wird beim Einfügen eines Dokumentes in der Klasse `Collection` veranlasst. Von dem Datenbanksystem wird garantiert, dass beim Einfügen über die REST- oder XML-RPC-Schnittstelle eine Konsistenzprüfung durchgeführt wird. Dasselbe gilt für die XML:DB API. Für jede dieser Schnittstellen wurde das Einfügen eines konsistenten Dokumentes (cidoc.owl), eines nicht wohlgeformten Dokumentes (cidoc_nwf.owl) und eines Dokumentes, das

semantisch inkonsistent ist (pizza.owl) getestet. Das Einfügen über die XML-RPC-Schnittstelle und die XML:DB API wurde über den mitgelieferten Client[1] getestet. Für das Einfügen über die REST-Schnittstelle wurde das mitgelieferte Skript `put.py`[2] verwendet. Beim Arbeiten mit dem Skript stellte sich heraus, dass auch ein Fehler angezeigt wird, wenn der HTTP Antwortcode „201“ (Created) als Antwort vom Datenbanksystem gesendet wird.

Das Einfügen lieferte über alle Schnittstellen dasselbe Ergebnis. Der Versuch das Dokument `cidoc.owl` in das Datenbanksystem einzufügen war erfolgreich. Die anderen beiden Dokumente wurden vom Datenbanksystem abgelehnt.

A.2.2 ValidationTrigger

Mit dem `ValidationTrigger` ist eine Konsistenzprüfung möglich, wenn ein Dokument in das Datenbanksystem eingefügt werden soll, wenn ein Dokument in dem Datenbanksystem mit einem XQuery-Update-Extension Ausdruck, oder einem XUpdate Ausdruck geändert werden soll. Für jeden dieser Fälle wird getestet, ob das Datenbanksystem richtig reagiert, wenn ein konsistentes Dokument, ein Dokument mit einem syntaktischen Fehler oder einem semantischen Fehler geprüft wird.

Insert

Der Test der Konsistenzprüfung beim Einfügen eines Dokumentes wurde analog zur impliziten Validierung durchgeführt. Der Unterschied liegt darin, dass die Konsistenzprüfung des `ValidationTriggers` schnittstellenunabhängig ist. Deshalb wurde das Einfügen nur über die XML-RPC-Schnittstelle getestet. Das Ergebnis des Tests ist identisch zu dem Ergebnis, welches die implizite Validierung liefert.

XUpdate Ausdruck

Um zu testen, ob die Konsistenzprüfung die korrekten Ergebnisse liefert, wurden erneut die Dokumente `cidoc.owl` und `pizza.owl` verwendet. Das Dokument `cidoc.owl` wurde herangezogen, um eine Änderung, bei der das Dokument weiterhin konsistent ist und eine Änderung, bei der das Dokument danach einen syntaktischen Fehler enthält, zu testen. An dem Dokument `pizza.owl`, welches semantische Fehler enthält, soll eine Änderung vorgenommen werden, welche wenn der `ValidationTrigger`

[1] Zu finden in: $EXIST_HOME/bin
[2] Zu finden unter: $EXIST_HOME/samples/http

korrekt funktioniert, verworfen wird. In diesem Fall werden alle Änderungen durch die Anwendung eines XUpdate Ausdrucks vorgenommen.

Die konsistente Änderung am Dokument `cidoc.owl` wurde als solche erkannt und übernommen. Bei der Änderung des Dokumentes `pizza.owl` wurde bei der Konsistenzprüfung erkannt, dass das Dokument einen sementischen Fehler enthält. Der Versuch das Dokument `cidoc.owl` so zu verändern, dass es nicht mehr wohlgeformt ist, schlug fehlt, da das Datenbanksystem dies bereits bei der Übergabe des XUpdate Ausdrucks erkannte.

XQuery-Update-Extension Ausdruck

Dieser Fall wurde analog zu dem letzten Fall ausgeführt. Statt XUpdate Ausdrücken zur Änderung wurde in diesem Fall XQuery-Update-Extension Ausdrücke verwendet. Dieser Testfall lieferte identische Ergebnisse wie der letzte Fall.

A.2.3 Explizite Validierung

Die Konsistenzprüfung bei der expliziten Validierung wird in der Klasse `Validator` veranlasst. Dabei wird eine Konsistenzprüfung von Dokumenten, die bereits im Datenbanksystem enthalten sind durchgeführt. Da bereits beim Einfügen eines Dokumentes darauf geachtet wird, dass jedes Dokument wohlgeformtes XML enthält, kann der Fall, dass ein Dokument nicht wohlgeformt ist, nicht auftreten. Es gibt zwei Möglichkeiten eine explizite Validierung zu veranlassen.

Die erste Möglichkeit ist eine XQuery Anfrage für ein Dokument zu stellen. Als Referenzdokument wird das Dokument `cidoc.owl` und das Dokument `pizza.owl` herangezogen. Die Konsistenzprüfung des Dokumentes `cidoc.owl` ergibt, dass es ein konsistentes Dokument handelt, während die Prüfung des Dokumentes `pizza.owl` ergibt, dass das Dokument Widersprüche enthält.

Über die XML-RPC-Schnittstelle und die XML:DB API ist es ebenfalls möglich auf die Funktionalität der Klasse `Validator` zuzugreifen. Die Konsistenzprüfung wurde dabei auf die schon vorher verwendeten Dokumente `cidoc.owl` und `pizza.owl` angewendet. Um beide Schnittstellen zu testen, wurde wiederum der mitgelieferte Client, der bereits bei der impliziten Validierung erwähnt wurde, verwendet. Mit diesem Client kann sowohl die XML-RPC-Schnittstelle, als auch die XML:DB API angesprochen werden. Die Prüfung der Dokumente ergab für beide Schnittstellen, dass `cidoc.owl` konsistent ist und `pizza.owl` nicht.

A.2.4 Ergebnis

Die Referenzdokumente `cidoc.owl`, `cidoc_nwf.owl` und `pizza.owl` wurden vorher mit RacerPro untersucht, damit Vergleichswerte vorliegen, mit denen die Ergebnisse des Tests des Datenbanksystems verglichen werden können. Die Ergebnisse, die im Rahmen dieser Testfälle vorliegen stimmen mit den Ergebnissen, die der Test der einzelnen Dokumente mit RacerPro direkt ergab, exakt überein.

Literaturverzeichnis

[BCF+07] Boag, Scott ; Chamberlin, Don ; Fernández, Mary F. ; Florescu, Daniela ; Robie, Jonathan ; Siméon, Jérôme: *XQuery 1.0: An XML Query Language*. http://www.w3.org/TR/xquery. Version: 01 2007

[BCM+07a] Baader, Franz ; Calvanese, Diego ; McGuinness, Deborah L. ; Nardi, Daniele ; Patel-Schneider, Peter F.: *The Description Logic Handbook*. Cambridge University Press, 2007. – 51–55, 461–479 S.

[BCM+07b] Baader, Franz ; Calvanese, Diego ; McGuinness, Deborah L. ; Nardi, Daniele ; Patel-Schneider, Peter F.: *The Description Logic Handbook*. Cambridge University Press, 2007. – 56–58, 72–73 S.

[Bec03] Bechhofer, Sean: *The DIG Description Logic Interface: DIG/1.1*. Februar 2003

[Bec06] Bechhofer, Sean: *DIG Interface*. http://dl.kr.org/dig/interface.html. Version: März 2006

[BPSM+08] Bray, Tim ; Paoli, Jean ; Sperberg-McQueen, C. M. ; Maler, Eve ; Yergeau, François: *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. XML Core Working Group. http://www.w3.org/TR/2008/REC-xml-20081126/. Version: November 2008

[CD99] Clark, James ; DeRose, Steve: *XML Path Language (XPath) Version 1.0*. http://www.w3.org/TR/1999/REC-xpath-19991116. Version: November 1999

[CFM+08] Chamberlin, Don ; Florescu, Daniela ; Melton, Jim ; Robie, Jonathan ; Siméon, Jérôme: *XQuery Update Facility 1.0*. http://www.w3.org/TR/2008/CR-xquery-update-10-20080801/. Version: August 2008

[Cla09] Clark&Parsia: *Pellet: The Open Source OWL DL Reasoner*. http://clarkparsia.com/pellet. Version: 2009

[eXi09] eXist: *Open Source Native XML Database*. http://exist.sourceforge.net/. Version: 2009

[Fie00] FIELDING, R. T.: *Architectural Styles and the Design of Network-based Software Architectures*, University of California, Diss., 2000

[Fou07] FOUNDATION, The Apache S.: *Apache Xindice - Frequently Asked Questions*. `http://xml.apache.org/xindice/1.1/faq.html`. Version: Dezember 2007

[Grü09] GRÜN, Christian: *BaseX Homepage*. `http://www.inf.uni-konstanz.de/dbis/basex/`. Version: 2009

[Gru93] GRUBER, Thomas R.: *A translation approach to portable ontology specifications*. 1993

[HKS06] HORROCKS, Ian ; KUTZ, Oliver ; SATTLER, Ulrike: *The Even More Irresistible SROIQ*. 2006

[Hor98] HORROCKS, Ian: The FaCT System. In: DE SWART, Harrie (Hrsg.): *Proc. of the 2nd Int. Conf. on Analytic Tableaux and Related Methods (TABLEAUX'98)* Bd. 1397, Springer, 1998 (Lecture Notes in Artificial Intelligence), S. 307–312

[Ini03] INITIATIVE, The X.: *XML-Database FAQ*. `http://xmldb-org.sourceforge.net/faqs.html#faq-1`. Version: 2003

[ISO86] *ISO8879:1986 - Information processing – Text and office systems – Standard Generalized Markup Language (SGML)*. 1986

[KST02] KAZAKOS, Wassilios ; SCHMIDT, Andreas ; TOMCZYK, Peter: *Datenbanken und XML*. Springer-Verlag, 2002. – 7–13 S.

[MMM04] MANOLA, Frank ; MILLER, Eric ; MCBRIDE, Brian: *RDF Primer*. `http://www.w3.org/TR/2004/REC-rdf-primer-20040210/`. Version: Februar 2004

[Rac05] RACER SYSTEMS GMBH & CO. KG (Hrsg.): *RacerPro User's Guide*. Version 1.9. Racer Systems GmbH & Co. KG, 12 2005. `http://www.racer-systems.com/products/racerpro/manual.phtml`

[SPG+07] SIRIN, Evren ; PARSIA, Bijan ; GRAU, Bernardo C. ; KALYANPUR, Aditya ; KATZ, Yarden: *Pellet: A Practical OWL-DL Reasoner*. 2007

[SWM04] SMITH, Michael K. ; WELTY, Chris ; MCGUINNESS, Deborah L.: *OWL Web Ontology Language Guide*. W3C Web Ontology Working Group. `http://www.w3.org/TR/2004/REC-owl-guide-20040210/`. Version: Februar 2004

[SXAML01] Staken, Kimbro ; XML:DB API Mailing List, Members of t.: *XML Database API Draft*. `http://xmldb-org.sourceforge.net/xapi/xapi-draft.html`. Version: September 2001

[TH06] Tsarkov, Dmitry ; Horrocks, Ian: *FaCT++ Description Logic Reasoner: System Description*. 2006

[Tsa07] Tsarkov, Dmitry: *FaCT++ Homepage*. `http://owl.man.ac.uk/factplusplus/`. Version: 05 2007

[Tsa09] Tsarkov, Dmitry: *FaCT++ Homepage*. `http://code.google.com/p/factplusplus`. Version: 2009

[Win03] Winer, Dave: *XML-RPC Specification*. `http://www.xmlrpc.com/spec`. Version: Juni 2003

[XTC08] XTC Project (Hrsg.): *ReferenceSimple - XTC Project Wiki*. XTC Project, September 2008