JOHANNES KEPLER
UNIVERSITÄT LINZ

Eingereicht von
Patrick Göbl, BSc

Angefertigt am
Institut für Wirtschafts-informatik - Data & Knowledge Engineering

Betreuer
o. Univ.-Prof. Dipl.-Ing. Dr. techn. Michael Schrefl

Mitbetreuung
Dr. Bernd Neumayr

February 2021

Prototypische Umsetzung und Evaluierung von Zugriffsmöglichkeiten auf GML-Daten für ein deduktives Datenbanksystem basierend auf F-Logic

Masterarbeit

zur Erlangung des akademischen Grades

Master of Science

im Masterstudium

Wirtschaftsinformatik

JOHANNES KEPLER
UNIVERSITÄT LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, 10.01.2021

Ort, Datum

Unterschrift

Kurzfassung

Die Geography Markup Language (GML) ist ein etablierter Standard für die Beschreibung und den Austausch von geografischen Informationen, der auf Basis von XML entwickelt wurde und über XML-Schemadefinitionen definiert ist. GML ist aber im Vergleich zu XML strukturierter, da es mit dem Object-Property-Modell ein spezifisches Datenmodell verwendet. GML wurde mit dem Hintergrund der Erweiterbarkeit entwickelt und bildet die Grundlage für die Entwicklung von domänenspezifischen Applikationsschemas. Diese werden in verschiedensten Bereichen für die Erstellung von Systemen verwendet, die geografische Daten verarbeiten (z.B. Luftfahrt, Wetterberichte, Städtemodelle). Ein Anwendungsszenario ist die Erstellung von deduktiven Datenbanksystemen für die intelligente Filterung von Flugverkehrsnachrichten. Dabei werden logische Programmiersprachen und Regelinterpreter für die Erstellung von Ableitungsregeln verwendet um neues Wissen aus bestehenden Daten abzuleiten. Damit diese Systeme GML-Daten verarbeiten können, müssen diese auf die GML-Daten zugreifen können. F-Logic und dessen Erweiterungen Flora-2 und ErgoAI bieten sich für diesen Zweck an. Der integrierte ErgoAI XML-Parser bietet zwar die Möglichkeit, beliebige XML-Dokumente in ErgoAI zu laden, berücksichtigt aber nicht die Besonderheiten von GML. Eine spezifische Transformation der GML-Daten anhand des Object-Property-Modells bietet jedoch Vorteile im Vergleich zur generischen Transformation von XML-Dokumenten. Es gibt bereits diverse Arbeiten, die sich mit der spezifischen Transformation von GML-Daten in unterschiedliche Datenformate beschäftigt. Darunter finden sich aber kaum Arbeiten, die sich mit F-Logic oder dessen Erweiterungen als Zielsystem beschäftigen. In dieser Arbeit wird deshalb eine prototypische Umsetzung und Evaluierung von Zugriffsmöglichkeiten auf GML-Daten für die Verwendung in deduktiven Datenbanksystemen auf Basis von F-Logic durchgeführt. Als durchgängiges Anwendungsbeispiel für die Evaluierung und Veranschaulichung der Zugriffsmöglichkeiten werden die beiden domänenspezifischen GML-Applikationsschemas AIXM und IWXXM verwendet, die zur Erstellung von Flugverkehrsnachrichten und Wetterberichten dienen.

Abstract

The Geography Markup Language (GML) is a well established standard for the description and interchange of geographical information. The GML standard is based on XML and defined by XML schema definitions, but GML is more structured than XML because it is based on a specific data model, the Object-Property Model. GML was developed with consideration for extensibility and provides the basis for the development of domain specific application schemes. These application schemes are utilised in a variety of areas to develop systems that process geographical data (e.g. aviation, weather reports, city models). One application scenario is the creation of deductive database systems for the intelligent filtering of air traffic messages. For this purpose logical programming languages and reasoning engines can be used for the definition of deduction rules to deduct new knowledge from existing data. In order for these systems to be able to process GML data it is necessary that they have access to the data. F-Logic and its extensions Flora-2 and ErgoAI are well suited logical programming languages for this task. Although the integrated ErgoAI XML parser offers the possibility to load arbitrary XML documents in ErgoAI, it does not consider the special characteristics of GML. However, the specific transformation of GML data according to the Object-Property Model provides benefits compared to the generic transformation of XML documents. There is already various research that deals with making GML data accessible in systems with different data formats, but hardly any research has been done that uses F-Logic or its extensions as the target system. This master thesis conducts the prototypical implementation and evaluation of different approaches for accessing GML data from deductive database systems based on F-Logic. The domain specific GML application schemes AIXM and IWXXM, which are used for the creation of air traffic messages and weather reports, serve as a coherent application scenario for the evaluation and exemplification of the implemented approaches.

Inhaltsverzeichnis

II Umsetzung und Evaluierung

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Kapitel 1

Einleitung

Die Geography Markup Language (GML) ist ein weit verbreiteter Standard zur Beschreibung und zum Austausch von geografischen Informationen, der seit dem Jahr 2000 vom OpenGeospatial Consortium entwickelt und gewartet wird. Der GML-Standard wurde auf Basis von XML entwickelt und stellt sowohl ein konzeptuelles Datenmodell, als auch ein Austauschformat für die Repräsentation von geografischen Daten dar. GML ist aber im Vergleich zu XML strukturierter, da GML mit dem Object-Property-Modell ein spezifisches Datenmodell verwendet. GML stellt unter anderem XML-Schemadefinitionen für die Beschreibung von geometrischen Objekten, Koordinatensystemen, Richtungsangaben oder zeitlichen Informationen zur Verfügung. GML wurde mit dem Hintergrund der Erweiterbarkeit entwickelt und soll als Basis für die Entwicklung von domänenspezifischen Applikationsschemas (z.B. Luftfahrt, Wetterinformationen, Stadtmodelle) dienen.

Das Aeronautical Information Exchange Model (AIXM) und das ICAO Meteorological Information Exchange Model (IWXXM) sind solche applikationsspezifische Ausprägungen von GML. Sie dienen zur Erstellung von digitalen Nachrichten im Flugverkehr (AIXM) und zur Erstellung von Berichten zu Wetterinformationen (IWXXM). Ein Anwendungsszenario für diese Datenmodelle ist die Filterung von Nachrichten, die die Flugsicherheit beeinflussen können. Dazu gehören beispielsweise Nachrichten zum aktuellen Status eines Flughafens (z.B. Sperre von Landebahnen) oder Sturmwarnungen.

Für die Umsetzung der Filterung dieser Nachrichten bietet sich die Verwendung von logischen Programmiersprachen wie F-Logic an. F-Logic ist eine deklarative Programmiersprache zur Wissensrepräsentation, die die logische Programmierung um Vorteile der objektorientierten Programmierung (z.B. Vererbung) erweitert. F-Logic ermöglicht durch die Definition von Ableitungsregeln und anderen Konstrukten die Generierung von implizitem Wissen aus bestehenden Daten. Dadurch können deduktive Datenbanksysteme erstellt werden, die mit Hilfe eines Regelinterpreters und einem Set von Filterregeln eine intelligente Filterung vornehmen können. Dafür ist es aber notwendig, dass diese Systeme auf die GML-Daten zugreifen können.

Bei diesem Datenzugriff entstehen jedoch diverse Probleme, die gelöst werden müssen. XML ist ein hierarchisch semi-strukturiertes Datenformat, bei dem die Zuordnung von Elementen bereits aus der Struktur der Dokumente hervorgeht. F-Logic verwendet ein objektorientiertes semi-strukturiertes Datenformat, bei dem die Anordnung der Elemente im Normalfall keine Rolle spielt. Die Herausforderung ist es, diese beiden Ansätze der Datenhaltung zu vereinen. Der in ErgoAI integrierte XML-Parser bietet zwar die Möglichkeit, beliebige XML-Dokumente in ErgoAI zu

laden, dieser berücksichtigt aber nicht die Besonderheiten von GML. Die besonderen Eigenschaften des Object-Property-Modells können aber genutzt werden, um eine auf GML-Daten zugeschnittene Transformation zu entwickeln. Dadurch kann eine kompakte Transformation umgesetzt werden, bei der trotzdem keine Informationen aus dem Ursprungsdokument verloren gehen. Diese spezifische Transformation bietet dann Vorteile (z.B. kürzere Navigationswege, Identifizierung von GML-Objekten) im Vergleich zu einer generischen Transformation von XML-Dokumenten.

Es gibt in der Literatur bereits diverse Arbeiten, die sich mit der Transformation von XML-Dokumenten in andere Datenformate beschäftigen. Diese Arbeiten zielen aber fast immer auf die Entwicklung von Transformationsalgorithmen in relationale Datenmodelle oder auf die Entwicklung von Abragesprachen für XML ab. Außerdem werden dabei die Besonderheiten des Object-Property-Modells nicht berücksichtigt. Es gibt auch bereits Arbeiten, die sich mit der Transformation von GML in Ontologiesprachen wie RDF oder OWL beschäftigen. Es gibt jedoch kaum Arbeiten, die sich speziell mit dem Zugriff auf GML-Daten aus F-Logic beschäftigen.

Das Ziel dieser Arbeit ist deshalb die prototypische Umsetzung und Evaluierung von Zugriffsmöglichkeiten auf GML-Daten für die Verwendung in deduktiven Datenbanksystemen auf Basis von F-Logic. Als Zielsprache bzw. Zielsystem wird in dieser Arbeit aber nicht F-Logic, sondern ErgoAI verwendet. ErgoAI ist eine kommerzielle Erweiterung von Flora-2, was wiederum eine Erweiterung von F-Logic ist. Der Vorteil von ErgoAI ist, dass es im Vergleich zu F-Logic und Flora-2 ein erweitertes Schnittstellenangebot bietet, das für den Zugriff auf GML-Daten von Vorteil ist. Bei den vorgestellten Ansätzen handelt es sich sowohl um Ansätze, die eine Transformation von GML-Daten in in ein anderes Datenformat bedingen, als auch um Ansätze bei denen Ad-Hoc Abfragen auf Basis der integrierten Schnittstellen von ErgoAI durchgeführt werden.

Der Aufbau dieser Arbeit gliedert sich in zwei Teile. Der erste Teil behandelt die technischen Grundlagen, die für diese Arbeit relevant sind. Dabei wird zuerst auf die wichtigsten Konzepte von XML und GML eingegangen und in weiterer Folge auf die domänenspezifischen GML-Applikationsschemas AIXM und IWXXM. Diese beiden Applikationsschemas dienen als durchgängiges Anwendungsbeispiel für diese Arbeit und werden für die Evaluierung der Zugriffsmöglichkeiten und für die Veranschaulichung des Datenzugriffs auf GML-Daten aus ErgoAI in den unterschiedlichen Zugriffsmöglichkeiten verwendet. Abschließend wird auf die wichtigsten Grundbegriffe und die für diese Arbeit relevanten Schnittstellen von ErgoAI eingegangen.

Der zweite Teil behandelt die prototypische Umsetzung und Evaluierung der Zugriffsmöglichkeiten. Es werden zunächst Anforderungen, Ziele und Nicht-Ziele für die Umsetzung der Zugriffsmöglichkeiten definiert. Anschließend wird ein Überblick über die Archtitektur und gemeinsamen Problemstellungen gegeben. Danach wird auf die konkrete Umsetzung der Zugriffsmöglichkeiten eingegangen. In einem anschließenden Kapitel erfolgt die Evaluierung und Gegenüberstellung der Ansätze bezüglich der Anforderungserfüllung und Zielerreichung. Abschließend wird das Fazit dieser Arbeit und ein Ausblick über zukünftige Forschungsmöglichkeiten, die auf dieser Arbeit aufbauen können, gegeben. Im Anhang befindet sich eine Installationsanleitung für die Softwarekomponenten, die für die Umsetzung dieser Arbeit notwendig sind.

Kapitel 2

State of the Art

Deduktive Datenbanken sind Datenbankmanagementsysteme (DBMS), die in Bezug auf ihre Abfragesprache und Datenhaltung auf Basis eines logischen Datenmodells entworfen werden (Ullman & Ramakrishnan, 1995). Deduktive Datenbanken arbeiten auf einer höheren Ebene als relationale Datenbankmanagementsysteme, da sie mit Hilfe von Fakten und Regeln die Ableitung von neuem Wissen aus bestehenden Daten ermöglichen. Sie werden daher oft auch als Erweiterung von relationalen Datenbankmanagementsystemen gesehen, da sie diese um Konzepte der logischen Programmierung erweitern. Diese Ableitung von neuem Wissen erfolgt durch einen Regelinterpreter, oft auch als Rule Engine oder Reasoning Engine bezeichnet. Deduktive Datenbanken haben ihren Ursprung in der automatisierten Beweisführung von mathematischen Theoremen und später in der logischen Programmierung (Ullman & Ramakrishnan, 1995). Erste Forschungsansätze, die die automatisierte Beweisführung von mathematischen Theoremen mit der Deduktion in Datenbanken in Verbindung setzen, gehen auf die späten 1960er Jahre zurück (Green & Raphael, 1968).

Die ersten deduktiven Datenbankmanagementsysteme wurden in den 1980er Jahren entwickelt. Morris et al. (1986) entwickelten mit ihrem NAIL! System ein erweitertes Datenbankmanagementsystem, das Datenbankabfragen mit Prolog-ähnlichen Regeln ermöglicht. Dieses System verwendete zunächst nur eine rein deklarative Programmiersprache, die später durch die prozedurale Programmiersprache Glue erweitert wurde (Phipps et al., 1991). Zur selben Zeit wurde am Microelectronics and Computer Consortium (MCC) das LDL-System mit dem Ziel entwickelt, eine deklarative Abfragesprache zu erschaffen, die den Funktionsumfang von Prolog mit der einfachen Verwendbarkeit von relationalen Abfragesprachen verbindet (Tsur & Zaniolo, 1986). Das LDL System wurde später von Arni et al. (2002) erweitert und unter dem Namen LDL++ veröffentlicht. In diesem Nachfolger wurden die Unzulänglichkeiten des ursprünglichen Systems überarbeitet und eine offenere Architektur eingeführt, die eine transparentere Gestaltung der Schnittstellen zu externen Datenbankmanagementsystemen und prozeduralen Programmiersprachen (C/C++) vorwies.

Die Stärke von deduktiven Datenbanken liegt in der Ableitung von Regeln und in weiterer Folge in der Erstellung von abgeleiteten Relationen. Kellogg und Travis (1981) prägte hierfür die Begriffe extensionale Datenbasis (extensional database) und intensionale Datenbasis (intensional database). Die extensionale Datenbasis beinhaltet die Fakten des deduktiven Datenbankmanagementsystems und entspricht einer Menge von Relationen in einer relationalen Datenbank. Die intensionale Datenbasis ergibt sich durch Auswertung der Fakten und Regeln aus der extensionalen Datenbasis und kann mit (Materialized) Views in einer relationalen Datenbank verglichen werden. Eine wichtige

Aufgabe eines deduktiven Datenbankmanagementsystems ist es, einen effizienten Zugriff auf die extensionale und intensionale Datenbasis zur Verfügung zu stellen. LDL/LDL++ und Glue-Nail ermöglichten jedoch noch nicht die explizite Speicherung der intensionalen Datenbasis. In den 1990er Jahren wurden mit den Systemen CORAL (Ramakrishnan et al., 1993), Aditi (Vaghanl et al., 1994), EKS-V1 (Vieille et al., 1992), LogicBase (Han et al., 1994), LOGRES (Cacace et al., 1993), Starburst (Widom, 1992) und XSB (Sagonas et al., 1994) Systeme entwickelt, die auch die Speicherung der intensionalen Datenbasis auf Sekundärspeicher erlaubten. In dieser Zeit wurden noch einige weitere deduktive Datenbankmanagementsysteme entwickelt. Eine ausführliche Übersicht und Gegenüberstellung von diesen wurde von Ullman und Ramakrishnan (1995) durchgeführt.

Die Forschung in Bezug auf deduktive Datenbankmanagementsysteme für räumliche Daten ist jedoch überschaubar. Abdelmoty et al. (1993) erhoben die Anforderungen und den potenziellen Nutzen von räumlichen deduktiven Datenbankmanagementsystemen und implementierten zur Demonstration einen Prototyp für die Extraktion von Straßennetzwerken. Y. Zhang et al. (2000) präsentierten eine objektorientierte Architektur für räumliche deduktive Datenbankmanagementsysteme und Fernandes et al. (1999) erweiterten ihr objektorientiertes deduktives Datenbankmanagementsystem ROCK & ROLL um Funktionen für die Verarbeitung von räumlichen Daten. Robare (2003) meldete 2003 ein Patent für die Architektur eines geografischen deduktiven Datenbankmanagementsystems für die Verwendung in Navigationssystemen an. Vaz et al. (2007) entwickelten ein deduktives geografisches Informationssystem auf Basis der *OpenGis Simple Features Specification for SQL* (Open Geospatial Consortium et al., 1999), einem Vorgänger des heutigen GML-Standards.

Bei diesen Arbeiten stehen jedoch relationale Datenbankmanagementsysteme als Basissysteme im Vordergrund. Der GML-Standard wurde auf Basis von XML entwickelt und hat somit ein hierarchisches und semi-strukturiertes Datenmodell zugrunde liegen. Um GML-Daten für deduktive Datenbankmanagementsysteme zugänglich zu machen, ist es notwendig, dass die Reasoning Engine auf die Daten zugreifen kann. Da deduktive Datenbankmanagementsysteme historisch als Erweiterung von relationalen Datenbanken entwickelt wurden, in einer Zeit in der XML noch nicht existierte (die erste Version von XML wurde 1998 herausgegeben), ist es naheliegend, die GML-Daten für diesen Zweck in ein relationales Schema zu transformieren. Diese Transformation ist jedoch nicht trivial, da die hierarchische und relationale Datenmodellierung zwei unterschiedliche Paradigmen der Datenstrukturierung und des Datenzugriffs sind. Die Entwicklung von Ansätzen zur Transformation von semi-strukturierten Daten begann bereits kurz nach der Veröffentlichung von XML.

Hierarchische Daten können auch als Graphen oder Baumstrukturen dargestellt werden. Daher ist die Entwicklung von Algorithmen zum Parsen dieser Strukturen eine Möglichkeit für die Transformation. Diese Ansätze werden auch als Model-Mapping Ansätze bezeichnet. Eine weitere Möglichkeit ist die Transformation anhand von Dokumenttypdefinitionen (DTD) oder XML-Schemas, was auch als Structure-Mapping bezeichnet wird. Model-Mapping Ansätze wurden unter anderem von Florescu und Kossmann (1999), Shimura et al. (1999), Kim (2001), Jiang et al. (2002) und Schmidt et al. (2000) entwickelt. He et al. (2008), Kappel et al. (2000), Klettke und Meyer (1999) und Bohannon et al. (2002) entwickelten Structure-Mapping-Ansätze für die Transformation, wobei Bohannon et al. (2002) ihre Methode um einen kostenbasierten Ansatz erweitern, der das Mapping optimieren sollte. Deutsch et al. (1999) entwickelten eine deklarative Abfragesprache, die ein Mapping zwischen semi-strukturierten Datenmodellen und relationalen

Datenmodellen ermöglichte. Dieses Mapping war zwar verlustfrei, aber konnte dennoch nicht alle XML-Strukturen in ein relationales Modell überführen, sondern musste für diese Elemente auf einen Overflow-Graphen zurückfallen.

Die bisher genannten Ansätze haben generell das Problem, dass XML-Dokumenten ein geordnetes Datenmodell zugrunde liegt, wohingegen relationale Daten ungeordnet sind. Tatarinov et al. (2002) entwickelten daher verschiedene Methoden für die Kodierung der Reihenfolge der XML-Elemente, bei der das ursprüngliche XML-Dokument aus dem relationalen Modell wiederhergestellt werden kann. Lee und Chu (2000), Lee und Chu (2001), Liu et al. (2004) und Davidson et al. (2007) kritisierten, dass sich viele Mapping-Ansätze nur auf strukturelle Aspekte konzentrieren, aber funktionale und semantische Bedingungen (z.B. Domäneneinschränkungen, Kardinalitäten) unbeachtet blieben und entwickelten deshalb Ansätze, die diese Aspekte berücksichtigten. 2005 meldete Krupa (2005) ein U.S. Patent für einen generischen Algorithmus an, bei dem das resultierende relationale Datenmodell sich nicht ändern muss, wenn sich die Struktur des XML-Dokuments ändert und bei dem das ursprüngliche XML-Dokument mit verhältnismäßig einfachen Abfragen aus dem resultierenden relationalen Schema wieder abgefragt werden kann.

Einige relationale Datenbankmanagementsysteme unterstützen auch die direkte Speicherung von XML-Dokumenten. Die XML-Dokumente werden dabei entweder in CLOB-Spalten (Character Large Object) oder in eigenen XML-Datentypen gespeichert. Diese Datenbankmanagementsysteme werden auch als XML-Enabled Systeme bezeichnet. Dazu gehören unter anderem Oracle XML DB (Murthy & Banerjee, 2003), IBM DB2 pureXML (Nicola & Kumar-Chatterjee, 2009), Microsoft SQL Server (Rys, 2005) und PostgreSQL (PostgreSQL Global Development Group, n. d.). Neben der Einbindung von XML in relationale Datenbankmanagementsysteme, existieren auch native XML-Datenbanken. Im Gegensatz zu XML-Enabled Systemen ist das zugrunde liegende Datenmodell in nativen XML-Datenbanken nicht relational, sondern hierarchisch. Die interne Speicherung unterscheidet sich auch von der Speicherung in relationalen Datenbanken und erfolgt in speziellen Datenstrukturen, die für XML-Dokumente optimiert sind. Beispiele für native XML-Datenbanken sind BaseX (Holupirek et al., 2007), eXist (Meier, 2002), MarkLogic Server (Hunter & Wooldridge, 2016) oder Sedna (Taranov et al., 2010).

Die Tatsache, dass GML ein etablierter Standard ist, der über klar definierte XML-Schemas verfügt und mit dem Object-Property-Modell (vgl. Kapitel 4.1) einer standardisierten Struktur folgt, ermöglicht die Entwicklung von Mapping-Ansätzen, die speziell auf GML-Dokumente zugeschnitten sind. Chaves et al. (2006), Zhu et al. (2006) und Zhu et al. (2007) stellen Ansätze für das Mapping von GML in ein objekt-relationales Datenmodell vor. Chaves et al. (2006) verwenden ein Set von Transformationsregeln, mit denen GML-Schemas umgewandelt werden können. Zhu et al. (2006) verwenden einen GML-Schema-Graph, auf dessen Basis das Mapping durchgeführt wird. Zhu et al. (2007) verwenden einen Algorithmus, bei dem das GML-Applikationsschema in einem Vorverarbeitungsschritt normalisiert wird und bei dem auch funktionale und semantische Bedingungen erhalten bleiben. Mao et al. (2014) und Koch und Löwner (2017) entwickelten Methoden zur Speicherung von CityGML-Dokumenten in NoSQL-Datenbanken (MongoDB und BaseX). CityGML ist ein auf GML basierendes Applikationsschema für die Definition von Stadtmodellen.

Jeung und Park (2004) und Li et al. (2004) beschäftigten sich mit der Speicherung von GML-Dokumenten in Spatial Database Management Systemen (SDBMS). Die Verarbeitung von räumlichen und zeitlichen Informationen erfordert zum Teil komplexe Operationen, für die die Unterstüt-

zung in herkömmlichen Datenbankmanagementsystemen nicht entsprechend gegeben ist. SDBMS sind Datenbankmanagementsysteme, die dafür einen erweiterten Funktionsumfang bieten. Das sind z.B. Funktionen zur Berechnung von Distanzen und Schnittpunkten, eigene geometrische Datentypen oder spezielle räumliche Indizes für die Optimierung von Abfragen. Beispiele für Spatial Database Management Systeme sind Oracle Spatial (Kothuri et al., 2008), IBM DB2 Spatial Extender (Davis, 1998), PostgreSQL PostGIS Extension (Corti et al., 2014) oder GeoMesa (Hughes et al., 2015).

Die erfolgreiche Ablage bzw. Speicherung von GML-Dokumenten, sei es in relationalen Datenbanken oder mittels anderer Systeme, ist jedoch nicht ausreichend, damit deduktive Datenbankmanagementsysteme diese verarbeiten können. Abfragen über XML-Dokumente können z.B. mittels XQuery oder XPath erfolgen. Das funktioniert in nativen XML-Datenbanken, aber auch in den gängigen XML-Enabled Systemen, da diese mittlerweile ebenfalls über eine eingebaute Unterstützung für Abfragen über XML-Dokumente verfügen. Da der GML-Standard auf XML basiert, können allgemein auch sämtliche Ansätze zur Speicherung und Abfrage von XML-Dokumenten für GML-Daten verwendet werden. Es existieren aber auch bereits Abfragesprachen für räumliche Daten und speziell für GML. Egenhofer (1994) definierten die Anforderungen an räumliche Abfragesprachen und stellten eine SQL-Erweiterung mit Funktionen für räumliche Abfragen vor. Wang et al. (2000) stellten eine Abfragesprache mit räumlichen Prädikaten für die Verwendung in Geoinformationssystemen (GIS) vor. Córcoles und González (2001) und Boucelma und Colonna (2004) schlagen die Erweiterung von existierenden XML-Abfragesprachen um räumliche Operatoren und Funktionen (z.B. Vergleichsoperatoren, Funktionen zur Berechnung von Überlappungen oder Schnittpunkten) vor. Eine ausführliche Beschreibung der Möglichkeiten zur Speicherung von GML-Dokumenten in XML-Enabled und nativen XML-Datenbanken, sowie zu Abfragesprachen für XML- und GML-Dokumente wurde von Shrestha (2004) durchgeführt.

Eine weitere Möglichkeit, um GML-Daten für deduktive Datenbankmanagementsysteme zugänglich zu machen, ist die direkte Transformation von GML in die Syntax der Reasoning Engine. Als Reasoning Engine wird in dieser Arbeit ErgoAI verwendet, das seinen Ursprung in der logischen Programmiersprache Prolog hat. ErgoAI ist eine kommerzielle Erweiterung von Flora-2, die Flora-2 um diverse Schnittstellen (vgl. Kapitel 7.3), eine fortgeschrittene Entwicklungsumgebung und weitere Features erweitert. Flora-2 ist eine Erweiterung von F-Logic, die Konzepte wie Metaprogrammierung und Defeasible Reasoning (vgl. Kapitel 7.1.5) einführt und F-Logic ist eine objektorientierte Erweiterung von Prolog. Da diese Sprachen aufeinander aufbauen, weißen sie auch großteils die gleiche oder sehr ähnliche Syntax bezüglich der Formulierung von Fakten, Regeln, Variablen und Abfragen auf. Arbeiten zur direkten Transformation von XML in Form von Mapping-Algorithmen, wie es bei der Transformation von XML in relationale Datenmodelle der Fall ist, existieren für diese Sprachen jedoch kaum. ErgoAI und Flora-2 verfügen aber bereits über einen integrierten XML-Parser und es gibt auch Ansätze zur Erweiterung von bestehenden XML-Abfragesprachen um Konzepte aus der logischen Programmierung (z.B. Negation, deduktive Regeln). W. Zhang et al. (2005) und Bassiliades et al. (2003) verfolgen einen objektorientierten Ansatz, bei dem deduktive Regeln für die Formulierung von Abfragen auf XML-Dokumente verwendet werden können. May (2003) entwickelte eine auf XPath und F-Logic basierende Abfragesprache, die auch für die Manipulation von XML-Dokumenten verwendet werden kann.

Zum Abschluss seien noch einige Arbeiten erwähnt, die mit dem Thema dieser Arbeit verwandt sind.

Die erste Version von GML wurde in Konformität zu RDF (Resource Description Framework) und RDFS (Resource Description Framework Schema), dem Standard-Datenmodell des Semantic Webs, erstellt. Mit Version 2.0 wurde das RDF-Profil verworfen und auf XML und XML Schema umgestellt (Van den Brink et al., 2013). Es existieren daher auch Arbeiten, die sich mit der Transformation und Abfrage von XML und GML in Standards zur Wissensrepräsentation aus dem Semantic Web (RDF, RDFS, OWL, usw.) beschäftigen. Kyzirakos et al. (2018), Van den Brink et al. (2013), Van den Brink et al. (2014), Faqir et al. (2019) und Hietanen et al. (2016) entwickelten Ansätze zum Mapping von GML nach RDF. Die Standard-Abfragesprache für RDF ist die vom World Wide Web Consortium entwickelte Abfragesprache SPARQL Protocol and RDF Query Language (kurz SPARQL). GeoSPARQL (Perry & Herring, 2012) ist eine Erweiterung von SPARQL, die vom Open Geospatial Consortium entwickelt wurde und als Standard für die Repräsentation und Abfrage von räumlichen Daten im RDF Format definiert wurde.

Teil I

Technische Grundlagen

Kapitel 3

XML-Konzepte und Begriffsabgrenzung

In diesem Kapitel werden XML-Konzepte erläutert, die für diese Arbeit relevant sind. Dabei wird darauf eingegangen, wie diese Konzepte in XML definiert sind und wie sie in GML verwendet werden. Es erfolgt außerdem eine Begriffsabgrenzung und Unterscheidung von ähnlichen Begriffen, welche häufig synonym verwendet werden oder miteinander in Zusammenhang stehen. Es wird auf die Konzepte von XML-Schema und XML-Dokumenten und auf deren Unterscheidung eingegangen. Außerdem wird auf die Begriffe Element und Attribut und deren Verwendung in XML und GML eingegangen. Zuletzt wird die Definition von Namespaces in XML und die Verwendung von diesen in GML diskutiert.

3.1 XML-Schema und XML-Dokument

XML-Dokumente sind Datenobjekte, die nach den Vorgaben des World Wide Web Consortiums (W3C) (World Wide Web Consortium, 2008) wohlgeformt sind. XML-Schemas beschreiben die Struktur von XML-Dokumenten mit Hilfe der XML Schema Definition Language (XSD). Sie beinhalten unter anderem Typdefinitionen, sowie Element- und Attributdeklarationen. GML stellt durch die Definition von XML-Schemas neben einem konzeptuellen Datenmodell auch ein standardisiertes Austauschformat für den Austausch von geografischen Informationen dar. XML-Dokumente sind Instanziierungen von XML-Schemas, die sich an die Definitionen dieser Schemas halten. Ein konkreter Anwendungsfall hierfür ist die Erstellung von Nachrichten im Flugverkehr (Digital NOTAM). Digital NOTAMs sind XML-Dokumente, die auf Basis von anwendungsspezifischen Schemadefinitionen erstellt werden, die auf den GML-Schemadefinitionen aufbauen. Listing 3.1 zeigt ein Beispiel für die Deklaration und Instanziierung eines GML-Elements in XML. XML-Dokumente können grundsätzlich auch ohne die Definition eines Schemas erstellt werden. Um die Interoperabilität von den Systemen zu gewährleisten, ist es jedoch ratsam, XML-Dokumente auf Basis eines XML-Schemas zu erstellen und diese gegen das Schema zu validieren. Für die Validierung stehen im Internet diverse OpenSource-Werkzeuge zur Verfügung.

3.2 Elemente und Attribute

XML-Dokumente bestehen aus Elementen, die durch einen Start-Tag <Name Attribute*> und einen End-Tag </Name> gekennzeichnet werden. Ein Sonderfall sind leere Elemente, die durch einen Start-Tag direkt gefolgt von einem End-Tag <Name Attribute* /> definiert werden. XML-Elemente stellen die Basis-Dateneinheit von XML-Dokumenten dar und müssen auf Schema-Ebene deklariert werden, damit die Validierung des Elements in einem Dokument bestanden wird (World Wide Web Consortium, 2008). Features und Properties des Object-Property-Modells (vgl. Kapitel 4.1), auf dem GML und darauf aufbauende Datenmodelle basieren, werden durch XML-Elemente dargestellt. GML-Features werden häufig auch als GML-Objekte bezeichnet, wobei diese inhaltlich über reine Datenobjekte im technischen Sinn hinausgehen, da ihnen durch das Object-Property-Modell spezielle Eigenschaften zugeschrieben werden (z.B. das Vorhandensein einer globalen ID). Listing 3.1 zeigt die Deklaration (Schema) und Verwendung (Dokument) eines GML-Objekts in XML.

Listing 3.1: XML-Element Deklaration und Verwendung

```
1  <!-- Deklaration (Schema) -->
2  <element name="TimePeriod" substitutionGroup="gml:AbstractTimeGeometricPrimitive"
   type="gml:TimePeriodType">
3     <annotation>
4        <documentation>
5           ...
6        </documentation>
7     </annotation>
8  </element>
9
10 <!-- Verwendung (Dokument) -->
11 <gml:TimePeriod gml:id="ID_HDL_E012">
12    <gml:beginPosition>2019-03-11T11:00:00Z</gml:beginPosition>
13    <gml:endPosition>2019-03-11T13:00:00Z</gml:endPosition>
14 </gml:TimePeriod>
```

XML-Elemente können auch Attribute beinhalten. XML-Attribute werden von den Start- und End-Tags eines Elements umschlossen und folgen dem Muster Name="Wert". Sie dienen dazu die Eigenschaften eines Elements genauer zu beschreiben und müssen ebenfalls in der Schemadefinition deklariert werden, bevor sie in einem Dokument verwendet werden können. Listing 3.1 zeigt die Zuordnung des gml:id Attributs zu einem TimePeriod-Objekt. Der Begriff Attribut wird in GML analog zum Attribut-Begriff in XML verwendet. Sowohl Features, als auch Properties können in GML Attribute enthalten.

3.3 Namespace

Namespaces werden in XML dazu verwendet um Elemente und Attribute eindeutig identifizieren zu können. XML wurde mit der Idee erschaffen, Dokumente erstellen zu können, die von verschiedenen Autoren stammen und von verschiedenen Systemen verwendet werden können. Das ermöglicht die Erstellung von Fachvokabularen, die wiederverwendet werden können. XML-Dokumente bestehen oft aus einer Vielzahl von Vokabularen, die aus unterschiedlichen Quellen stammen. Bei solchen

Dokumenten können jedoch Namenskollisionen entstehen, die mit Hilfe von Namespaces aufgelöst werden können. Um Namespaces in einem Dokument verwenden zu können, müssen diese vorher definiert werden. Das geschieht durch die Angabe des vordefinierten Schlüsselwortes bzw. Präfixes `xmlns`.

Listing 3.2: XML-Namespace Deklaration und Verwendung

```
1 <iwxxm:AIRMET xmlns:gml="http://www.opengis.net/gml/3.2"
  xmlns:iwxxm="http://icao.int/iwxxm/3.0">
2    <gml:TimeInstant gml:id="uuid.6ffb87f7-6eef-4bae-b4f2-016e2951d7dd">
3       <gml:timePosition>2014-05-15T15:20:00Z</gml:timePosition>
4    </gml:TimeInstant>
5 </iwxxm:AIRMET>
```

Laut World Wide Web Consortium (2006) bestehen Namespaces aus einem Namen und einer IRI (Internationalized Resource Identifier) Referenz. Sie gelten für das Element, in dem sie definiert wurden und dessen Kindelemente. Deshalb werden diese häufig schon beim Root-Element des Dokuments angegeben. Sobald ein Namespace definiert ist, kann dieser für die Qualifizierung von Elementen und Attributen verwendet werden. Listing 3.2 zeigt die Deklaration und Verwendung der GML- und IWXXM-Namespaces, die über die Schemadefinition unter der angegebenen IRI definiert sind. Die Elemente AIRMET und TimeInstant, sowie das zum TimeInstant zugehörige id-Attribut stammen aus diesen Namespaces.

Obwohl Namespaces laut XML-Spezifikation nicht verpflichtend angegeben werden müssen, sieht der GML-Standard durchaus Vorgaben für die Verwendung von Namespaces vor. Elemente müssen durch einen Namespace qualifiziert werden. Bei Attributen wird standardmäßig kein Namespace angegeben. Die einzige Ausnahme bildet das `gml:id` Attribut (Portele, 2007).

Kapitel 4

Geography Markup Language

Die Geography Markup Language (GML) ist ein Datenmodell und Austauschformat, das vom OpenGeospatial Consortium entwickelt und gewartet wird. GML stellt einen internationalen Standard zum Austausch von geografischen Informationen dar und basiert auf der *ISO 19100 Series of International Standards* und der *OpenGIS Abstract Specification* (Portele, 2007). GML stellt sowohl ein konzeptuelles Modell zur Beschreibung von geografischen Informationen bzw. zur Abstraktion von geografischen Objekten der realen Welt, als auch XML-Schemadefinitionen für den Austausch von geografischen Daten dar. Darauf basierend können dann Applikations-Schemas wie das Aeronautical Information Exchange Model (AIXM) oder das ICAO Weather Information Exchange Model (IWXXM) erstellt werden, welche in Kapitel 5 und Kapitel 6 genauer erläutert werden. Im Folgenden werden die wichtigsten Konzepte des GML-Standards beschrieben.

4.1 Object-Property-Modell

Die zwei wichtigsten Elemente eines GML Dokuments sind Objekte und Properties. GML-Objekte sind alle Objekte, die vom `gml:AbstractGMLType` abgeleitet werden. Das können Objekte wie Koordinatensysteme, Zeitintervalle oder geometrische Elemente sein. GML-Objekte werden durch ihre Properties genauer beschrieben. Properties können einfache Elemente sein (z.B. Name oder Beschreibung) oder komplexe Elemente (GML-Objekte) beinhalten. Dadurch entsteht ein Object-Property-Modell, bei dem sich Objekte und Properties abwechseln. Das bedeutet, dass ein Objekt kein anderes Objekt und ein Property kein anderes Property als direkten Kind-Knoten beinhalten darf (Portele, 2007). Listing 4.1 zeigt ein vereinfachtes Beispiel einer AIXM-Nachricht, die den Aufbau eines XML-Dokuments anhand des Object-Property-Modells veranschaulicht.

Das Root-Element ist in diesem Beispiel `message:AIXMBasicMessage`. Dieses enthält weitere Objekte, die immer in ein Property (z.B. `hasMember`, `timeslice`, `validTime`) geschachtelt werden müssen. Properties können weitere komplexe Objekte enthalten (z.B. `aixm:AircraftGroundService`, `gml:TimePeriod`), aber auf unterster Ebene finden sich nur noch Properties, die keine weiteren Objekte mehr enthalten, sondern textuellen Inhalt darstellen (z.B. `gml:beginPosition`, `gml:endPosition`, `aixm:interpretation`). Jedes GML-Objekt besitzt laut GML-Standard ein `gml:id` Attribut. Dieses Attribut identifiziert ein GML-Objekt eindeutig. (Portele, 2007) Dazu werden häufig Universally Unique Identifier (UUID) verwendet (siehe Kapitel 5.1).

Listing 4.1: Object-Property-Modell

```
<message:AIXMBasicMessage gml:id="M00001">
   <message:hasMember>
      <aixm:AircraftGroundService gml:id="A00001">
         <aixm:timeSlice>
            <aixm:AircraftGroundServiceTimeSlice gml:id="ID_HDL_T011">
               <gml:validTime>
                  <gml:TimePeriod gml:id="ID_HDL_T012">
                     <gml:beginPosition>2019-03-11T11:00:00Z</gml:beginPosition>
                     <gml:endPosition>2019-03-11T13:00:00Z</gml:endPosition>
                  </gml:TimePeriod>
               </gml:validTime>
               <aixm:interpretation>TEMPDELTA</aixm:interpretation>
            </aixm:AircraftGroundServiceTimeSlice>
         </aixm:timeSlice>
      </aixm:AircraftGroundService>
   </message:hasMember>
</message:AIXMBasicMessage>
```

Die ursprüngliche Version von GML basierte jedoch noch nicht auf XML, sondern auf RDF. Das Object-Property-Modell wurde bereits in dieser ursprünglichen Version eingeführt (Van den Brink et al., 2014). Die Grundidee von RDF ist es, Aussagen über zwei Entitäten in der Form von Tripeln zu machen. Ein Tripel besteht aus <Subjekt><Prädikat><Objekt>. Der Zusammenhang zwischen Subjekt und Objekt wird über das Prädikat hergestellt und die Subjekte und Objekte werden im Normalfall über eine eindeutige URI identifiziert. Ein Beispiel in natürlicher Sprache dafür ist 'Der Flughafen München (MUC) verfügt über die Flugzeuglandebahn Runway1'. In RDF kann das wie in Listing 4.2 dargestellt werden.

Sowohl der Flughafen München, als auch die Landebahn sind über ihre URI im `rdf:about` Attribut eindeutig identifiziert. Diese Landebahn kann wiederum Properties beinhalten, diese wurden aber zur Vereinfachung des Beispiels weggelassen. Der grundlegende Unterschied der heutigen auf XML basierenden Version zu der auf RDF basierenden Version ist jedoch, dass in der heutigen Version in sämtlichen Properties, die wiederum auf ein GML-Objekt verweisen, diese GML-Objekte reifiziert (als eigene Objekte definiert) werden müssen. In der heutigen Version ist die Vergabe der `gml:id` bei GML-Objekten durch das GML-Schema vorgeschrieben. Das war in der ursprünglichen RDF Version nicht der Fall, da RDF auch Blank Nodes zulässt. Blank Nodes werden dazu verwendet, Aussagen über etwas zu machen, das man nicht näher spezifizert. In RDF könnte man beispielsweise auch die Aussage 'Der Flughafen München (MUC) verfügt über etwas, auf dem ein Flugzeug landen kann' abbilden. Diese Aussage kann in RDF wie in Listing 4.3 dargestellt werden.

Listing 4.2: RDF-Tripel Beispiel

```
<rdf:Description rdf:about="http://<URI_Zu_Flughafen>/MUC">
   <landebahn>
      <rdf:Description rdf:about="http://<URI_Zu_Landebahn>/Runway1">
      ...
      </rdf:Description>
   </landebahn>
</rdf:Description>
```

Listing 4.3: RDF-Tripel mit Blank Node Beispiel

```
<rdf:Description rdf:about="http://<URI_Zu_Flughafen>/MUC">
   <landebahn>
      <rdf:Description>
      ...
      </rdf:Description>
   </landebahn>
</rdf:Description>
```

Hier wird die URI der Landebahn nicht angegeben, es wird nur ausgesagt, dass der Flughafen München eine Landebahn hat. Diese Aussage ist in der heutigen GML-Version so nicht möglich. Blank Nodes können zwar optional auch eine ID bekommen, diese ist aber nicht wie die `gml:id` von außen referenzierbar.

4.2 Geometrische Objekte

Geometrische GML-Objekte wurden auf Basis der *ISO 19107 - Geographic Information – Spatial Schema Norm* spezifiziert und lassen sich in primitive, aggregierte, zusammengesetzte und komplexe geometrische Objekte unterteilen. Sie werden vom abstrakten Typ `gml:AbstractGeometryType` abgeleitet und können als Sammlung von direkten Positionsangaben verstanden werden. Objekte dieses Typs erben eine optionale Referenz auf ein Koordinatenreferenzsystem (siehe Kapitel 4.3) und sollten wenn möglich auf ein solches verweisen. Die Positionsangaben können direkt in Form von Positionslisten (Aufeinanderfolge von Positionsangaben), Vektoren oder in Form von Envelopes erfolgen. Envelopes bestehen aus einer Angabe von gegenüberliegenden Koordinatenpaaren, welche die Eckpunkte von geometrischen Objekten in verschiedenen Dimensionen definieren (Portele, 2007).

4.2.1 Primitive geometrische Objekte

Primitive geometrische Objekte sind Objekte, die nicht mehr weiter zerlegt werden können. Bei diesen Objekten wird davon ausgegangen, dass ihre Ausrichtung anhand der Reihenfolge ihrer Positionsangaben abgeleitet werden kann. Sie werden von `gml:AbstractGeometricPrimitiveType` abgeleitet und lassen sich in 0- bis 3-dimensionale Objekte gliedern. Das einfachste geometrische Objekt ist der Punkt (0-dimensional), welcher durch eine einzelne Positionsangabe definiert werden kann. Zu den 1-dimensionalen Objekten zählen Kurven, Linien, Kreise, Bögen und weitere spezielle Ausprägungen von Kurven (z.B. Bezierkurven). Letztendlich lassen sich die meisten 1-dimensionalen Objekte auf Kurven zurückführen, denn eine Linie ist auch nur eine Kurve ohne Zwischensegmente mit Start- und Endpunkt und einer linearen Interpolation zwischen diesen Punkten.

2-dimensionale Objekte definieren geometrische Oberflächen in unterschiedlicher Komplexität. Im einfachsten Fall ist das ein einzelnes Polygon, dessen Umgrenzungen in der gleichen Ebene liegen. Eine Oberfläche kann jedoch auch aus mehreren Polygonen zusammengesetzt werden, die als mehrflächige Oberfläche zusammengefasst werden können. GML beinhaltet außerdem vordefinierte Spezialvarianten von Polygonen wie z.B. Dreiecke oder Rechtecke, auch wenn diese in GML technisch gesehen nicht von Polygonen abgeleitet werden.

3-dimensionale Objekte werden in GML auch als Festkörper (Solid) bezeichnet. `gml:SolidType` stellt die Ausgangsbasis für 3-dimensionale GML-Objekte dar und wird durch die Umgrenzungen der Oberflächen definiert, aus denen dieser besteht. Im Fall von Festkörpern werden diese Umgrenzungen mit Hüllen (Shell) definiert. Hüllen sind abgeschlossene Komponenten von ganzheitlichen Umgrenzungen, die wiederum aus einer Reihe von Oberflächen-Komponenten bestehen können (Portele, 2007). Ein vereinfachtes Beispiel des Aufbaus eines 3-dimensionalen GML-Objekts wird in Listing 4.4 dargestellt.

Listing 4.4: 3-dimensionales GML-Objekt

```
<gml:Solid gml:id="SOLID_00001">
  <gml:description>Fiktives Solid Beispiel</gml:description>
  <gml:exterior>
    <gml:Shell>
      <gml:surfaceMember>
        <gml:Polygon gml:id="POLYGON_00001">
          <gml:description>Einfaches Polygon</gml:description>
          <gml:exterior>
            <gml:LinearRing>
              <gml:pos>1.0 5.0</gml:pos>
              <gml:pos>10.0 10.0</gml:pos>
              <gml:pos>20.0 30.0</gml:pos>
              <gml:pos>1.0 5.0</gml:pos>
            </gml:LinearRing>
          </gml:exterior>
          <gml:interior>
          ...
          </gml:interior>
        </gml:Polygon>
      </gml:surfaceMember>
      <gml:surfaceMember>
        <gml:Polygon gml:id="POLYGON_00002">...</gml:Polygon>
      </gml:surfaceMember>
      <gml:surfaceMember>
        <gml:Polygon gml:id="POLYGON_00003">...</gml:Polygon>
      </gml:surfaceMember>
    </gml:Shell>
  </gml:exterior>
  <gml:interior>
    <gml:Shell>...</gml:Shell>
  </gml:interior>
</gml:Solid>
```

Dieses Beispiel zeigt einen Festkörper, der seine äußere Grenze durch eine Hülle definiert, die selbst aus mehreren Oberflächenelementen (Polygone) besteht. Die Grenzen des ersten Polygons werden durch einen linearen Ring mit mehreren direkten Positionsangaben definiert.

4.2.2 Komplexe geometrische Objekte

Primitive geometrische Objekte stellen die Grundlage für die Definition von geometrischen Strukturen dar. Es ist aber auch möglich diese zu sammeln und zu kombinieren, um komplexere Strukturen darstellen zu können. Dabei unterscheidet GML zwischen Aggregationen, zusammengesetzten geometrischen Objekten und geometrischen Komplexen.

Aggregationen sind Sammlungen von beliebigen geometrischen Objekten und werden vom abstrakten Typ `gml:AbstractGeometricAggregateType` abgeleitet. Dies ist die offenste Form der Kombination von geometrischen Objekten, da nicht davon ausgegangen wird, dass diese Sammlungen eine spezielle Struktur aufweisen. Geometrische Aggregationen können ebenfalls in 0- bis 3-dimensionale Aggregationen unterteilt werden und für die meisten primitiven geometrischen Objekte existiert auch ein entsprechender Aggregationstyp. Für 0- und 1-dimensionale Objekte (Punkte und Kurven) existieren die entsprechenden Aggregationstypen `gml:MultiPointType` und `gml:MultiCurveType`. Für 2-dimensionale Oberflächen existiert der Aggregationstyp `gml:MultiSurfaceType`. Dieser kann eine Sammlung von beliebigen Elementen vom Typ `gml:AbstractSurface` enthalten. Für 3-dimensionale Objekte existiert der Aggregationstyp `gml:MultiSolidType`, welcher eine Sammlung von Objekten vom Typ `gml:AbstractSolid` enthalten kann.

Zusammengesetzte geometrische Objekte sind Sammlungen von primitiven geometrischen Objekten. Sie werden je nach Dimensionalität von `gml:CompositeCurve`, `gml:CompositeSurface` oder `gml:CompositeSolid` abgeleitet. Der Unterschied zu Aggregaten ist, dass zusammengesetzte Objekte selbst wieder die Struktur eines primitiven geometrischen Objekts aufweisen müssen. Die Sammlung der Mitglieder von zusammengesetzten geometrischen Objekten muss selbst wieder als homogenes primitives Objekt betrachtet werden können. Das bedeutet für zusammengesetzte Kurven, dass jede Teilkurve am Endpunkt der vorhergehenden Kurve fortgesetzt werden muss. Zusammengesetzte Oberflächen müssen Mitglieder enthalten, die aneinander angrenzen und als Ganzes betrachtet wiederum eine abgeschlossene Oberfläche darstellen. Zusammengesetzte Festkörper müssen an gemeinsamen Oberflächenteilen verbunden sein und gesamt betrachtet einen abgeschlossenen Festkörper darstellen (Portele, 2007).

4.3 Koordinatenreferenzsysteme

Die Schemadefinition für Koordinatenreferenzsysteme in GML implementiert die *ISO19111 - Geographic Information – Spatial referencing by coordinates* Norm. Sie dient zur Darstellung von geografischen Positionsangaben in verschiedenen Koordinatensystemen und zur Transformation der Koordinaten zwischen den Systemen. Die tatsächliche Position von geografischen Elementen in der realen Welt ist abhängig vom gewählten Koordinatenreferenzsystem. Dieses kann durch die Angabe des `srsName` Attributs bei der Definition eines geometrischen Objekts angegeben werden.

Geografische Informationen können neben räumlichen Informationen auch eine Zeitkomponente enthalten. Geografische Positionen können sich beispielsweise aufgrund von tektonischen Plattenbewegungen oder aufgrund der Beweglichkeit von Elementen (z.B. Fahrzeuge, Flugzeuge) mit der Zeit ändern. GML unterstützt eine Vielzahl von Koordinatenreferenzsystemen, unter anderem auch räumlich-zeitliche Systeme. Im Folgenden werden zunächst einige wichtige Begrifflichkeiten und die verschiedenen Arten von Koordinatenreferenzsystemen beschrieben, die von GML unterstützt werden.

Ein Koordinatenreferenzsystem dient zur Bestimmung der Position von Objekten in der realen Welt und besteht aus einem Koordinatensystem und einem Datum. Dazu gehören beispielsweise geodätische und vertikale Systeme, sowie Image- und Engineering-Systeme. Ein Koordinatensystem ist eine Menge von mathematischen Regeln, die definieren, wie Koordinaten zu geografischen Punkten zugeordnet werden. Es gibt verschiedene Arten von Koordinatensystemen, wie geodätische, sphärische, ellipsoide, vertikale, kartesische oder zylindrische Koordinatensysteme. Ein Datum ist eine Menge von Parametern, die den Ursprung, Maßstab und die Orientierung eines Koordinatensystems definieren. Es wird häufig auch der Begriff Referenzrahmen anstelle von Datum verwendet, da ein Datum den Bezugspunkt zu den Positionen in der realen Welt bestimmt. Zu den Datumsarten zählen beispielsweise geodätische, vertikale und temporale Daten. Koordinaten sind skalaren Zahlen, die die Position eines Punktes bestimmen. Ein Koordinatentupel ist eine geordnete Liste von Koordinaten, wobei die Anzahl der Koordinaten der Dimensionalität des Koordinatenraumes entspricht. Bei einem Koordinatenset handelt es sich um eine Sammlung von Koordinatentupeln, die das gleiche Koordinatenreferenzsystem referenzieren (Open Geospatial Consortium, 2018).

4.3.1 GML-unterstützte Koordinatenreferenzsysteme

Die Klassifizierung von Koordinatenreferenzsystemen erfolgt anhand der Art des verwendeten Datums und in manchen Fällen anhand des verwendeten Koordinatensystems. Koordinatenreferenzsysteme können in geodätische Systeme, Engineering-Systeme, vertikale Systeme, parametrische Systeme und temporale Systeme eingeteilt werden. Neben diesen Hauptklassen existieren noch die Sonderklassen abgeleitete Systeme und zusammengesetzte Systeme (Open Geospatial Consortium, 2018). Diese Arten von Koordinatenreferenzsystemen werden im Folgenden näher erläutert.

Geodätische Systeme

Geodätischen Systeme sind 2- oder 3-dimensionale Koordinatenreferenzsysteme, die zur Beschreibung von räumlichen Punkten über die gesamte Erde oder einen Großteil dieser dienen. Eine Sonderform von geodätischen Systemen sind geografische Systeme. Das sind Koordinatenreferenzsysteme, die mit einem ellipsoiden Koordinatensystem verbunden sind, was meistens bei kartesischen und sphärischen Koordinatensystemen der Fall ist.

Geodätische Systeme lassen sich in statische und dynamische Systeme unterscheiden. Statische Systeme ignorieren die Tatsache, dass sich die tektonischen Platten der Erde mit der Zeit verschieben. Diese sind oftmals regional auf tektonische Platten beschränkt, da sich aus Sicht eines Beobachters, der sich auf der tektonischen Platte befindet, die geografischen Positionen nicht ändern. Dynamische Systeme berücksichtigen die tektonische Plattenverschiebung und somit die Tatsache, dass sich geografische Koordinaten mit der Zeit ändern. Da dynamische Systeme komplexer sind, werden regionale und nationale Referenzsysteme häufig statisch auf die tektonische Platte fixiert, auf der sie sich befinden (Open Geospatial Consortium, 2018).

Engineering-Systeme

Engineering-Systeme sind lokal beschränkte Koordinatenreferenzsysteme, die für Ingenieurstätigkeiten verwendet werden. Sie lassen sich in 3 Subtypen unterteilen (Open Geospatial Consortium, 2018):

- Koordinatenreferenzsysteme zur Beschreibung von geografischen Punkten auf einem beschränkten Teil der Erde. Diese verwenden eine Annäherung für die 2-dimensionale Projektion der Erdoberfläche (Flat-Earth-Approximation), welche keine Korrekturen für die Krümmung der Erde berücksichtigt. Aus diesem Grund sind diese Systeme auch nur über kurze Distanzen sinnvoll, da ansonsten zu große Ungenauigkeiten bei der Berechnung von Distanzen entstehen. Ein typisches Anwendungsgebiet ist das Bauwesen.
- Koordinatenreferenzsysteme zur Beschreibung von räumlichen Standorten von beweglichen Objekten wie beispielsweise Fahrzeuge, Schiffe oder Flugzeuge.
- Koordinatenreferenzsystem zur Beschreibung von räumlichen Punkten intern auf einem Bild. Diese sind lokal auf das Bild beschränkt und im Normalfall nicht geographisch referenziert, außer das Bild wird durch die Transformation in ein geodätisches oder projiziertes Koordinatenreferenzsystem referenziert.

Vertikale Systeme

Vertikale Systeme sind räumliche Koordinatenreferenzsysteme, die zusätzlich einen vertikalen Referenzrahmen zur Angabe von Höheninformationen beinhalten. Dieser vertikale Referenzrahmen kann auf verschiedene Arten realisiert werden (Open Geospatial Consortium, 2018):

- **Levelling:** Hier wird der Nullpunkt der vertikalen Achse aufgrund der Beobachtung des Meeresspiegels bei Ebbe und Flut über einen gewissen Zeitraum definiert und dann lokal bzw. regional veröffentlicht.
- **Geoid:** Hier wird der Nullpunkt der vertikalen Achse aufgrund der Annäherung an den durchschnittlichen Meeresspiegel definiert. Die Angabe der Höhe basiert auf einem Referenz-Ellipsoid in einem geodätischen Koordinatenreferenzsystem.
- **Tidal:** Der Nullpunkt der vertikalen Achse ist anwendungsabhängig und wird anhand einer Oberfläche definiert, die für das jeweilige Einsatzgebiet eine Bedeutung hat. Beispiele hierfür sind Lowest Astronomical Tide (LAT) und Lowest Low Water Springs (LLWS).

Parametrische Systeme

Parametrische Systeme sind 1-dimensionale Koordinatensysteme, die Parameter oder Funktionen als Koordinaten verwenden. Diese werden häufig in wissenschaftlichen Communities verwendet, insbesondere in den Umweltwissenschaften. Ein Beispiel für parametrische Koordinatenreferenzsysteme ist die Verwendung des barometrischen Drucks zur Berechnung der Höhe. Dieser kann stellvertretend für die Angabe der Höhe verwendet werden, da der barometrische Druck mit zunehmender Höhe stetig abnimmt. Die exakte Umrechnung in eine Höhenangabe hängt von dem jeweiligen lokalen atmosphärischen Profil ab. Der barometrische Druck wird häufig als Standardmaß für Höhenangaben im Flugverkehr und der Meteorologie verwendet (Open Geospatial Consortium, 2018).

Temporale Systeme

Temporale Systeme sind 1-dimensionale Koordinatenreferenzsysteme, die zur Beschreibung von Zeit verwendet werden. Sie bestehen ebenfalls aus einem Koordinatensystem und einem Datum. In diesem Fall ist das Datum jedoch keine geografische Komponente, sondern die Zeit. Häufig wird für die Angabe des Datums der gregorianische Kalender verwendet. Es wäre aber auch die Hinterlegung eines anderen Kalenders wie z.B. des Marskalenders denkbar. Für Zeitangaben in temporalen Systemen gibt es folgende Möglichkeiten (Open Geospatial Consortium, 2018):

- **dateTime:** Angabe eines Zeitstempels wie nach der *ISO 8601 - Data elements and interchange formats – Information interchange – Representation of dates and times* Norm definiert.
- **temporal count:** Angabe von diskreten Werten, wobei die Zeitachse auf dem jeweils definierten Kalender/Datum beruht.
- **temporal measure:** Angabe von kontinuierlichen Werten, wobei die Zeitachse auf dem jeweils definierten Kalender/Datum beruht.

Die Angabe und Umrechnung von zeitlichen Informationen ist nicht trivial, da aufgrund von unterschiedlich langen Monaten oder kalendarischen Anpassungen (z.B. Schaltjahr) die Kalenderarithmetik eine Rolle spielt. Kalenderarithmetik beschreibt den Prozess des Addierens bzw. Subtrahierens von zeitlichen Einheiten. Es muss beispielsweise beim Addieren eines Monats mit Ausgangspunkt am 30. Jänner berücksichtigt werden, ob es sich um ein Schaltjahr handelt oder nicht, da der Februar in einem Schaltjahr einen zusätzlichen Tag hat.

Abgeleitete Systeme

Abgeleitete Systeme sind Koordinatenreferenzsysteme, die durch die Konvertierung von Koordinaten eines anderen bereits existierenden Systems definiert werden. Dieses System wird auch als Basissystem bezeichnet und das abgeleitete System übernimmt das Datum des Basissystems. Im Normalfall haben abgeleitete Systeme das selbe Koordinatensystem wie das Basissystem. Ein abgeleitetes geografisches Koordinatenreferenzsystem hat beispielsweise immer ein ellipsoides Koordinatensystem, da geografische Koordinatenreferenzsysteme ein ellipsoides Koordinatensystem verwenden müssen.

Eine Sonderform von abgeleiteten Systemen sind projizierte Systeme. Diese werden von einem geografischen Koordinatenreferenzsystem abgeleitet und verwenden eine Koordinatenprojektion in Längen- und Breitengrade. Diese sind im Umgang mit geografischen Informationen sehr weit verbreitet. Projizierte Koordinatenreferenzsysteme können selbst wiederum die Basis für die Ableitung von weiteren Systemen sein, wobei eventuell auftretende Verzerrungen bzw. Ungenauigkeiten bei der Kartenprojektion dann in den abgeleiteten Systemen übernommen werden (Open Geospatial Consortium, 2018).

Zusammengesetzte Systeme

Zusammengesetzte Koordinatensysteme sind die Kombination von zwei oder mehreren Koordinatenreferenzsystemen. Die Teilsysteme dürfen selbst jedoch keine zusammengesetzten Systeme sein und müssen voneinander unabhängig sein. In der Entstehung der verschiedenen Koordinatensysteme hat sich eine Trennung von horizontalen und vertikalen Systemen entwickelt und es ist bis heute üblich, horizontale und vertikale Koordinatensysteme als Alternative zu einem integrierten 3-dimensionalen Ansatz zu kombinieren. Zusammengesetzte räumliche Systeme können außerdem mit temporalen Systemen kombiniert werden um ein räumlich-zeitliches Koordinatenreferenzsystem zu erstellen.

Tabelle 4.1: Kombination von zusammengesetzten Koordinatensystemen

Typ	**Teilsysteme**
Räumlich	Geografisch 2-dimensional und Vertikal Geografisch 2-dimensional und Engineering 1-dimensional Abgeleitet 2-dimemsional und Vertikal Abgeleitet 2-dimensional und Engineering 1-dimensional Engineering 2-dimensional und Vertikal Engineering 1-dimensional, Linear und Vertikal
Räumlich-Zeitlich	Sämtliche 2-dimensionale räumliche Systeme und ein oder mehrere temporale Systeme. Die Kombination von mehreren unabhängigen temporalen Systemen ist zulässig (z.B. Geografisch 2-dimensional und Temporal).
Räumlich-Parametrisch	Sämtliche 2-dimensionale räumliche Systeme und ein oder mehrere parametrische Systeme. Die Kombination von mehreren unabhängigen parametrischen Systemen ist zulässig (z.B. Projiziert 2-dimensional und Parametrisch)
Räumlich-Parametrisch-Zeitlich	Sämtliche Räumlich-Parametrische Systeme und Temporale Systeme

Dabei ist zu beachten, dass die Angabe der Koordinaten der Reihenfolge entspricht, in der die Teilsysteme des zusammengesetzten Koordinatensystems definiert sind. Eine klare Reihenfolge ist nicht vorgegeben, es ist jedoch üblich, dass horizontale Systeme vor vertikalen Systemen angegeben werden und räumliche Systeme vor temporalen Systemen. Es dürfen außerdem keine räumlichen Systeme miteinander kombiniert werden, wenn diese eine gleiche Achse beinhalten (z.B. mehrere 2-dimensionale Systeme). Tabelle 4.1 zeigt eine Übersicht über die gültigen Kombinationen von zusammengesetzten Koordinatenreferenzsystemen und deren Kategorisierung (Open Geospatial Consortium, 2018).

4.4 Zeitliche Informationen

Das temporale GML-Schema unterstützt die Beschreibung von temporalen Referenzsystemen und zeitlichen Aspekten von geografischen Informationen. GML verwendet dafür als Basis ein räumlich-zeitliches Referenzsystem, wie es in der *ISO 19108 - Temporal schema (temporal geometry and topology objects, temporal reference systems)* Norm definiert ist. Die Angabe von zeitlichen Informationen wird auf Feature- und Propertyebene unterstützt.

Zeit kann auf zwei verschiedene Arten gemessen werden, nämlich in Form von Intervallen oder auf einer Ordinalskala. Eine Intervallskala stellt die Grundlage für die Messung von Zeitdauer dar, wohingegen eine Ordinalskala nur einen bestimmten Zeitpunkt abbildet. Die Darstellung von Zeitintervallen und Zeitpunkten in GML basiert auf den Normen *ISO 8601 - Data elements and interchange formats — Information interchange — Representation of dates and times* und *ISO/IEC 11404 - Information technology — Programming languages, their environments and system software interfaces — Language-independent datatypes*, welche die Kodierung von Zeitintervallen und Zeitpunkten als Strings mit bestimmter Struktur beschreiben (Portele, 2007).

Die Definition von Objekten und Datentypen zur Beschreibung von zeitlichen Informationen erfolgt in GML in einem eigenen Schema, dem temporalen GML-Schema. Zeitliche Angaben werden in GML von `gml:AbstractTimeObject` abgeleitet. Die Beschreibung von zeitlicher Geometrie erfolgt in Form von Zeitpunkten, Zeitintervallen, Positionen und Längen. Diese Angaben werden von `gml:AbstractTimeGeometricPrimitive` abgeleitet. Eine temporale Geometrie sollte immer mit einem temporalen Referenzsystem verbunden sein (vgl. Kapitel 4.3.1) und GML sieht dafür auch ein eigenes `frame` Attribut vor. Laut der ISO 19108 Norm wird standardmäßig der gregorianische Kalender als zeitliches Referenzsystem verwendet, wobei auch andere Systeme zulässig sind. Für die Darstellung von Zeitpunkten und Zeitintervallen bietet GML die Typen `gml:TimeInstant` und `gml:TimePeriod`.

Listing 4.5: TimeInstant

```
1 <gml:TimeInstant gml:id="t11">
2    <gml:description>Hochzeitstag</gml:description>
3    <gml:timePosition>2001-05-23</gml:timePosition>
4 </gml:TimeInstant>
```

Ein `gml:TimeInstant` Objekt bezeichnet ein primitives Objekt, das einen eindeutig identifizierbaren Punkt in der Zeit beinhaltet. Im Normalfall enthält ein `gml:TimeInstant` Objekt ein `gml:timePosition` Property, das den Zeitpunkt bzw. das Datum als Wert enthält. Ein `gml:TimePeriod` Objekt ist ein primitives Objekt, das einen eindeutig identifizierbaren Zeitraum angibt. Dieser Zeitraum wird mit einem Start- und Endzeitpunkt definiert. Dieser kann wie beim `gml:TimeInstant` Objekt direkt durch die Angabe eines `gml:timePosition` Attributs erfolgen oder auf ein bereits existierendes `gml:TimeInstant` Objekt referenzieren. Optional kann ein `gml:duration` Property angegeben werden, welches von `xsd:duration` abgeleitet wird und eine alternative Schreibweise für Datums- und Zeitangaben darstellt.

Listing 4.6: TimePeriod mit Angabe von TimeInstant Objekten

```
<gml:TimePeriod gml:id="p22">
   <gml:begin>
      <gml:TimeInstant gml:id="t11">
         <gml:timePosition>2001-05-23</gml:timePosition>
      </gml:TimeInstant>
   </gml:begin>
   <gml:end>
      <gml:TimeInstant gml:id="t12">
         <gml:timePosition>2001-06-23</gml:timePosition>
      </gml:TimeInstant>
   </gml:end>
</gml:TimePeriod>
```

Listing 4.7: TimePeriod mit direkter Angabe von Zeitpositionen

```
<gml:TimePeriod gml:id="p22">
   <gml:beginPosition>2002-05-20</gml:beginPosition>
   <gml:endPosition>2002-05-23</gml:endPosition>
   <gml:duration>P1Y</gml:duration>
</gml:TimePeriod>
```

Die Beschreibung von Zeitangaben ist abhängig vom verwendeten temporalen Referenzsystem. Aus diesem Grund ist es wichtig, dass das Referenzsystem angegeben wird. Dieses muss jedoch nicht immer in vollem Umfang definiert werden, es reicht auch die Verlinkung auf ein bestehendes Referenzsystem mit einer gültigen URI. Zeitangaben, die auf Kalendern basieren, werden als strukturierte Textformate repräsentiert wie sie in der *ISO 8601 - Data elements and interchange formats – Information interchange – Representation of dates and times* Norm definiert sind. Für Kalender, die mehr als eine Ära beinhalten (z.B. japanischer Kalender), kann das optionale Attribut `gml:calendarEraName` angegeben werden. Nicht genau spezifizierte Zeitangaben können durch das optionale Attribut `gml:indeterminatePosition` angeben werden, welches folgende Ausprägungen zulässt (Portele, 2007):

- **unknown:** Zeitangabe ist nicht bekannt bzw. wird nicht angegeben.
- **now:** Der Wert wird mit dem aktuellen Zeitstempel ersetzt, sobald dieser aufgerufen wird.
- **before:** Die genaue Zeitangabe ist nicht bekannt. Es ist nur bekannt, dass sie sich zeitlich vor dem genannten Wert befindet.
- **after:** Die genaue Zeitangabe ist nicht bekannt. Es ist nur bekannt, dass sie sich zeitlich nach dem genannten Wert befindet.

4.4.1 Repräsentation von dynamischen Features

In Kapitel 4.4 wurden die wichtigsten Konzepte zur Darstellung von zeitlichen Angaben in GML vorgestellt. Diese können verwendet werden, um Eigenschaften von geografischen Objekten darzustellen, die sich mit der Zeit ändern können. GML implementiert hierfür drei grundlegende temporale Entitäten, die von Langran (1992) festgelegt wurden, nämlich Zustände (States), Ereignisse (Events) und Nachweise (Evidence). Nachweise werden in GML durch die Properties `gml:dataSource` bzw. `gml:dataSourceReference` dargestellt und bezeichnen die Quelle der zeitlichen Daten (z.B. menschlicher Beobachter, Sensor). Der Status eines Features wird durch einen Zeitstempel eingefangen und jede Instanz eines Features mit einem Zeitstempel stellt einen Snapshot (vgl. Kapitel 5.2) dieses Features dar. Features können aber auch eine Historie von gültigen Zuständen beinhalten. Die Historie wird durch ein oder mehrere TimeSlices ((vgl. Kapitel 5.2) dargestellt.

Ein TimeSlice beinhaltet die zeitlich variierenden Properties eines Features und reflektiert somit ein Ereignis, das zu einem geänderten Zustand geführt hat. TimeSlices enthalten jedoch nur die dynamischen Werte, die sich auch aufgrund eines Ereignisses tatsächlich geändert haben. Ein Beispiel ist der Kauf eines Hauses, bei dem sich der Name des Besitzers ändert. Bleiben alle anderen Attribute des Hauses unverändert, enthält das TimeSlice nur das geänderte Besitzerattribut. Snapshots beinhalten hingegen immer den Gesamtzustand eines Features zu einem bestimmten Zeitpunkt. GML stellt damit ein zeitliches Modell bereit, das bis auf Attribut-Ebene granulare Zustandsänderungen abbilden kann. Dieses Zeitmodell ist ein wichtiges Konzept von GML und wird in vielen anwendungsspezifischen GML-Applikationsschemas übernommen (Portele, 2007).

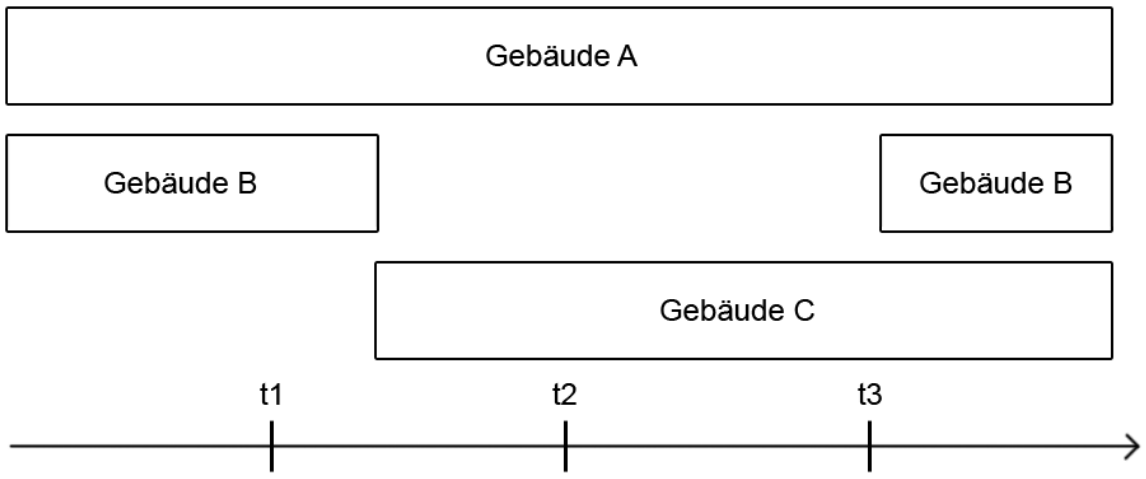


Abbildung 4.1: Lebenszyklus eines Gebäudekomplexes

Abbildung 4.1 zeigt einen Gebäudekomplex, der aus mehreren Teilgebäuden besteht. Zum Zeitpunkt t1 besteht dieser Komplex nur aus Gebäude A und Gebäude B. Im Zeitraum zwischen t1 und t2 wird ein neues Gebäude C gebaut, wobei Gebäude B wieder verschwunden ist. Zum Zeitpunkt t3 umfasst der Gebäudekomplex alle 3 Gebäude. Gebäude B könnte beispielsweise ein Gebäude sein, das nur saisonal aufgebaut wird.

4.5 Maßeinheiten

Maßeinheiten sind ein wichtiger Bestandteil von GML, da sie in vielen Komponenten verwendet werden und notwendig sind. Die Definition der Maßeinheit von Werten findet man in GML beispielsweise bei Properties und Features, bei der Beschreibung von Beobachtungen oder bei der Definition von Koordinatenreferenzsystemen. Maßeinheiten werden zwischen Basiseinheiten, konventionellen Einheiten und abgeleiteten Einheiten unterschieden. GML sieht auch die Definition von benutzerdefinierten Einheiten vor, falls für ein bestimmtes Anwendungsgebiet noch keine passende Maßeinheit existiert.

Basiseinheiten sind die bevorzugten Einheiten für fundamentale Größen, die innerhalb eines Systems von Maßeinheiten nicht von anderen Basiseinheiten abgeleitet werden können. Im internationalen Einheitensystem (SI) sind das beispielsweise Sekunden (Zeit), Meter (Länge), Kilogramm (Masse), Ampere (Stromstärke), Kelvin (Temperatur), Mol (Stoffmenge) und Candela (Lichtstärke). Abgeleitete Einheiten sind Einheiten innerhalb eines Systems, die durch Kombination oder algebraische Umrechnung von Basiseinheiten definiert werden können. Beispiele für abgeleitete Einheiten sind Hertz (Frequenz), Meter pro Sekunde (Geschwindigkeit) oder Quadratmeter (Fläche). Konventionellen Einheiten sind weder Basiseinheiten noch abgeleitete Einheiten, da diese keinen direkten Bezug zu Basiseinheiten haben oder durch Kombination von Basiseinheiten entstehen. Im Normalfall besteht aber durchaus eine Umrechnungskonvention in Basiseinheiten oder abgeleitete Einheiten. Beispiele für konventionelle Einheiten sind Fuß (Länge) oder nautische Meilen (Seemeilen) (Portele, 2007).

4.6 Richtungsvorgaben

GML sieht ein Schema zur Beschreibung von Richtungsangaben und zur Lage und Ausrichtung von Objekten vor. Diese Informationen werden durch das `gml:direction` Attribut ausgedrückt und Features zugeordnet. Richtungsangaben können in Form von Richtungsvektoren oder Richtungsbeschreibungen erfolgen. Richtungsbeschreibungen können Angaben in Form von Kompassrichtungen, Schlüsselwörtern oder textuelle Beschreibungen sein (z.B. südlich von Wien) (Portele, 2007).

4.7 Beobachtungen

GML beinhaltet ein Schema für die Beschreibung von Beobachtungen. Das können menschliche Beobachtungen, Fotoaufnahmen mit einer Kamera oder Aufnahmen durch ein anderes Instrument sein. Sie werden in GML durch ein Observation-Feature dargestellt. Beobachtungen beinhalten die Metadaten über den Prozess der Informationssammlung und das Ergebnis der Beobachtung. Häufig werden bei Beobachtungen die Properties `gml:validTime` (Zeitpunkt der Beobachtung), `gml:using` (Prozess der Aufzeichnung) und `gml:target` (Ziel der Beobachtung) angegeben (Portele, 2007).

4.8 Coverages

Geografische Elemente können in Elemente mit diskreten und Elemente mit kontinuierlichen Werten eingeteilt werden. Elemente mit diskreten Werten sind Objekte mit klar definierten räumlichen Grenzen wie beispielsweise Gebäude. Die Definition dieser Elemente erfolgt in GML durch geometrische Objekte (vgl. Kapitel 4.2). Elemente mit kontinuierlichen Werten variieren über die Zeit und haben keine eindeutigen Abgrenzungen (z.B. Temperatur, Bodenzusammensetzung). Die Angabe dieser Werte macht daher nur Sinn im Zusammenhang mit einer Positionsangabe.

Diskrete und kontinuierliche Elemente schließen sich jedoch nicht gegenseitig aus. Eine Stadt kann beispielsweise als diskretes Objekt mit Attributen wie Name, Fläche oder Gesamtbevölkerung gesehen werden. Sie kann aber auch als Feature betrachtet werden, das Angaben zur Bevölkerungsdichte, zum Grundstückswert oder zur Luftverschmutzung an jedem Punkt der Stadt beinhaltet. In diesem Fall sprechen wir von einer Coverage. Eine Coverage ist ein spezieller Feature-Subtyp, der für jedes Attribut mehrere Werte für jede geometrische Position beinhaltet. Sie ist eine Funktion, die Positionen einer räumlich-zeitlichen Domäne mit Werten in einem bestimmten Attributbereich verknüpft. Coverages sind die bevorzugte Datenstruktur in diversen Anwendungsgebieten (z.B. Vermessungswesen, Meteorologie, Boden- und Vegetationsforschung). Beispiele für Coverages sind Rasterbilder, polygonale Overlays oder Höhenmatrizen.

Coverages bestehen aus einer Domäne und einem Attributbereich. Die Domäne ist eine Sammlung an geometrischen Objekten, die durch direkte Positionsangaben beschrieben werden können. Ihre Positionsangaben sind an ein Koordinatenreferenzsystem gebunden und sie unterscheiden sich in der Dimensionalität ihres Koordinatenraums. 3-dimensionale Domänen können geometrische Objekte wie Punkte, Kurven, Oberflächen oder solide Objekte enthalten. 2-dimensionale Domänen können jedoch beispielsweise keine 3-dimensionale geometrische Objekte beinhalten. Der Attributbereich definiert ein Set von Feature-Attribut-Werten. Coverages mappen dieses Set auf Positionen in der Domäne (z.B. Temperatur, Druck, Luftfeuchtigkeit und Windstärke an jedem Punkt in Österreich). Der Attributbereich bezeichnet das Set bestehend aus diesen Werten und die Domäne bezeichnet den geografischen Bereich Österreich.

Eine diskrete Coverage hat eine endliche Domäne und einen endlichen Attributbereich. Jedes geometrische Objekt in der Domäne wird auf ein Set von Attributwerten gemappt. Eine diskrete Coverage könnte beispielsweise in der Bodenforschung das Mapping von einer Domäne bestehend aus mehreren Polygonen zu dem jeweiligen Bodentyp des Polygons sein. Kontinuierliche Coverages können aus einer unendlichen Domäne und einem kontinuierlichen Attributbereich bestehen (Open Geospatial Consortium, 2006). Die Angabe von Temperatur, Druck, Luftfeuchtigkeit und Windstärke in Österreich ist ein Beispiel für eine kontinuierliche Coverage, da Österreich aus unendlich vielen Punkten besteht und diese Attribute einen kontinuierlichen Wertebereich haben.

Kapitel 5

Aeronautical Information Exchange Model

Das Aeronautical Information Exchange Model (AIXM) ist ein Applikationsschema, das auf der GML 3.2 Spezifikation basiert. Es wurde von der amerikanischen Federal Aviation Administration (FAA) und der European Organisation for the Safety of Air Navigation (EUROCONTROL) entwickelt und dient als Datenmodell und strukturiertes Austauschformat im Flugverkehr. Es stellt außerdem die Grundlage zur Erstellung von digitalen NoticesToAirmen (dNOTAM), also digitalen Nachrichten im Flugverkehr dar. Es liegt derzeit in der Version 5.1 vor und beinhaltet Schemadefinitionen zu Beschreibung von Entitäten im Flugverkehr (z.B. Flughäfen, Navigationshilfen, Flugrouten). Das AIXM Datenmodell ist sehr umfangreich und wird hier deshalb nicht in vollem Umfang beschrieben. Stattdessen wird auf die grundlegenden Konzepte von AIXM eingegangen. Listing 5.1 zeigt einen Auszug aus einer AIXM-Nachricht.

AIXM-Nachrichten halten sich ebenfalls an das Object-Property-Modell (vgl. Kapitel 4.1). AIXM verwendet für die Beschreibung von räumlichen und zeitlichen Eigenschaften Elemente aus dem GML-Namespace und erweitert diese Konzepte. Elemente wie Zeitperioden und TimeSlices sind bereits in der GML-Spezifikation enthalten (vgl. Kapitel 4.4) und AIXM verwendet beispielsweise für die Beschreibung der Lebenszeit eines Features standardmäßig das TimePeriod-Objekt, wie es in GML definiert ist. AIXM verwendet das TimeSlice-Konzept um die zeitlich variierenden Eigenschaften von Features einzufangen. Zur Definition eines TimeSlices wird die Klasse `gml:AbstractTimeSlice` aus der GML-Spezifikation erweitert und für den speziellen Fall (z.B. `aixm:TaxiwayTimeSlice`) abgeleitet. AIXM stellt ein eigenes Applikationsschema dar, das auf den Konzepten des GML-Modells basiert und dieses um eigene Elemente erweitert, die durch den Namespace `aixm` gekennzeichnet sind. Außerdem verwendet AIXM zur eindeutigen Identifikation von Features das `gml:identifier` Attribut. Die Konzepte zur eindeutigen Identifikation und das AIXM-Zeitmodell werden im Folgenden näher beschrieben.

Listing 5.1: AIXM-Nachricht

```
<message:AIXMBasicMessage gml:id="M00001">
   <message:hasMember>
      <aixm:Taxiway gml:id="uuid.0cb4da5a-9c98-4484-92ed-edc336b8b437">
         <gml:identifier codeSpace="urn:uuid:">
            0cb4da5a-9c98-4484-92ed-edc336b8b437
         </gml:identifier>
         <aixm:timeSlice>
            <aixm:TaxiwayTimeSlice gml:id="twts1267">
               <gml:validTime>
                  <gml:TimePeriod gml:id="vt266126">
                     <gml:beginPosition>2009-01-01T00:00:00Z</gml:beginPosition>
                     <gml:endPosition indeterminatePosition="unknown"/>
                  </gml:TimePeriod>
               </gml:validTime>
               <aixm:interpretation>BASELINE</aixm:interpretation>
               <aixm:sequenceNumber>1</aixm:sequenceNumber>
               <aixm:featureLifetime>
                  <gml:TimePeriod gml:id="lt266126">
                     <gml:beginPosition>2009-01-01T00:00:00Z</gml:beginPosition>
                     <gml:endPosition indeterminatePosition="unknown"/>
                  </gml:TimePeriod>
               </aixm:featureLifetime>
               <aixm:designator>F</aixm:designator>
               <aixm:type>GND</aixm:type>
               <aixm:width uom="M">23.0</aixm:width>
               <aixm:surfaceProperties>
                  <aixm:SurfaceCharacteristics gml:id="S-d82b7f8d">
                     <aixm:composition>CONC</aixm:composition>
                  </aixm:SurfaceCharacteristics>
               </aixm:surfaceProperties>
               <aixm:associatedAirportHeliport
               xlink:href="urn:uuid:1b54b2d6-a5ff-4e57-94c2-f4047a381c64"/>
            </aixm:TaxiwayTimeSlice>
         </aixm:timeSlice>
      </aixm:Taxiway>
   </message:hasMember>
</message:AIXMBasicMessage>
```

5.1 Identifikation von AIXM-Features

Die eindeutige Identifikation von AIXM-Features erfolgt durch das `gml:identifier` Attribut, das vom Typ `gml:AbstractGMLType` geerbt wird, von dem alle GML-Features abgeleitet sind. Laut dem AIXM-Zeitmodell ist dieses Property das einzige, das über die Zeit konstant bleibt und wird deshalb als einziges Attribut außerhalb eines TimeSlices definiert. Die verwendeten IDs müssen folgenden Anforderungen genügen:

- **Einzigartigkeit:** Eine ID darf nicht zur Beschreibung von verschiedenen Objekten verwendet werden.
- **Universalität:** Die gleiche ID muss einheitlich in allen Systemen zur Identifikation eines bestimmten Features verwendet werden.

Zur Erfüllung der Einzigartigkeit werden in der Praxis häufig Universally Unique Identifier (UUID) benutzt. Universally Unique Identifier dienen zur Generierung von IDs, bei denen die Wahrscheinlichkeit sehr gering ist, dass die gleiche ID von einem anderen System generiert wird und somit unbeabsichtigt zur Identifikation eines anderen Objekts verwendet werden würde (EUROCONTROL & Federal Aviation Administration, 2011). Die Algorithmen zur Generierung von UUIDs existieren in verschiedenen Versionen, denen jeweils unterschiedliche Mechanismen zur ID-Generierung unterliegen. Die AIXM-Spezifikation empfiehlt die Verwendung von UUIDv4. Diese basiert auf Algorithmen zur Generierung von Zufallszahlen im Vergleich zu zeitstempel- oder namensbasierten UUIDs. UUIDv4 ist in vielen Programmiersprachen und Systemen implementiert. In Java ist das beispielsweise die Klasse `java.util.UUID`.

Die Erfüllung der Universalität ist unter Umständen nicht immer möglich. Die IDs für ein Feature sollten nach Möglichkeit nur vom Herausgeber des Features generiert werden und alle beteiligten Stakeholder sollten die gleichen Daten zu einem bestimmten Feature haben. Das ist in der Praxis oft schwierig, da zu einem Feature mehrere Pseudo-Primärquellen und somit Duplikate von Daten zu einem Feature existieren können. Es könnte vorkommen, dass in zwei verschiedenen Systemen Daten zu dem selben Feature mit unterschiedlichem `gml:identifier` existieren. Wenn das passiert, ist es notwendig, Duplikate anhand der enthaltenen Attributwerte zu identifizieren und zusammenzuführen. Das ist sehr umständlich und darum ist es wichtig, dass der Informationsmanagementprozess unter allen Beteiligten solche Situationen verhindert. UUIDs können auch verwendet werden um die Konsistenz und Integrität der Daten festzustellen. Wenn zwei Systeme die gleiche UUID für ein bestimmtes Feature verwenden, ist die Wahrscheinlichkeit sehr hoch, dass sie die Daten von der gleichen Quelle beziehen oder zumindest Prozesse existieren, die die Datenkonsistenz zwischen verschiedenen Systemen gewährleisten (EUROCONTROL & Federal Aviation Administration, 2011).

5.2 Zeitkonzept

Zeit spielt im Austausch von Informationen im Luftverkehr eine wichtige Rolle, da aktuelle Zustände von beteiligten Entitäten und zukünftige Zustandsänderungen ausgetauscht werden müssen. Das kann beispielsweise die Information sein, ob ein Flughafen geöffnet ist oder ob eine Landebahn benutzbar ist. Außerdem müssen aus juristischen Gründen auch vergangene Informationen aufbewahrt werden. Es ist daher notwendig, die zeitliche Entwicklung von Features möglichst genau und möglichst umfassend beschreiben zu können. Der AIXM-Standard versucht dies unter Berücksichtigung folgender Anforderungen (EUROCONTROL & Federal Aviation Administration, 2010) umzusetzen:

- **Vollständigkeit:** Alle zeitlichen Zustände müssen vollständig dargestellt werden können.
- **Minimalismus:** Es sollte nur die minimal notwendige Anzahl von Elementen verwendet werden.
- **Konsistenz:** Es sollen keine Elemente mit unterschiedlicher Bedeutung wiederverwendet werden.
- **Kontextunabhängigkeit:** Die Bedeutung von atomaren Elementen sollte kontextunabhängig sein und es sollen keine funktionalen Abhängigkeiten auf Ebene der Datenenkodierung zwischen diesen bestehen.

AIXM implementiert dafür ein umfassenden Zeitmodell, welches im Folgenden vorgestellt wird. Die Grundlage dieses Modells bildet das Konzept von TimeSlices. Das AIXM-Zeitmodell geht davon aus, dass sich jede Eigenschaft eines Features (außer die globale ID) mit der Zeit verändern kann. Ein TimeSlice dient dazu den Zustand eines Features zu einem bestimmten Zeitpunkt oder in einer bestimmten Zeitperiode zu beschreiben. Aus diesem Grund werden in AIXM alle Properties eines Features (außer die globale ID) in TimeSlice Objekte gekapselt. Die Grundlage dieses Zeitkonzepts wird in Abbildung 5.1 verdeutlicht.

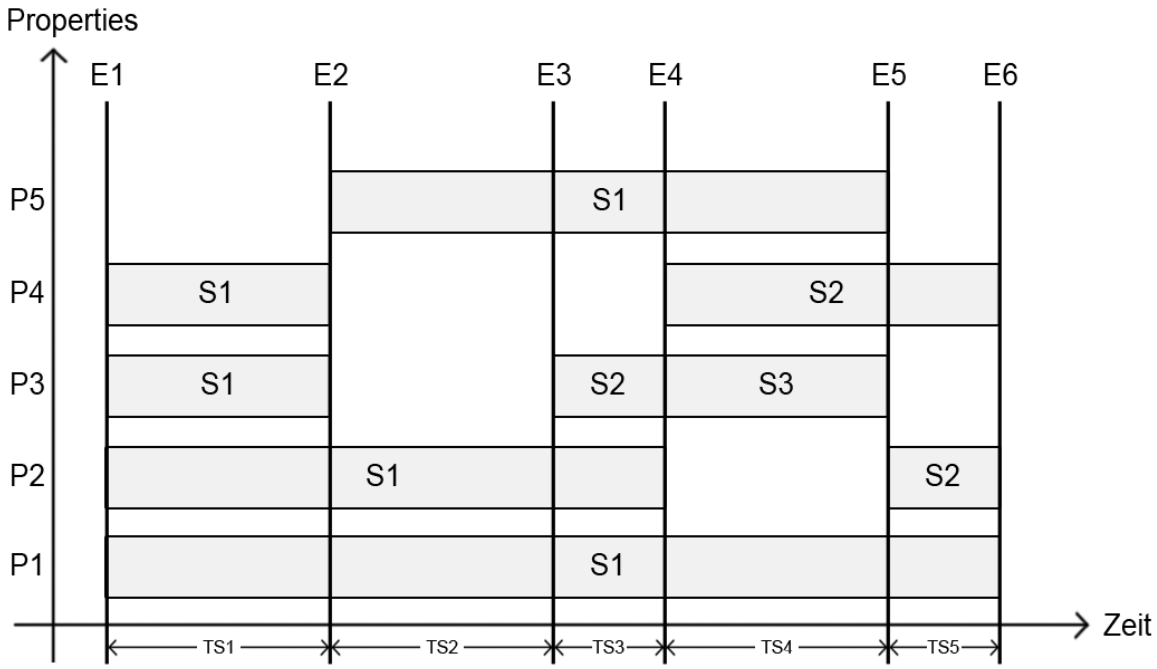


Abbildung 5.1: AIXM Zustände und Events

Abbildung 5.1 zeigt ein beliebiges Feature mit fünf Properties. P1 - P5 bezeichnen die Properties und S1 - S3 bezeichnet jeweils den aktuellen Zustand der Properties. Die Werte dieser Properties können sich jedoch mit der Zeit ändern und der Übergang des Zustands eines Properties in einen anderen Zustand wird durch Events gekennzeichnet. Events sind durch Kanten repräsentiert und in der Abbildung durch E1 - E6 bezeichnet. Die Zustände von Properties befinden sich zwischen den Kanten und jeder Zustand eines Properties zu einer bestimmten Zeit wird durch ein TimeSlice eingefangen. Die TimeSlices sind durch TS1 - TS5 gekennzeichnet. Dieses beinhaltet den gültigen Wert der Properties in einem bestimmten Zeitraum oder zu einem bestimmten Zeitpunkt. AIXM definiert verschiedene Arten von TimeSlices, wobei grundlegend zwischen zwei Arten von zeitlichen Änderungen unterschieden wird. Handelt es sich bei der Änderung von Zuständen um permanente Änderungen, spricht man von Baseline-Daten bzw. Baseline-Timeslices. Im Flugverkehr kommt es aber häufig vor, dass Features von temporären Events (z.B. Ausfall einer Flugnavigationshilfe, temporäre Flugverbotszone) betroffen sind. In diesem Fall spricht man von Tempdelta-Timeslices. Baseline- und Tempdelta-TimeSlices unterscheiden sich folgendermaßen:

- Die Baseline beschreibt den Zustand eines Features als Resultat einer dauerhaften Änderung. Sie beinhaltet immer das gesamte Property-Set eines Features. Das Baseline-TimeSlice T2 aus Abbildung 5.2 beinhaltet beispielsweise alle Werte für P1 bis P5.

- Ein Tempdelta-TimeSlice beschreibt den Zustand eines Features aufgrund eines temporären Ereignisses. Es beinhaltet nur die Properties eines Features, die in seinem Gültigkeitszeitraum tatsächlich geändert wurden. Es kann als Overlay auf eine Baseline betrachtet werden, das die gültigen Werte der Baseline für einen bestimmten Zeitraum überschreibt. Abbildung 5.2 zeigt ein Tempdelta-TimeSlice, das für einen bestimmten Zeitraum den Baseline-Wert für P1 überschreibt.

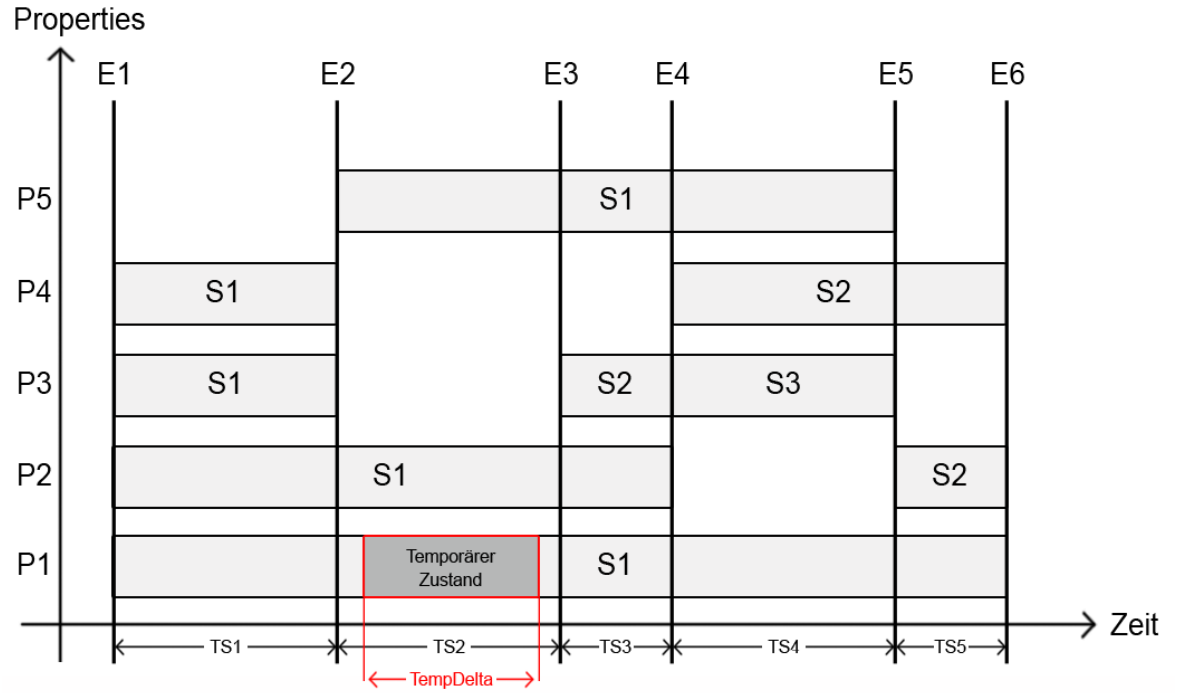


Abbildung 5.2: Unterscheidung AIXM Baseline und TempDelta

Da zu einem bestimmten Zeitpunkt mehrere Tempdelta-TimeSlices in Kombination mit der Baseline eines bestimmten Features gültig sein können, kann es schwierig sein, den aktuellen Zustand eines Features festzustellen. Für diesen Zweck führt AIXM Snapshot-TimeSlices ein. Snapshots entstehen durch das Zusammenführen von Baseline-Daten und allen darüberliegenden Tempdelta-TimeSlices, die zum Zeitpunkt des Snapshots gültig sind. Snapshots wären technisch gesehen nicht unbedingt notwendig, da sich die Anforderungen an das AIXM-Zeitmodell (Vollständigkeit, Minimalismus, Konsistenz und Kontextunabhängigkeit) mit Baseline- und Tempdelta-TimeSlices alleine erfüllen lassen. Sie stellen jedoch eine einfachere Möglichkeit zur Darstellung des gültigen Gesamtzustands eines Features zu einem bestimmten Zeitpunkt zur Verfügung.

Ein weiterer Komfortmechanismus des AIXM-Zeitmodells sind Permdelta-TimeSlices. Diese sind eine Mischung von Baseline- und Tempdelta-TimeSlices, da sie den geänderten Zustand eines Features als Resultat einer dauerhaften Änderung angeben, jedoch nur die tatsächlich geänderten Properties enthalten. Das hat den Vorteil, dass gezielt nur die tatsächlich geänderten Properties eines Features kommuniziert werden können. Das vereinfacht die Implementierung von Systemen, die nur an bestimmten Properties und deren Änderungen interessiert sind (z.B. Anwendungen zum Kartographieren). Permdelta-TimeSlices treten immer an der Kante zwischen zwei aufeinanderfolgenden Baselineabschnitten auf (EUROCONTROL & Federal Aviation Administration, 2010).

Kapitel 6

ICAO Meteorological Information Exchange Model

Das ICAO Meteorological Information Exchange Model (IWXXM) wird von der International Civil Aviation Organization (ICAO) und der World Meteorological Organization (WMO) entwickelt und gewartet. Es wurde erstmals 2013 von International Civil Aviation Organisation (2019) vorgestellt und stellt ein strukturiertes und standardisiertes Austauschformat für Operational Aeronautical Meteorological Data (OPMET) basierend auf XML dar. IWXXM soll langfristig dazu führen, von traditionellen OPMET-Daten in Form von Traditional Alphanumeric Code (TAC) wegzukommen. TAC wurde ursprünglich für menschliche Benutzer entworfen, IWXXM ist hingegen auf die maschinelle Verarbeitung ausgelegt. Dies soll gemeinsam mit der Verwendung von anderen standardisierten Datenmodellen wie AIXM, FIXM (Flight Information Exchange Model) oder METCE (Modèle pour l'Échange des informations sur le Temps, le Climat et l'Eau) zu einer Vereinfachung und Kostenreduktion bei der Entwicklung von neuen OPMET-Systemen führen. IWXXM liegt derzeit in der Version 3.0.0 vor und soll bis Ende 2020 zum internationalen Standard zum Austausch von OPMET-Daten werden. IWXXM basiert wie AIXM auf GML und stellt eine anwendungsspezifische Implementierung von GML für den Austausch von meteorologischen Informationen dar. Diese sind in Form von XML-Schemadefinitionen vorhanden. Die Verantwortung für die Entwicklung und Wartung ist zwischen WMO und ICAO aufgeteilt. Abbildung 6.1 zeigt eine Übersicht über IWXXM 3.0.0.

Abbildung 6.1 zeigt den Zusammenhang zwischen IWXXM, AIXM und GML. IWXXM verwendet diverse Komponenten von GML (z.B. geometrische Objekte, Koordinatensysteme, Beobachtungen) und AIXM (z.B. Features wie Flughäfen aus dem AIXM-Applikationsschema). Das zentrale Schema von IWXXM ist das IWXXM-Applikationsschema. Dieses Schema liegt im Verantwortungsbereich der ICAO und definiert die Darstellung meteorologischer Berichte (z.B. METAR, TAF, SIGMET). Diese Berichte wurden von der ICAO im *Annex 3 to the Convention on International Civil Aviation Meteorological Services For International Air Navigation* (International Civil Aviation Organisation, 2007) festgelegt und werden im Folgenden behandelt. Auf die Applikationsschemas, die von der WMO gewartet werden (z.B. METCE), wird in dieser Arbeit nicht weiter eingegangen.

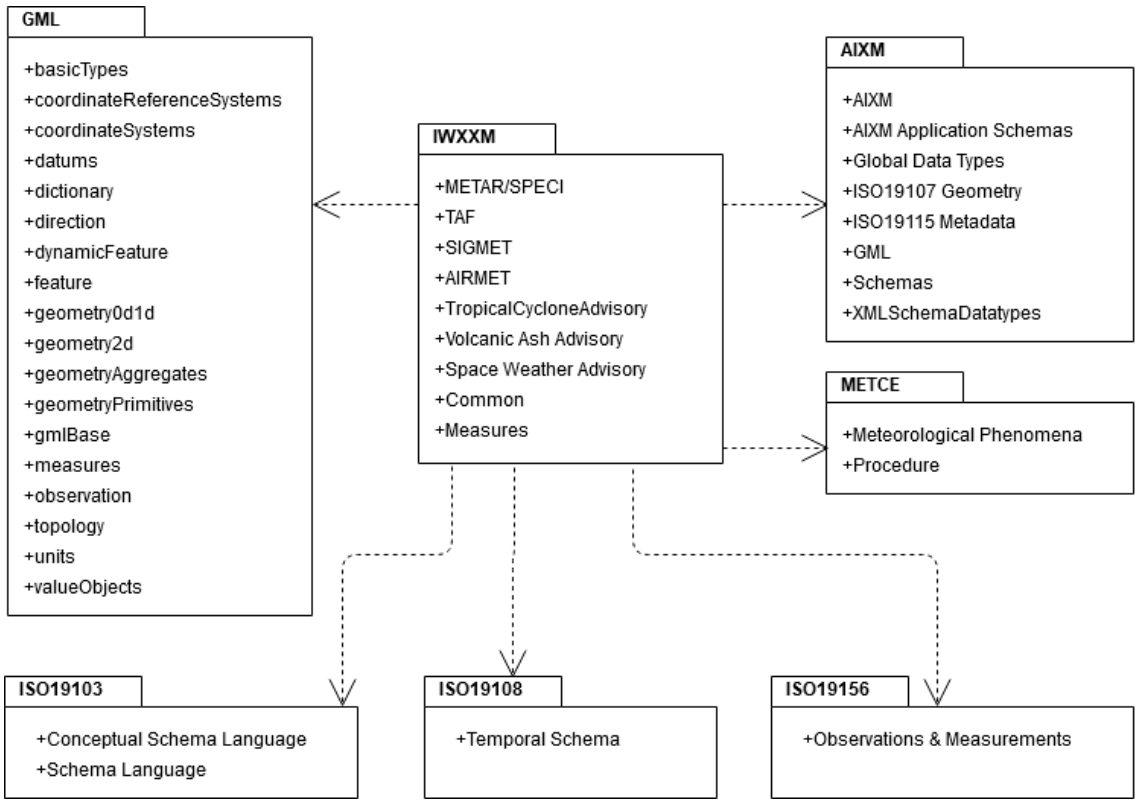


Abbildung 6.1: IWXXM 3.0.0 Package Übersicht

6.1 Meteorological Aerodrome Report und Special Weather Report

Meteorological Aerodrome Reports (METAR) und Special Weather Reports (SPECI) sind inhaltlich gleich aufgebaut, werden jedoch für unterschiedliche Zwecke erstellt. METARs sind routinemäßige Berichte über die meteorologischen Verhältnisse auf Flughäfen und werden je nach regionalen Flugsicherheitsbestimmungen stündlich oder halbstündlich erstellt. SPECIs sind Berichte, die je nach Bedarf herausgegeben werden um außergewöhnliche Beobachtungen bekanntzugeben. Die Notwendigkeit für die Erstellung eines SPECI-Berichts erfolgt nach bestimmten Kriterien, die mit der jeweils zuständigen meteorologischen Behörde, Flugsicherheitsbehörde und anderen relevanten beteiligten Institutionen festgelegt werden. METARs und SPECIs dienen hauptsächlich der Flugplanung und der Erstellung von Wetterberichten für Luftfahrzeuge. Sie beinhalten beispielsweise Informationen zu den aktuellen Sichtverhältnissen, aktuellen Windverhältnissen und Wettertrendvorhersagen an Flughäfen. (International Civil Aviation Organisation, 2012).

Die IWXXM-Repräsentation von METARs und SPECIs wird vom gemeinsamen Supertyp `iwxxm:MeteorologicalAerodromeObservationReportType` und in weiterer Folge vom allgemeinen Typ `iwxxm:ReportType` abgeleitet. Der `iwxxm:ReportType` ist in IWXXM die Basis für sämtliche Berichte. METARs und SPECIs sind Features im Sinne von GML, wie auch dynamische Features (vgl. Kapitel 4.4.1), Beobachtungen (vgl. Kapitel 4.7) oder Coverages (vgl. Kapitel 4.8). Listing 6.1 zeigt ein verkürztes Beispiel eines METAR/SPECI-Berichts.

Listing 6.1: METAR/SPECI-Bericht

```
<iwxxm:METAR gml:id="uuid.ce3b61b9-d360-4da1-83b9-67e668406869">
   <iwxxm:issueTime>
      <gml:TimeInstant gml:id="TI1">
         <gml:timePosition>2007-07-25T12:00:00Z</gml:timePosition>
      </gml:TimeInstant>
   </iwxxm:issueTime>
   <iwxxm:aerodrome>
      <aixm:AirportHeliport gml:id="AH1">
         <aixm:timeSlice>
            <aixm:AirportHeliportTimeSlice gml:id="AHTS1">
               <gml:validTime>...</gml:validTime>
               <aixm:interpretation>SNAPSHOT</aixm:interpretation>
               <aixm:designator>LKKV</aixm:designator>
               <aixm:name>KARLOVY VARY INTERNATIONAL</aixm:name>
               <aixm:locationIndicatorICAO>LKKV</aixm:locationIndicatorICAO>
            </aixm:AirportHeliportTimeSlice>
         </aixm:timeSlice>
      </aixm:AirportHeliport>
   </iwxxm:aerodrome>
   <iwxxm:observationTime>
      <gml:TimeInstant gml:id="TI2">...</gml:TimeInstant>
   </iwxxm:observationTime>
   <iwxxm:observation>
      <iwxxm:MeteorologicalAerodromeObservation cloudAndVisibilityOK="false"
      gml:id="MAO1">
         <iwxxm:airTemperature uom="Cel">27</iwxxm:airTemperature>
         <iwxxm:dewpointTemperature uom="Cel">10</iwxxm:dewpointTemperature>
         <iwxxm:qnh uom="hPa">1010</iwxxm:qnh>
         <iwxxm:surfaceWind>
            <iwxxm:AerodromeSurfaceWind variableWindDirection="false">
               <iwxxm:meanWindDirection uom="deg">210</iwxxm:meanWindDirection>
               <iwxxm:meanWindSpeed uom="m/s">2.6</iwxxm:meanWindSpeed>
            </iwxxm:AerodromeSurfaceWind>
         </iwxxm:surfaceWind>
         <iwxxm:presentWeather xlink:href="http://codes.wmo.int/.../VCSH"/>
         <iwxxm:recentWeather xlink:href="http://codes.wmo.int/.../TS"/>
         ...
      </iwxxm:MeteorologicalAerodromeObservation>
   </iwxxm:observation>
</iwxxm:METAR>
```

Dieses Beispiel zeigt einen METAR-Bericht für den Flughafen Karlsbad in Tschechien. Der Bericht enthält die Beobachtungszeit (`iwxxm:observationTime`) und die Veröffentlichungszeit (`iwxxm:issueTime`) des Berichts. Beide Zeitangaben werden auf Basis von GML dargestellt (vgl. Kapitel 4.4). Die Angabe des betroffenen Flughafens erfolgt in Form eines Snapshot-TimeSlices und wurde aus dem AIXM-Zeitmodell übernommen (vgl. Kapitel 5.2). Die Angabe der meteorologischen Beobachtungen erfolgt in einem `iwxxm:MeteorologicalAerodoromeObservation` Element und beinhaltet Angaben zu Lufttemperatur, Kondensationstemperatur, Luftdruck, Windverhältnissen

und aktuellen und vergangenen Wetterkonditionen. Die Wetterkondtionen werden durch Codes angegeben, die von der WMO (World Meteorological Organization, n. d.) definiert wurden (VCHS = Shower(s) in the vicinity, TS = Thunderstorm).

6.2 Terminal Aerodrome Forecast

Terminal Aerodrome Forecasts (TAF) sind routinemäßige Berichte zu Wettervorhersagen an Flughäfen. Sie sind in einem bestimmten Zeitraum gültig und können gegebenenfalls auch modifiziert werden. Sie ähneln inhaltlich den METAR/SPECI Wetterbeobachtungen, sind aber detaillierter. METARs und TAFs werden regelmäßig von Flughäfen erstellt, wobei TAFs nicht zwingendermaßen von jedem Flughafen generiert werden, der auch METARs generiert. Sollte ein neuer TAF Bericht für den gleichen Zeitraum und gleichen Flughafen herausgegeben werden, ersetzt dieser sämtliche vorhergehende TAFs.

Listing 6.2: TAF-Bericht

```
<iwxxm:TAF gml:id="uuid.d6a85870-f32e-4ea8-8502-c7d9be7e0144">
   <iwxxm:issueTime>...</iwxxm:issueTime>
   <iwxxm:aerodrome>
      <aixm:AirportHeliport gml:id="AH1">...</aixm:AirportHeliport>
   </iwxxm:aerodrome>
   <iwxxm:validPeriod>
      <gml:TimePeriod gml:id="TP1">
         <gml:beginPosition>2012-08-16T00:00:00Z</gml:beginPosition>
         <gml:endPosition>2012-08-16T18:00:00Z</gml:endPosition>
      </gml:TimePeriod>
   </iwxxm:validPeriod>
   <iwxxm:baseForecast>
      <iwxxm:MeteorologicalAerodromeForecast gml:id="MAF1"
      cloudAndVisibilityOK="false">
         <iwxxm:phenomenonTime xlink:href="PT1"/>
         <iwxxm:prevailingVisibility uom="m">9000</iwxxm:prevailingVisibility>
         <iwxxm:cloud>
            <iwxxm:AerodromeCloudForecast gml:id="ACF1">
               ...
            </iwxxm:AerodromeCloudForecast>
         </iwxxm:cloud>
      </iwxxm:MeteorologicalAerodromeForecast>
   </iwxxm:baseForecast>
   <iwxxm:changeForecast>
      <iwxxm:MeteorologicalAerodromeForecast gml:id="MAF2"
      changeIndicator="BECOMING" cloudAndVisibilityOK="false">
         ...
      </iwxxm:MeteorologicalAerodromeForecast>
   </iwxxm:changeForecast>
</iwxxm:TAF>
```

TAFs bestehen inhaltlich aus Basis- und Änderungsvorhersagen. Basisvorhersagen enthalten sämtliche Vorhersagen zu den vorherrschenden Wetterbedingungen, wobei Änderungsvorhersagen die Basisvorhersagen modifizieren können. Technisch gesehen gibt es keine Beschränkung darüber, wie viele Änderungsvorhersagen ein TAF beinhalten kann. Die WMO empfiehlt jedoch die Anzahl von Änderungsvorhersagen möglichst gering zu halten und nicht mehr als fünf Vorhersagen pro TAF herauszugeben (International Civil Aviation Organisation, 2012). Listing 6.2 zeigt ein verkürztes Beispiel eines TAF-Berichts. Dieses Beispiel beinhaltet ebenfalls die Veröffentlichungszeit und den betroffenen Flughafen. Zusätzlich enthalten TAFs den Gültigkeitszeitraum für die Wettervorhersage. Dieser erstreckt sich von 2012-08-16 00:00 bis 2012-08-16 18:00. Die Basisvorhersage enthält Angaben zu den vorherrschenden Sichtverhältnissen (9000m) und zu vorhandenen Wolkenformationen. Zusätzlich zur Basisvorhersage beinhaltet der TAF-Bericht auch eine Änderungsvorhersage. Diese kann beispielsweise eine Änderung der Sichtverhältnisse oder eine Positionsänderung von Wolkenformationen enthalten.

6.3 Significant Meteorological Phenomena

Significant Meteorological Phenomena (SIGMET) sind Berichte über Wetterphänomene, die die Sicherheit des Flugverkehrs auf einer bestimmten Route beeinflussen können. SIGMETs sind die Basisklasse für die Darstellung von signifikanten Wetterverhältnissen und können Phänomene wie Stürme, Turbulenzen, Zyklone oder den Ausstoß von Vulkanasche beschreiben. Tropische Zyklone und Vulkanasche sind Spezialfälle und werden in Kapitel 6.5 und Kapitel 6.6 genauer beschrieben. SIGMETs können aktuell beobachtete Wetterphänomene oder Vorhersagen zu potenziell gefährlichen Wetterphänomenen beinhalten. Sie betreffen immer einen bestimmten geografischen Bereich (International Civil Aviation Organisation, 2012). Zur Definition dieses Bereichs werden Positionsangaben und geometrische Objekte von GML verwendet (vgl. Kapitel 4.2). Listing 6.3 zeigt ein verkürztes Beispiel eines SIGMET-Berichts.

SIGMETs beinhalten ebenfalls die Veröffentlichungszeit (`iwxxm:issueTime`) und den Gültigkeitszeitraum des Berichts (`iwxxm:validPeriod`). Zusätzlich wird das betroffene Flugverkehrsservice (`iwxxm:issuingAirTrafficServicesUnit`), die zuständige meteorologische Überwachungsbehörde (iwxxm:originatingMeteorologicalWatchOffice) und die betroffene Flugverkehrszone (`iwxxm:issuingAirTrafficServicesRegion`) angegeben. Außerdem wird die Art des Phänomens, auf das sich der Bericht bezieht, angegeben. Diese wird wiederum in Form von WMO-Codes (World Meteorological Organization, n. d.) (z.B. OBSC_TS = Obscured Thunderstorm) definiert.

SIGMETs können auch mehrere relevante Phänomene beinhalten. Diese werden in Form von Sammlungen angegeben (`iwxxm:SIGMETEvolvingConditionCollection`). In dem Beispiel handelt es sich um eine Vorhersage (`timeIndicator=FORECAST`) und diese gibt an, dass sich das Phänomen in nächster Zeit abschwächen wird (`intensityChange=WEAKEN`). Für die Beschreibung des betroffenen geografischen Raumes werden Elemente von AIXM und GML verwendet. Dieser wird durch einen Luftraum angegeben, der durch ein vertikales Höhenlimit und die horizontale Projektion über eine bestimmte Fläche definiert ist. Der horizontale Bereich wird durch ein `aixm:Surface` definiert (vgl. Kapitel 4.2.2).

Listing 6.3: SIGMET-Bericht

```
<iwxxm:SIGMET gml:id="uuid.8debfaca-7116-4b32-9378-b472ca64e823">
   <iwxxm:issueTime>...</iwxxm:issueTime>
   <iwxxm:issuingAirTrafficServicesUnit>
      ...
   </iwxxm:issuingAirTrafficServicesUnit>
   <iwxxm:originatingMeteorologicalWatchOffice>
      ...
   </iwxxm:originatingMeteorologicalWatchOffice>
   <iwxxm:issuingAirTrafficServicesRegion>
      ...
   </iwxxm:issuingAirTrafficServicesRegion>
   <iwxxm:sequenceNumber>2</iwxxm:sequenceNumber>
   <iwxxm:validPeriod>...</iwxxm:validPeriod>
   <iwxxm:phenomenon xlink:href="http://codes.wmo.int/.../OBSC_TS"/>
   <iwxxm:analysis>
      <iwxxm:SIGMETEvolvingConditionCollection gml:id="SECC1"
      timeIndicator="FORECAST">
         <iwxxm:member>
            <iwxxm:SIGMETEvolvingCondition gml:id="SEC1" intensityChange="WEAKEN">
               <iwxxm:geometry>
                  <aixm:AirspaceVolume gml:id="AV1">
                     <aixm:upperLimit uom="FL">390</aixm:upperLimit>
                     <aixm:upperLimitReference>STD</aixm:upperLimitReference>
                     <aixm:horizontalProjection>
                        <aixm:Surface gml:id="S1" axisLabels="Lat Long">
                        <!-- PolygonPatch mit LinearRing -->
                        </aixm:Surface>
                     </aixm:horizontalProjection>
                  </aixm:AirspaceVolume>
               </iwxxm:geometry>
               <iwxxm:directionOfMotion uom="deg">90</iwxxm:directionOfMotion>
               <iwxxm:speedOfMotion uom="[kn_i]">20</iwxxm:speedOfMotion>
            </iwxxm:SIGMETEvolvingCondition>
         </iwxxm:member>
      </iwxxm:SIGMETEvolvingConditionCollection>
   </iwxxm:analysis>
</iwxxm:SIGMET>
```

6.4 Airman's Meteorological Information

Airman's Meteorological Information (AIRMET) sind ebenfalls Berichte zu Wetterphänomenen, die die Flugsicherheit beeinflussen können. Sie sind strukturell gleich aufgebaut wie SIGMETs. SIGMETs beinhalten jedoch nur Wetterphänomene, die die Flugsicherheit massiv beeinflussen. AIRMETs können auch mildere Phänomene enthalten und sie können aus diesem Grund auch als spezielle Wetterberichte betrachtet werden, die für den Flugverkehr konzipiert wurden. AIRMETs enthalten beispielsweise Informationen über schlechte Wetterverhältnisse, Sichtverhältnisse, Turbulenzen oder Vereisungen bei niedrigen Temperaturen (International Civil Aviation Organisation, 2012). AIRMETs sind technisch gleich aufgebaut und beinhalten dieselben Informationen wie SIGMETs (z.B. Veröffentlichungszeit, Gültigkeitszeitraum, Flugverkehrsservice). Die Art der berichteten Phänomene unterscheidet sich jedoch von SIGMETs und wird von der WMO in einer eigenen Codeliste (World Meteorological Organization, n. d.) gepflegt(z.B. ISOL_TS = Isolated Thunderstorm).

6.5 Tropical Cyclone Advisory

Tropical Cyclone Advisory (TCA) existiert in IWXXM sowohl als spezielle Form (Subklasse) von SIGMETs, als auch als eigenständiger Bericht. Als SIGMET-Subklasse haben TCAs dieselbe Struktur wie SIGMETs. Als eigenständige Berichte sind sie jedoch strukturell anders aufgebaut. Beide Varianten stellen eigene Feature-Typen (`iwxxm:TropicalCycloneSIGMET`, `iwxxm:TropicalCyclone`) dar. TCAs sind spezielle Berichte, die das Auftreten von tropischen Wirbelstürmen signalisieren, welche die Flugsicherheit massiv beeinflussen können.

Listing 6.4: TCA-Bericht

```
<iwxxm:TropicalCycloneAdvisory gml:id="uuid.217b27d6-fbc0-499b-b58e-4ee062c86cd4">
   <iwxxm:issueTime>...</iwxxm:issueTime>
   <iwxxm:issuingTropicalCycloneAdvisoryCentre>
      ...
   </iwxxm:issuingTropicalCycloneAdvisoryCentre>
   <iwxxm:tropicalCycloneName>
      <metce:TropicalCyclone gml:id="TC1">
         <metce:name>GLORIA</metce:name>
      </metce:TropicalCyclone>
   </iwxxm:tropicalCycloneName>
   <iwxxm:advisoryNumber>2004/13</iwxxm:advisoryNumber>
   <iwxxm:observation>
      <iwxxm:TropicalCycloneObservedConditions gml:id="TCOC1">
         <iwxxm:phenomenonTime>
            <gml:TimeInstant gml:id="TI1">
               <gml:timePosition>2004-09-25T16:00:00Z</gml:timePosition>
            </gml:TimeInstant>
         </iwxxm:phenomenonTime>
         <iwxxm:tropicalCyclonePosition>
            <gml:Point gml:id="P1" axisLabels="Lat Long">
               <gml:pos>27.1 -73.1</gml:pos>
            </gml:Point>
         </iwxxm:tropicalCyclonePosition>
         <iwxxm:cumulonimbusCloudLocation>
            <aixm:AirspaceVolume gml:id="AV1">
            <!--Beinhaltet Positionsangaben auf GML basierend-->
            </aixm:AirspaceVolume>
         </iwxxm:cumulonimbusCloudLocation>
         <iwxxm:movement>MOVING</iwxxm:movement>
         ...
      </iwxxm:TropicalCycloneObservedConditions>
   </iwxxm:observation>
   <iwxxm:forecast>
      <iwxxm:TropicalCycloneForecastConditions gml:id="TCFC1">
         <!--Nach 6h-->
      </iwxxm:TropicalCycloneForecastConditions>
   </iwxxm:forecast>
   <iwxxm:forecast>...</iwxxm:forecast><!--Nach 12h, 18h, 24h-->
</iwxxm:TropicalCycloneAdvisory>
```

Listing 6.4 zeigt ein verkürztes TCA-Beispiel. Für diese Wetterphänomene existieren spezielle Beobachtungsstationen (Tropical Cyclone Advisory Centres), die TCAs herausgegeben (International Civil Aviation Organisation, 2012). In SIGMETS stellen diese nur eine spezielle Form von `iwxxm:EvolvingCondition` dar, deshalb wird im Folgenden auf TACs als eigenständige Berichte eingegangen. TCAs enthalten neben der Veröffentlichungszeit auch einen Namen (z.B. Gloria) und eine Advisory Nummer (`iwxxm:advisoryNumber`), die das Jahr des Auftretens angibt. Der Rest des Berichts besteht aus der aktuell beobachteten Lage des Wirbelsturms (`iwxxm:TropicalCycloneObservedCondition`) und eventuell aus einer oder mehreren Vorhersagen (`iwxxm:TropicalCycloneForecastCondition`) zur Entwicklung des Sturms. Diese werden bei Zyklonwarnungen im Normalfall für 24 Stunden im Voraus in 6-Stunden Intervallen angegeben. Die aktuelle Beobachtung beinhaltet auch Angaben zur Bewegung des Sturms (z.B. Geschwindigkeit, Richtung). Optional kann angegeben werden, wann der nächste Bericht voraussichtlich veröffentlicht wird.

6.6 Volcanic Ash Advisory

Volcanic Ash Advisory (VAA) Berichte existieren ebenfalls als spezielle Form von SIGMETs und als eigenständige Berichte. Die Unterscheidung erfolgt ebenfalls durch eigenständige Feature-Typen. Sie werden von Volcanic Ash Advisory Zentren (`iwxxm:issuingVolcanicAshAdvisoryCentre`) ausgegeben und warnen über das Vorkommen von Vulkanascheausstößen, die die Flugsicherheit beeinflussen. Sie sind strukturell ähnlich wie TCAs aufgebaut, da sie auch die derzeitige Lage (`iwxxm:VolcanicAshObservedOrEstimatedCondition`) und eventuell mehrere Vorhersagen (`iwxxm:VolcanicAshForecastCondition`) beinhalten können (International Civil Aviation Organisation, 2012). Zusätzlich enthält der Bericht den betroffenen Vulkan (`metce:EruptingVolcano`), der mit Hilfe des METCE-Anwendungsschemas (World Meteorological Organization, 2014) definiert wird. Zusätzlich sind noch Informationen zum betroffenen Land (`iwxxm:stateOrRegion`), zur Höhe des Vulkans (`iwxxm:summitElevation`), zur Informationsquelle (`iwxxm:informationSource`) und weitere Details zum Vulkanausbruch (`iwxxm:eruptionDetails`) enthalten. Eine Liste von Farbcodes, die sich auf das Risiko und die Schwere des Vulkanausbruchs beziehen, wurde ebenfalls von der WMO (World Meteorological Organization, n. d.) definiert (z.B. RED = Ausbruch steht unmittelbar bevor und es sind signifikante Mengen von Vulkanasche in der Atmosphäre zu erwarten).

Kapitel 7

Flora-2/ErgoAI

Flora-2 ist eine logische Programmiersprache und objektorientierte Reasoning Engine, die auf Basis von F-Logic entwickelt wurde. Anwendungsgebiete von Flora-2 sind die Integration von Informationen, Management von Ontologien oder die Verhaltenssteuerung von künstlicher Intelligenz. Die Sprache wurde unter Berücksichtigung von Flexibilität und Erweiterbarkeit entwickelt und besitzt einen eigenen Compiler, der F-Logic, HiLog und Transaction Logic Fragmente in eine einheitliche Sprache zusammenfasst und in Prolog-Code übersetzt. Flora-2 ist für Windows, Mac OS, Linux und andere Unix-basierte Betriebssysteme verfügbar (Yang et al., 2008).

ErgoAI ist eine kommerzielle Umsetzung von Flora-2, die die Kernfunktionalität von Flora-2 implementiert und erweitert. ErgoAI besteht aus dem ErgoAI Studio und der ErgoAI Reasoning Engine. Das ErgoAI Studio bietet im Vergleich zur Shell-Oberfläche von Flora-2 den Vorteil einer grafischen Entwicklungsumgebung. Außerdem stellt ErgoAI diverse Schnittstellen für den Zugriff auf externe Datenquellen zur Verfügung, unter anderem SQL, RDF, OWL, Python, Java und Webservices (Kifer, 2018a). Flora-2 bietet dagegen nur eine sehr eingeschränkte Schnittstellenimplementierung (vgl. Kapitel 7.3) (Nivalis, 2018). Da die Funktionalität von Flora-2 für diese Arbeit nicht ausreichen würde, wird ErgoAI verwendet. Eine akademische Lizenz für ErgoAI ist auf Anfrage beim Hersteller Coherent Knowledge kostenlos verfügbar. Im Folgenden werden die wichtigsten Konzepte von Flora-2 und ErgoAI erläutert.

7.1 Grundbegriffe

Dieses Kapitel behandelt die wichtigsten Grundbegriffe für die Erstellung von Wissensdatenbanken und die Formulierung von Abfragen in ErgoAI. Dazu gehören Fakten, Regeln, Abfragen und Direktiven. Deren syntaktische Umsetzung in ErgoAI wird anhand von Beispielen dargestellt.

7.1.1 Fakten

Fakten (Facts) sind grundlegende Aussagen, die ohne weitere Prüfung immer als wahr befunden werden. Sie können konzeptuell wie eine Regel ohne Voraussetzungen gesehen werden. Syntaktisch sehen sie ähnlich wie Abfragen (vgl. Kapitel 7.1.3) aus. Fakten werden auch als Literale bezeichnet und folgen der Form `predicate(argument)`. Listing 7.1 zeigt ein Beispiel für Fakten in ErgoAI (Kifer, 2018b).

Listing 7.1: ErgoAI Fakt

```
1 Aircraft(AirbusA380).
2 Weight(AirbusA380,t560).
```

Der Fakt `Aircraft(AirbusA380)` beschreibt den Sachverhalt, dass der Airbus A380 ein Flugzeug ist und besteht nur aus einem Argument. Fakten können aber auch aus mehreren Argumenten bestehen. Der Fakt `Weight(AirbusA380,t560)` gibt an, dass der Airbus A380 ein Gewicht von 560 Tonnen hat.

7.1.2 Regeln

Regeln sind das Kernelement von Wissensdatenbanken, da sie zur Ableitung von Wissen verwendet werden können. Sie unterscheiden dadurch Wissensdatenbanken von normalen Datenbanken und können beispielsweise für den Aufbau von Klassenhierarchien verwendet werden. Regeln bestehen aus einem Regelkopf (Rule Head) und einem Regelrumpf (Rule Body). Sie folgen der Form `ruleHead :- ruleBody`. Der Regelkopf wird durch den Regelrumpf impliziert. Das bedeutet, dass der Regelkopf dann wahr ist, wenn der Regelrumpf wahr ist. Der Regelrumpf kann auch aus mehreren Teilbedingungen zusammengesetzt sein. Listing 7.2 zeigt Beispiele für einfache und zusammengesetzte Regeln (Kifer, 2018b).

Listing 7.2: ErgoAI Regel

```
1 Aircraft(?X) :- hasWings(?X).
2 Airport(?X):- hasRunway(?X),hasTower(?X).
```

Die Regel `Aircraft(?X) :- hasWings(?X)` gibt an, dass X ein Flugzeug ist, wenn X Tragflächen besitzt. `?X` verweist hier auf die Verwendung einer Variable (vgl. Kapitel 7.1.4). Die Regel `Airport(?X):- hasRunway(?X),hasTower(?X)` gibt an, dass X dann ein Flughafen ist, wenn X über eine Landebahn und einen Tower verfügt. Regeln können außerdem mit Deskriptoren versehen werden. Deskriptoren können dazu verwendet werden, um Regeln einzigartige IDs zuzuweisen. Regeln können dann über diese IDs referenziert werden, was beispielsweise für die Vergabe von Prioritäten zur Konfliktauflösung verwendet werden kann. Deskriptoren sind optional und folgen der Form `@{ruleDescriptor}`.

7.1.3 Abfragen

Abfragen (Queries) werden dazu verwendet um Wissen aus der Wissensdatenbank abzufragen. Sie sind syntaktisch fast identisch zu Regelrümpfen aufgebaut. Das hat zur Folge, dass Regelrümpfe auch als Abfragen verwendet werden können. Abfragen können direkt in ErgoAI-Dateien eingebettet werden und werden beim Laden der Dateien in die ErgoAI Reasoning Engine automatisch ausgeführt. Sie können aber auch interaktiv über die Shell-Konsole oder über das ErgoAI Studio abgesetzt werden. Eingebettete Abfragen werden syntaktisch durch die Verwendung des Präfixes `-?` gekennzeichnet. Listing 7.3 zeigt eine eingebettete Abfrage (Kifer, 2018b).

Listing 7.3: ErgoAI-Abfrage

```
1 Airport(Heathrow).
2 hasTower(Gatwick).
3 hasRunway(Gatwick).
4 Heliport(Stansted).
5 hasTower(Stansted).
6 Airport(?X):- hasRunway(?X),hasTower(?X).
7 ?-Airport(?X).
```

Dieses Beispiel zeigt eine Wissensdatenbank mit Fakten und Regeln über die Definition von Flughäfen und Hubschrauberlandeplätzen. Die Abfrage `?-Airport(?X)` ermittelt alle Flughäfen aus der Wissensdatenbank. Der Fakt `Airport(Heathrow)` gibt an, dass Heathrow ein Flughafen ist. Die Regel `Airport(?X):- hasRunway(?X),hasTower(?X)` gibt an, dass alles ein Flughafen ist, was über eine Landebahn und einen Tower verfügt. Diese Regel wird für Gatwick durch die Fakten `hasTower(Gatwick)` und `hasRunway(Gatwick)` erfüllt. Das Ergebnis der Abfrage ist somit Heathrow und Gatwick. Stansted qualifiziert sich nicht als Flughafen, da es weder direkt über einen Fakt als solcher deklariert wurde, noch über Regeln die Bedingungen für einen Flughafen erfüllt.

7.1.4 Variablen

Variablen sind Ausdrücke, die mit dem ? Präfix beginnen. Sie können in Abfragen und für die Konstruktion von Regeln verwendet werden. Variablen mit gleichem Namen innerhalb von Regeln und Abfragen stehen auch für dieselbe Variable und sind lokal an die jeweilige Abfrage oder Regel gebunden. Das bedeutet umgekehrt, dass Variablen mit gleichem Namen in unterschiedlichen Regeln oder Abfragen auch für unterschiedliche Dinge stehen. Variablen können auch mit dem Präfix ?_ beginnen und werden dann als stille Variablen bezeichnet. Stille Variablen werden dann verwendet, wenn die Bindung der Variable für das Abfrageergebnis nicht relevant ist. Konstanten sind in ErgoAI alle sonstigen Zeichenfolgen (abgesehen von vordefinierten Schlüsselwörtern und Befehlen) ohne ? Präfix, die in Regeln oder Abfragen verwendet werden.

7.1.5 Direktiven

Direktiven können verwendet werden um das Verhalten der ErgoAI Reasoning Engine zu steuern. Sie können in Compiler- und Laufzeitdirektiven unterschieden werden. Compilerdirektiven können in ErgoAI-Dateien angegeben werden und wirken sich auf den Kompilierprozess aus. Sie folgen syntaktisch der Form `:- directiveName{arguments}`, wobei nicht alle Compilerdirektiven Argumente besitzen müssen (Kifer, 2018b). Listing 7.4 zeigt ein Beispiel für eine Compilerdirektive zur Definition des Standardverhaltens für die Annulierung (Defeasability) von widersprüchlichen Regeln. Compilerdirektiven müssen am Beginn eines ErgoAI-Dokuments stehen, vor den ersten Fakten oder Regeln. Ansonsten kann der ErgoAI-Compiler diese nicht berücksichtigen.

Listing 7.4: ErgoAI-Compilerdirektive

```
1 :- default_is_defeasible
2 :- default_is_strict
```

Laufzeitdirektiven werden syntaktisch wie Abfragen behandelt und hauptsächlich dazu verwendet, um das Verhalten der ErgoAI-Shell zu steuern (Kifer, 2018b). Einige Compilerdirektiven können

auch als Laufzeitdirektiven verwendet werden. Da diese syntaktisch wie Abfragen behandelt werden, können Laufzeitdirektiven ebenfalls in ErgoAI-Dateien eingebettet werden. In diesem Fall werden sie ebenfalls durch den ? Präfix gekennzeichnet. Laufzeitdirektiven können aber auch vom Benutzer direkt in die ErgoAI-Shell oder über das ErgoAI Studio eingegeben werden. Listing 7.5 zeigt ein Beispiel einer Laufzeitdirektive zur Definition des Verhaltens bei der Auflösung von Klassenhierarchien. Das geschieht durch die Angabe des `setsemantics` Befehls, was in ErgoAI nur zur Laufzeit möglich ist (Kifer, 2018b).

Listing 7.5: ErgoAI-Laufzeitdirektive

```
1 ?-setsemantics{subclassing=strict}
2 ?-setsemantics{subclassing=nonstrict}
```

Bei strikten Klassenhierarchien liefert der ErgoAI Reasoner zur Laufzeit einen Fehler bei zyklischen Hierarchien. Abfragen auf solche Hierarchien würden `false` zurückliefern. Ist die Auflösung von Klassenhierarchien nicht strikt, werden auch Abfragen auf zyklische Klassenhierarchien mit `true` beantwortet und keine Fehler ausgegeben.

7.2 Strukturierung von Wissensdatenbanken

Für kleine Wissensdatenbanken ist es durchaus praktikabel, sämtliches Wissen in einer Datei abzulegen. Große Wissensdatenbanken können aber schnell unübersichtlich werden. Dafür unterstützt ErgoAI die Aufteilung von Wissensdatenbanken in verschiedene Dateien und Module. Das ist beispielsweise nützlich um das Parsing von Daten aus unterschiedlichen Datenquellen zu vereinfachen. ErgoAI bietet verschiedene Möglichkeiten zur Strukturierung von Wissensdatenbanken, welche im Folgenden erläutert werden.

ErgoAI arbeitet mit einem eigenen Dateityp, der auf *.ergo endet. Eine Möglichkeit ist die Aufteilung in mehrere physische ErgoAI-Dateien. Das hat den Vorteil, dass jede Datei einen eigenen Namespace bekommt und bedeutet, dass Prädikate mit gleichem Namen in unterschiedlichen Dateien verwendet werden können, ohne dass es dabei zu Konflikten kommt (Kifer, 2018b). Ein weiterer Vorteil ist die Wiederverwendbarkeit von Dateien (z.B. Zugriff auf dieselbe Flugverkehrsnachricht aus verschiedenen Modulen). Dadurch kann eine inhaltliche Strukturierung des Wissens erfolgen (z.B. jede Nachricht ist eine eigene Datei). Es erleichtert außerdem die Arbeitsteilung bei Projekten mit vielen Beteiligten.

Mehrere physische Dateien können über den `\#include` Befehl zu einer einheitlichen Datei kombiniert werden. Die Dateien können dann in die ErgoAI Reasoning Engine geladen werden. Das Laden einer Datei in ein Modul überschreibt jedoch sämtliche bereits existierenden Daten (z.B. Fakten, Regeln). Falls das nicht gewünscht ist, gibt es auch eine kumulative Methode (add Befehl) zum Hinzufügen von Daten. Für das Zusammenfügen von Dateien verwendet ErgoAI einen eigenen Preprocessor, der zur Kompilierzeit die einzelnen Dateien kombiniert. ErgoAI überprüft beim Laden automatisch, ob es sich bei den Dateien um neuere Versionen handelt und rekompiliert diese im Bedarfsfall. ErgoAI kann sowohl mit Unix-Pfaden, als auch mit Windows-Pfaden umgehen. Es wird jedoch empfohlen, dass Unix-Pfade verwendet werden, da diese unabhängig vom Betriebssystem sind. Listing 7.6 zeigt ein Beispiel für eine zusammengesetzte Datei.

Listing 7.6: ErgoAI zusammengesetzte Datei

```
#include "../testdata/AIXM_MESSAGE_1.ergo"
#include "../testdata/AIXM_MESSAGE_2.ergo"
#include "../testdata/IWXXM_MESSAGE_1.ergo"

//Sonstige Statements
//Regeln, Fakten, eingebettete Queries, etc.
```

Dateien können auch mit dem add{file} bzw. add{file > module} Befehl hinzugefügt werden. Der wesentliche Unterschied zum Laden von Dateien ist, dass beim Hinzufügen die bereits vorhandenen Daten nicht überschrieben werden. Das Laden von Dateien ist beispielsweise dann sinnvoll, wenn ein Modul neu initialisiert werden soll. Das Hinzufügen von Dateien ist dann sinnvoll, wenn Wissen aus verschiedenen Quellen zusammengeführt werden soll. Geladene Regeln sind in den meisten Fällen performanter als hinzugefügte Regeln und aus diesem Grund sollte wenn möglich immer das Laden von Dateien dem Hinzufügen bevorzugt werden. Geladene und hinzugefügte Dateien können wiederum andere Dateien über den #include Befehl enthalten (Kifer, 2018b).

Wir haben beim Laden und Hinzufügen von Dateien schon mehrfach von Modulen gesprochen. Module stellen in ErgoAI eine Abstraktion zur logischen Gliederung von Wissensdatenbanken dar und wurden mit dem Hintergrund der Wiederverwendbarkeit eingeführt. ErgoAI beinhaltet standardmäßig ein Main-Modul. Dieses wird immer dann verwendet, wenn kein explizites Modul angegeben wird. ErgoAI hat bereits diverse Systemmodule und Module für den Aufruf von Prolog-Befehlen integriert. Es bietet aber auch die Möglichkeit zur Definition von benutzerdefinierten Modulen. Diese Modularten werden im Folgenden näher erläutert.

7.2.1 Benutzerdefinierte Module

Benutzerdefinierte Module dienen zur logischen Gliederung von Wissensdatenbanken und bestehen formal gesehen aus einem Namen (alphanumerischen Zeichen) und ihrem Inhalt (z.B. Fakten, Regeln). Dateien können in diese Module geladen oder hinzugefügt werden, wobei ErgoAI immer nur auf Modulebene arbeitet. Das bedeutet, dass ErgoAI nur die Modulnamen für den Aufruf kennt, jedoch nicht die dahinterliegenden Dateien, aus denen ein Modul besteht. Module können außerdem private Methoden zur Daten- und Funktionskapselung beinhalten. Dies ist neben der logischen Gliederung ein weiterer großer Vorteil der Verwendung von Modulen.

7.2.2 Prolog Module

Prolog-Module stellen in ErgoAI eine Abstraktion für den Aufruf von Prolog-Befehlen zur Verfügung. Diese Module sind statisch und eng mit ihrem Code verknüpft. Prolog-Module im ErgoAI-Sinn sind jedoch nicht mit Modulen in Prolog zu verwechseln. Sie stellen vielmehr eine Abstraktionsebene dar, die zur einfacheren Handhabung von Prolog-Befehlen eingeührt wurde, da sie die Handhabung der Prolog-Befehle auf die ErgoAI-Ebene heben.

Listing 7.7: ErgoAI Prolog Aufruf

```
?- predicate@\prolog(module).
```

Prolog-Module sind durch die Verwendung der Suffixe `@\prolog` oder `@\plg` erkennbar. Listing 7.7 zeigt die Syntax für den Aufruf von Prolog-Modulen. ErgoAI wird bereits mit diversen Prolog-Modulen ausgeliefert und diese müssen nicht mehr explizit geladen werden (Kifer, 2018b).

7.2.3 System Module

Systemmodule sind ein integrierter Bestandteil von ErgoAI und stellen diverse Systemoperationen (z.B. I/O-Operationen, Funktionen zur Datenspeicherung) zur Verfügung. Sie wurden eingeführt, um häufig verwendete Aktionen in einer einfachen und standardisierten Weise durchführen zu können.

Listing 7.8: Aufruf eines ErgoAI Systemmoduls

```
1 ?- write(Hello World!)@\io.
```

Sie werden durch die Verwendung der `@\modname` Syntax aufgerufen, wobei `modname` ein Platzhalter für den Namen des Systemmoduls ist. Listing 7.8 zeigt ein Beispiel für einen Methodenaufruf aus dem I/O-Modul (Kifer, 2018b). Dieses Beispiel ruft die `write(?Obj)` Methode aus dem I/O-Systemmodul (`@\io`) auf und schreibt den Text „Hello World!“ in den aktuellen Outputstream.

7.3 Schnittstellen und Datenzugriff

ErgoAI bietet im Vergleich zu Flora-2 einen erweiterten Umfang von Schnittstellen für den Zugriff auf externe Datenquellen. Neben den in Flora-2 vorhandenen Schnittstellen für den Zugriff auf Flora-2 aus Java Applikationen (Java-to-Flora2 Interface) und einem XML-Parser, bietet ErgoAI zusätzlich eine Python-Schnittstelle (Python-to-Ergo Interface), sowie den Aufruf von Java-Applikationen aus ErgoAI heraus (Ergo-to-Java Interface). Zusätzlich zum Parsen von XML-Dateien unterstützt ErgoAI auch das Parsen von JSON-Dateien und das Importieren von tabellarischen Daten (z.B. CSV-Dateien). Außerdem ermöglicht es den Zugriff auf Datenbanken (SQL/SPARQL), RDF- und OWL-Dateien. Es unterstüzt außerdem den Aufruf von Webservices.

Diese Schnittstellen werden über die ErgoAI-Systemmodule bereits mitgeliefert und müssen bis auf wenige Ausnahmen nicht zusätzlich installiert oder konfiguriert werden. Eine genaue Dokumentation findet man im ErgoAI-Packages Manual (Kifer, 2018a). Im Folgenden werden nur die Schnittstellen behandelt, die für diese Arbeit relevant sind. Dazu gehört der XML-Parser für den direkten Zugriff auf GML-Daten, die SQL-Schnittstelle für die Abfrage von GML-Daten aus einem relationalen Datenbankschema und der Aufruf von Java-Programmen aus ErgoAI, für die Möglichkeit zur Umsetzung eines GeoSpatial-Reasonings (in dieser Arbeit nicht umgesetzt, Details dazu finden sich in Kapitel 9).

7.3.1 XML-Schnittstelle

ErgoAI unterstützt das Parsen von XML- und (X)HTML Dokumenten und kann diese direkt in ErgoAI-Objekte konvertieren. Es können entweder gesamte Dokumenten geparst werden, oder über XPath-Ausdrücke auf Teildokumente zugegriffen werden. Der XPath-Support funktioniert jedoch derzeit noch nicht out-of-the-box und muss gesondert konfiguriert werden. Die Funktionen zum Laden von XML-Dokumenten befinden sich im `@\xml` Modul. Das Modul bietet unterschiedliche

Parametrisierungsmöglichkeiten für den Zugriff und das Handling während dem Parsen. Listing 7.9 zeigt den Aufruf zum Parsen eines gesamten XML-Dokuments (Kifer, 2018a).

Listing 7.9: ErgoAI XML-Dokument laden

```
1 ?InDoc[load_xml(?Module) -> ?Warn]@\xml
```

Der Parameter ?InDoc ist ein Platzhalter für einen Input, der in verschiedenen Formen bereitgestellt werden kann. Der Input kann in Form einer Url, eines Dateinamens oder direkt als String übergeben werden. Der Parameter ?Module gibt an in welches benutzerdefiniertes Modul die XML-Datei gespeichert werden soll. Der Parameter ?Warn dient als Variable zur Speicherung von Warnmeldungen. Das Mapping von XML-Daten auf ErgoAI-Objekte beruht auf drei Grundprinzipien nach (Kifer, 2018a) und kann teilweise gesteuert werden:

- XML-Elemente, Attributwerte und Strings werden in F-Logic-Objekte umgewandelt
- XML-Elemente sind von ihrem Parent-Objekt über F-Logic-Frames mit dem gleichen Namen wie das XML-Element erreichbar
- XML-Attribute werden über F-Logic Frames abgebildet, wobei hier der Name der Name des XML-Attributs ist

Kommentare und Verarbeitungsinstruktionen werden vom Mapping ignoriert. Das Handling von Whitespaces kann über die set_mode(...)@\xml Funktion parametrisiert werden, wobei standardmäßig sämtliche Strings mit Whitespaces getrimmt werden. Listing 7.10 zeigt ein Beispiel für das Mapping von XML nach ErgoAI.

Listing 7.10: ErgoAI XML-Dokument Input

```
1 <AircraftCharacteristic id="ID_ADL_16">
2    <weight uom="LB">25000</weight>
3 </AircraftCharacteristic>
```

Listing 7.11 zeigt die resultierenden ErgoAI-Objekte. Beim Mapping wird automatisch ein Root-Element (obj1) als Einstiegspunkt in das XML-Dokument angelegt. Das ErgoAI-Objekt obj2 bezieht sich auf das XML-Element AircraftCharacteristic, das ein ID-Attribut mit Inhalt ID_ADL_16 und ein weiteres geschachteltes XML-Element weight enthält. Das ErgoAI-Objekt Obj3 bezieht sich auf das weight XML-Element, welches ein uom Attribut mit Inhalt LB enthält. Dieses Element enthält keine weiteren geschachtelten Elemente, sondern eine Zeichenkette (\text) mit dem Wert 25000.

Listing 7.11: ErgoAI XML-Dokument Output

```
1 obj1[AircraftCharacteristic -> obj2]
2 obj2[attribute(id) -> 'ID_ADL_16']
3 obj2[weight -> obj3]
4 obj3[attribute(uom) -> 'LB']
5 obj3[\text -> '25000']
```

XML-Attribute werden in Attribute (attribute(?Name)) mit gleichem Namen umgewandelt. Die Attribute werdem dem ErgoAI-Objekt zugeordnet, das das ursprüngliche XML-Element beschreibt,

zu dem das Attribut gehört. Der XML-Parser beherrscht auch den Umgang mit gemischten Inhalten in Form von einfachen Textelementen und geschachtelten XML-Elementen. Wie der ErgoAI XML-Parser das Mapping durchführt, kann über das Setzen des Importmodus erfolgen (Kifer, 2018a). Listing 7.12 zeigt den Befehl zur Aktivierung/Deaktivierung von Navigationslinks zwischen den geparsten XML-Elementen.

Listing 7.12: ErgoAI XML-Dokument Importmodus

```
1 ?- set_mode(nonavlinks)@\xml.
2 ?- set_mode(navlinks)@\xml.
```

Standardmäßig werden keine Navigationslinks angefordert (nonavlinks). Das hat den Vorteil, dass das Mapping etwas vereinfacht wird, ist aber nur sinnvoll wenn die Struktur des XML-Dokuments vorab bekannt ist. In Fällen wo die Struktur des Dokuments nicht bekannt ist und Navigationsinformationen zwischen Parent- und Child-Knoten notwendig sind, sollten Navigationslinks (navlinks) verwendet werden. Das verkompliziert zwar das Mapping, da für jeden Textknoten ein eigenes F-Logic-Objekt angelegt wird, dafür bleibt die Reihenfolge der Elemente erhalten und die Struktur des ursprünglichen XML-Dokuments kann bei Bedarf rekonstruiert werden.

7.3.2 SQL-Schnittstelle

ErgoAI unterstützt den Aufbau von Datenbankverbindungen und die Abfrage von externen Daten mittels SQL. Die notwendigen Funktionen befinden sich im `@\sql` Systemmodul. Bevor SQL-Abfragen abgesetzt werden können, muss eine Datenbankverbindung etabliert werden. ErgoAI bietet hierfür einen ODBC-Treiber und einen MySQL-Treiber. Der ODBC-Treiber ist der allgemeine Treiber für relationale Datenbanken und kann für sämtliche Datenbanken verwendet werden, die das ODBC-Protokoll unterstützen. Das umfasst unter anderem Produkte wie Microsoft SQL Server, Oracle und DB2. Der MySQL-Treiber ist der native Treiber für MySQL-Datenbanken. Da in dieser Arbeit eine Oracle-Datenbank verwendet wird, wird auf den MySQL-Treiber nicht näher eingegangen. Listing 7.13 zeigt den Aufruf zum Öffnen einer ODBC-Verbindung (Kifer, 2018a).

Listing 7.13: ErgoAI SQL-Verbindung aufbauen (ODBC)

```
1 odbc[open(?ConnectId,?DSN,?User,?Password)]@\sql.
```

Der Parameter `?ConnectId` muss vergeben werden und identifiziert die Verbindung um SQL-Abfragen absetzen zu können. Der Parameter `?DSN` muss einen gültigen Data Source Name enthalten. Die Datenquelle (Server, Datenbankname, etc.) muss separat definiert werden. Die Parameter `?User` und `?Password` definieren den Benutzer und das Passwort, mit dem die Datenbankverbindung hergestellt werden soll. Es wird empfohlen die Datenbankverbindung wieder zu schließen um Ressourcen freizugeben, wenn diese nicht mehr benötigt wird. Listing 7.14 zeigt den Befehl zur Schließung einer Datenbankverbindung durch Referenzierung auf die `?ConnectId` (Kifer, 2018a).

Listing 7.14: ErgoAI SQL-Verbindung schließen

```
1 ?ConnectId[close]@\sql.
```

ErgoAI unterstützt SQL-Statements zur Datenabfrage und Manipulation (z.B. Select, Update, Insert, Delete) und DDL-Befehle (z.B. Create, Alter, Drop). Listing 7.15 zeigt den Methodenaufruf einer Datenbankabfrage (Kifer, 2018a).

Listing 7.15: ErgoAI SQL-Abfrage absetzen

```
1 ?ConnectId[query(?QueryId,?QueryList,?ReturnList)]@\sql.
```

Der Parameter ?QueryId ist eine ID, über die die Abfrage referenziert werden kann. Der Parameter ?QueryList beschreibt das tatsächliche SQL-Statement und muss nach Konkatenierung zu einem validen SQL-Statement führen. Dieses kann aus einem reinen String bestehen oder auch ErgoAI-Variablen beinhalten. Der Parameter ?ReturnList ist das Resultset der Abfrage und ist nur für Select-Statements relevant. Wie auch in anderen Sprachen üblich, können Abfragen in ErgoAI auch in Form von Prepared Statements vorbereitet werden. Das ist im Vergleich zur direkten Ausführung oft effizienter, da diese Abfragen vorkompiliert und optimiert werden. Die Ausführung von Prepared Statements ist ein zweistufiger Prozess, welcher in Listing 7.16 gezeigt wird (Kifer, 2018a).

Listing 7.16: ErgoAI SQL-Abfrage vorbereiten und ausführen

```
1 ?ConnectId[prepare(?QueryId,?QueryList)]@\sql.
2 ?QueryId[execute(?BindList,?ReturnList)]@\sql.
```

Die Methode prepare(?QueryId,?QueryList) dient zur Vorbereitung der Abfrage. Die Bedeutung der Parameter ist dieselbe wie bei der direkten Ausführung. Der Parameter ?QueryList muss in diesem Fall Platzhalter in Form von ? beinhalten, an die später Variablen gebunden werden können. Die Funktion execute(?BindList,?ReturnList) dient zur Ausführung der Abfrage. Der Parameter ?BindList enthält eine Liste von Bind-Variablen und der Parameter ?ReturnList ist wiederum das Resultset der Abfrage.

7.3.3 Java-Schnittstelle

Die ErgoAI-to-Java Schnittstelle ermöglicht den Aufruf von beliebigen Java-Anwendungen aus ErgoAI heraus. Die Funktionalität befindet sich im Systemmodul @\e2j. Neben der Ausführung von beliebigen Java-Programmen, bietet das Modul auch einige Built-In Funktionen beispielsweise für die Anzeige von Benutzerdialogen, deren Input zurück in die Wissensdatenbank eingebunden werden kann oder zur Öffnung von benutzerdefinierten Fenstern aus ErgoAI heraus. Diese Funktionen sind jedoch für diese Arbeit nebensächlich und werden daher nicht näher behandelt. Eine genaue Dokumentation befindet sich im ErgoAI-Packages Manual (Kifer, 2018a). Über die ErgoAI-to-Java Schnittstelle ist es möglich der aufgerufenen Java-Anwendung Parameter zu übergeben und das Ergebnis zurück nach ErgoAI zu übermitteln. Listing 7.17 zeigt den Befehl zum Laden eines Java-Archivs in das ErgoAI-System (Kifer, 2018a).

Listing 7.17: ErgoAI Java Jar laden

```
1 System[addJar(?Jar)]@\e2j
```

Der Parameter ?Jar beinhaltet den Pfad zum gewünschten Java Archiv. Nach dem Laden des Archivs können dessen öffentliche Methoden aufgerufen werden und das Ergebnis des Methodenaufrufs zurückgegeben werden. Listing 7.18 zeigt einen Methodenaufruf eines Java Archivs aus ErgoAI. Der Begriff JavaObjSpec ist ein Platzhalter für den übergebenen Datentyp oder die Art der Datenstruktur. Derzeit wird jedoch nur eine begrenzte Anzahl von Datentypen und Datenstrukturen unterstützt, unter anderem Integer, Double, Long, Strings, Listen und Arrays.

Listing 7.18: ErgoAI Java-Methode aufrufen

```
JavaObjSpec[message(JavaMethodWithArgs) -> ?Result]@\e2j
```

Das Ergebnis des Methodenaufrufs wird in der ?Result Variable abgelegt, wobei der Datentyp vom Rückgabetyp der Java-Methode abhängig ist. Listing 7.19 zeigt ein Beispiel für den Aufruf einer Java-Methode zur Durchführung von String-Operationen (Kifer, 2018a). In diesem Beispiel wird eine Split-Methode aufgerufen, der der String '123abc789' und die Splitbedingung 'abc' übergeben wird. Das Ergebnis ist ein Array ['123', '789'], welches in der Variable ?P abgelegt wird.

Listing 7.19: ErgoAI Java-Methodenaufruf Ergebnis zurückgeben

```
?- '123abc789'[message(split(abc))->?P]@\e2j.
?P = ['123', '789']
```

ErgoAI kann umgekehrt auch von Java-Programmen aufgerufen werden. Das ist dann nützlich, wenn die Reasoning Engine nicht durch einen menschlichen Benutzer verwendet wird, sondern maschinell aufgerufen werden soll. Die Java-To-Ergo Schnittstelle unterteilt sich in eine low-level und eine high-level API. Die low-level API kann verwendet werden, um beliebige Abfragen an ErgoAI zu schicken und das Ergebnis an das aufrufende Programm zurückzugeben.

Listing 7.20: ErgoAI Wissensdatenbank von Java-Programm aus laden und abfragen

```
import java.util.*;
import net.sf.flora2.API.*;

public class JavaToErgo {

   public static void main(String[] args) {

      // Flora Session erzeugen
      FloraSession session = new FloraSession();

      // Datei in Modul default laden
      session.loadFile("C:\\Path_to_File\\JavaToErgo.flr", "default_mod");

      // Alle Flugzeuge aus JavaToErgo.flr abfragen
      Iterator<FloraObject> aircrafts =
      session.ExecuteQuery("?X:aircraft@default_mod.");

      // Informationen zu den Flugzeugen ausgeben
      while (aircrafts.hasNext()) {
         FloraObject aircraft = aircrafts.next();
         System.out.println("Aircraft:" + aircraft);
      }

      // Session beenden
      session.close();
   }
}
```

Die high-level API stellt eine Verbindung zwischen Java-Klassen und ErgoAI-Klassen über Proxies her. Die high-level API ist jedoch derzeit noch experimentell und in seiner Funktionalität im Vergleich zur low-level API relativ eingeschränkt (z.B. nur alphanumerische Namen zulässig). Aus diesem Grund wird auf die high-level API an dieser Stelle nicht näher eingegangen. Beide APIs benötigen die XSB-Schnittstelle (Interprolog) (Calejo, 2004). Damit die low-level API verwendet werden kann, müssen im Java-Programm beispielsweise mit `System.setProperty()` die Parameter `PROLOGDIR` (XSB-Binary Verzeichnis) und `FLORADIR` (ErgoAI Installationsverzeichnis) gesetzt werden.

Um Abfragen an ErgoAI schicken zu können, muss zuerst eine Session zu einer ErgoAI-Instanz aufgebaut werden. Das geschieht automatisch bei der Anlage eines `FloraSession` Objekts. Diese Klasse kapselt sämtliche Funktionalität, die für das Laden von ErgoAI-Dateien und für Abfragen auf diese notwendig ist. Listing 7.20 zeigt ein Beispiel-Programm für das Laden einer ErgoAI-Datei in ein Basismodul und die Definition einer Abfrage über alle Flugzeuge in der Wissensdatenbank. Eine genaue Dokumentation über die vorhandenen Methoden kann in (Kifer, 2018a) gefunden werden.

Teil II

Umsetzung und Evaluierung

Kapitel 8

Anforderungen, Ziele und Nicht-Ziele

Dieses Kapitel behandelt die Anforderungen, die die Zugriffsmöglichkeiten auf GML-Daten aus ErgoAI erfüllen müssen. Im Folgenden werden für jede Anforderung sowohl deren Problembeschreibung, als auch der Hintergrund für die Definition der Anforderung diskutiert. Es werden außerdem die Ziele und Nicht-Ziele des Systems definiert. Da für die Transformation der GML-Daten in dieser Arbeit drei grundlegende Ansätze präsentiert werden, muss auch für jeden Ansatz die Erfüllung der Anforderungen und Ziele gesondert betrachtet werden, wobei sämtliche Anforderungen und Ziele für alle drei Lösungswege gelten. Eine Gegenüberstellung und Evaluierung der Erfüllung der Anforderungen und Ziele für jeden Ansatz erfolgt gesondert in Kapitel 13.

8.1 Anforderungen

In diesem Abschnitt werden die Anforderungen für den Zugriff auf GML-Daten aus ErgoAI beschrieben. Es wird pro Anforderung eine Problembeschreibung und der Hintergrund für die Definition der Anforderung angegeben.

A1 - Das Ursprungsdokument muss vollständig abgebildet werden:
Die XML-Dokumente müssen vollständig im Zielsystem abgebildet bzw. abgefragt werden können. Das bedeutet, dass der Zugriff auf sämtliche GML-Objekte, GML-Features und GML-Attribute auch im Zielsystem gewährleistet sein muss. Diese Anforderung lässt sich in folgende Anforderungen untergliedern:

- **A1.1 - Es muss zu sämtlichen Elementen des Ursprungsdokuments navigiert werden können:**
  In XML-Dokumenten ergibt sich die Navigierbarkeit der Dokumente aufgrund der hierarchischen Struktur von XML. In objektorientierten oder relationalen Datenstrukturen wie beispielsweise einer Datenbank oder in ErgoAI muss die Navigierbarkeit über die Vergabe von eindeutigen IDs und Schlüsseln sichergestellt werden.
- **A1.2 - Namespaces müssen berücksichtigt werden:**
  XML ist darauf ausgelegt, dass Dokumente aus verschiedenen Quellen in Applikationen gemeinsam verarbeitet werden können. Das Namespace-Konzept wie es in XML umgesetzt ist muss auch im Zielsystem berücksichtigt werden, damit Elemente aus verschiedenen Quellen unterschieden werden können.

- **A1.3 - Datentypen der Werte müssen ermittelt werden können:**
  Um logische, arithmetische und Vergleichsoperationen zu ermöglichen, müssen die Datentypen der Felder im Zielsystem verfügbar sein.

A2 - Es muss eine Zuordnung der transformierten XML-Elemente zu ihren Ursprungsdokumenten möglich sein:
Da in GML und aufbauenden Applikationsschemas der Kontext zu anderen Elementen des Ursprungsdokuments eine Rolle spielen kann, muss dieser Kontext auch in ErgoAI verfügbar sein.

A3 - Die Transformation muss generisch sein:
Bei Ansätzen, die einer Transformation bedürfen, muss diese Transformation generisch erfolgen. Das bedeutet, dass die Transformation für alle GML-basierten Daten unabhängig von domänenspezifischen GML-Implementierungen (z.B. AIXM, IWXXM) funktionieren muss.

A4 - Die Transformation muss schemalos sein:
Bei Ansätzen, die einer Transformation bedürfen, muss auf XML-Schemainformationen verzichtet werden. Die Transformation muss rein auf XML-Dokumentebene erfolgen, da XML-Schemas nicht generisch sind, sondern für die jeweiligen domänenspezifischen GML-Implementierungen spezifisch sind.

8.2 Ziele

In diesem Abschnitt werden die Ziele für die Zugriffsmöglichkeiten auf GML-Daten vorgestellt. Die Ziele für die Umsetzung der Zugriffsmöglichkeiten lassen sich in Performanceziele und Ziele mit Bezug auf die Wartbarkeit und Benutzerfreundlichkeit der umgesetzten Varianten gliedern. Die Messung der Performance erfolgt für jeden Ansatz gesondert und die Ansätze sollen anschließend hinsichtlich ihrer Performance verglichen werden können.

Z1 - Performance:
Für die Messung der Performance müssen die Aspekte Transformation, Laden und Abfrage der Daten separat betrachtet werden, wobei nicht jeder dieser Teilaspekte für jede Variante relevant ist. Welcher Teilaspekt für welchen Ansatz für die Auswertung der Performance relevant ist wird in Kapitel 13 erläutert. Dieses Ziel lässt sich in folgende Ziele unterteilen:

- **Z1.1 - Schnelle Transformation:**
  Bei Ansätzen, die einer Transformation bedürfen, soll diese möglichst schnell erfolgen. Die benötigte Zeit für die Transformation wird für jeden Ansatz anhand des gleichen Testdatensets (GML-Dokumente) verglichen (siehe Kapitel 13).
- **Z1.2 - Schnelles Laden:**
  Bei Ansätzen, bei denen die GML-Dokumente vollständig in ErgoAI geladen werden, soll die benötigte Zeit für das Laden möglichst gering sein. Die Ladezeiten werden für jeden Ansatz anhand des gleichen Testdatensets verglichen.
- **Z1.3 - Schnelle Abfragen:**
  Die Abfrage der Daten soll möglichst schnell erfolgen. Um die Abfragezeiten vergleichen zu können, werden für jeden Ansatz die gleichen Testabfragen verwendet.

Z2 - Wartbarkeit und Benutzerfreundlichkeit:
Die Wartbarkeit der Transformationsprogramme (sofern relevant) und die Benutzerfreundlichkeit bei der Formulierung von Abfragen soll für jede Zugriffsmöglichkeit erhoben und miteinander verglichen werden. Dieses Ziel lässt sich in folgende Ziele unterteilen:

- **Z2.1 - Wartbarkeit der Transformationsprogramme:**
  Dieses Ziel bezieht sich nur auf Ansätze, die einer Transformation bedürfen. Dabei geht es um die Code-Wartbarkeit der implementierten Transformationsprogramme. Die Definition dieses Ziels legt den Fokus auf Systementwickler, die zu einem späteren Zeitpunkt eventuell Anpassungen an den Transformationsprogrammen durchführen müssen.
- **Z2.2 - Benutzerfreundlichkeit bei der Formulierung von Abfragen:**
  Dieses Ziel ist für alle Ansätze relevant, da bei allen Ansätzen Abfragen gestellt werden. Die GML-Daten sollen in einer derartigen Form zur Verfügung gestellt werden, dass Abfragen auf diese für Endbenutzer möglichst einfach zu definieren sind.

8.3 Nicht-Ziele

In diesem Abschnitt werden zur Abgrenzung der Arbeit Nicht-Ziele definiert. Diese Punkte müssen bei der Umsetzung der Zugriffsmöglichkeiten nicht berücksichtigt werden.

N1 - Rücktransformation:
Bei Ansätzen, die einer Transformation bedürfen, muss das Ursprungsdokument aus den transformierten Daten nicht wiederhergestellt werden können. Es müssen daher Aspekte wie beispielsweise die Reihenfolge der Elemente in der ursprünglichen XML-Struktur nicht berücksichtigt werden.

N2 - Erstellung einer Wissensdatenbank:
Der Fokus dieser Arbeit liegt auf der Datenhaltung und dem Datenzugriff von GML-Dokumenten aus ErgoAI. Die Definition von Ableitungsregeln oder sonstigen Konstrukten im Sinne eines deduktiven Ansatzes zur Generierung von impliziten Fakten ist die Aufgabe eines Knowledge Engineers. Der Umfang der erstellten Wissensdatenbank beschränkt sich daher auf die GML-Dokumente, die entweder durch Transformation und Laden oder durch Ad-Hoc-Zugriff in ErgoAI zur Verfügung stehen.

N3 - Umsetzung von komplexer Rechenlogik für geometrische GML-Objekte (GeoSpatial Reasoning):
Die Verarbeitung von geometrischen Formen wie Punkten, Kurven, Linien, Flächen oder 3-dimensionalen Objekten (siehe Kapitel 4.2) ist an sich ein komplexes Unterfangen bei der Verarbeitung von GML-Daten, da dies teilweise komplexe Rechenoperationen erfordert. Die Umsetzung solcher Rechenlogik ist daher nicht Teil dieser Arbeit, es wird in Kapitel 9 jedoch kurz angesprochen, wie man mit den Möglichkeiten von ErgoAI eine solche Rechenlogik umsetzen könnte.

Kapitel 9

Architektur und Überblick

In diesem Kapitel wird ein Überblick über die Gesamtarchitektur für die Zugriffsmöglichkeiten auf GML-Daten aus ErgoAI gegeben und es wird auf gemeinsame Problemstellungen eingegangen, die alle Zugriffsmöglichkeiten betreffen. Außerdem werden die Zugriffsmöglichkeiten bezüglich der Zugriffsart und der Art ihres In- und Outputs gegenübergestellt. Die Ausgangsbasis für den Zugriff stellt die Geography Markup Language dar, da darauf diverse domänenspezifische Applikationsschemas (z.B. AIXM, IWXXM) aufsetzen. Der Datenzugriff soll jedoch nicht nur für spezifische Applikationsschemas funktionieren, sondern generisch für alle Datenmodelle, die auf GML basieren. Aus diesem Grund wurden folgende Ansätze für den Zugriff auf GML-Daten aus ErgoAI definiert:

- **Zugriffsmöglichkeit 1:** Transformation in ErgoAI-Fakten und Laden in ErgoAI
- **Zugriffsmöglichkeit 2:** Transformation in relationales Datenmodell und Zugriff über SQL-Schnittstelle (2 Varianten)
- **Zugriffsmöglichkeit 3:** Direkter Zugriff auf GML-Dokumente

Die Gesamtarchitektur für die Zugriffsmöglichkeiten auf GML-Daten aus ErgoAI ist in Abbildung 9.1 dargestellt. Für die Transformation in ein relationales Datenmodell wird eine Datenbank benötigt. ErgoAI enthält einen integrierten Open Database Connectivity (ODBC) Treiber und unterstützt damit den Zugriff auf die gängigsten Datenbanksysteme. Als Datenbanksystem wird eine Oracle Express 18c Datenbank verwendet. Die Transformation der GML-Daten in ein relationales Schema erfolgt durch ein Transformationsprogramm (GML-to-Relational Mapper), das in Java implementiert wurde. Die direkte Transformation der GML-Daten in ErgoAI-Fakten erfolgt ebenfalls durch ein Java Transformationsprogramm (GML-to-ErgoAI Mapper). Für den direkten Zugriff auf GML-Dokumente wird der in ErgoAI integrierte XML-Parser verwendet.

Bei den umgesetzen Zugriffsmöglichkeiten handelt es sich sowohl um Ansätze, die einer Transformation bedürfen, als auch um Ansätze bei denen ein Ad-Hoc-Zugriff erfolgt. Es ist auch eine Mischung von Transformation und Ad-Hoc-Zugriff möglich. Als Input erhalten alle Ansätze GML-Dokumente, der Output ist jedoch abhängig vom jeweiligen Ansatz und spielt eine Rolle für die Abfrage der GML-Daten. Tabelle 9.1 gibt einen Vergleich der umgesetzen Zugriffsmöglichkeiten bezüglich der Art ihres resultierenden Outputs und der Art des Zugriffs.

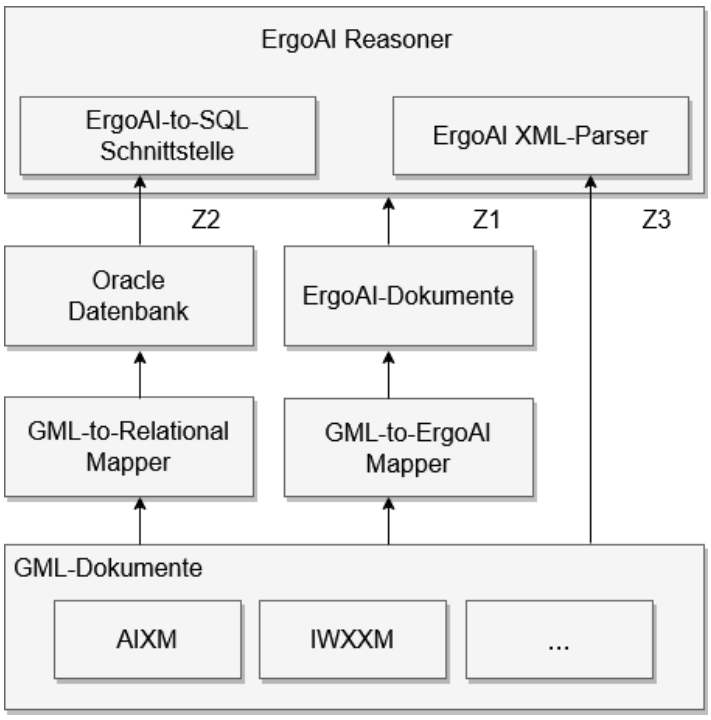


Abbildung 9.1: Gesamtarchitektur für die Zugriffsmöglichkeiten auf GML-Daten aus ErgoAI

Beim Zugriff auf GML-Daten aus ErgoAI entstehen einige Probleme, die gelöst werden müssen. XML ist ein hierarchisch semi-strukturiertes Datenformat, wohingegen ErgoAI ein objektorientert semi-strukturiertes Datenformat verwendet. Dazu kommen noch Eigenheiten von GML und Einschränkungen, die sich aus den Anforderungen an die Zugriffsmöglichkeiten ergeben (z.B. kein Zugriff auf XML-Schemainformationen). Dieses Kapitel beschreibt die Probleme, die bei den Zugriffsmöglichkeiten berücksichtigt werden müssen. Ob und wie diese Probleme gelöst werden können, wird in den Kapiteln zur Umsetzung der Zugriffsmöglichkeiten erläutert.

Tabelle 9.1: Outputs und Zugriffsart der Ansätze

Ansatz	**Zugriffsart**	**Output**
Transformation und Laden in ErgoAI	Transformation	ErgoAI-Objekte (in Dokumenten)
Transformation in relationales Schema Variante 1 (Spezifische Abfragen für ein bestimmtes Problem)	Transformation Ad-Hoc	Relationales Datenmodell
Transformation in relationales Schema Variante 2 (Generische Abfragen zum Laden der Wissensdatenbank)	Transformation	ErgoAI-Objekte
Direkter Zugriff auf GML-Dokumente	Ad-Hoc	ErgoAI-Objekte

Navigierbarkeit und Vergabe von eindeutigen IDs

In XML ergibt sich die Zuordnung der Elemente aufgrund der hierarchischen Struktur der Dokumente. In ErgoAI und in relationalen Datenbanken ist die Anordnung der Objekte nicht relevant. Es muss daher ein Mechanismus gefunden werden, um weiterhin zwischen den ursprünglichen XML-Elementen navigieren zu können. Dafür ist die Vergabe von eindeutigen IDs für die Erstellung von ErgoAI-Objekten bzw. den Entwurf eines relationalen Datenmodells notwendig. Da GML über das `gml:id` Attribut bereits über einen Mechanismus zur eindeutigen Identifikation von GML-Objekten verfügt, sollte dafür auch diese ID verwendet werden.

Für GML-Properties sieht der GML-Standard eine solche ID jedoch nicht vor. Deshalb muss für Properties zunächst ein Mechanismus für die eindeutige Identifizierung gefunden werden, bevor eine Transformation stattfinden kann. Dabei muss beispielsweise auf das Namespace-Konzept von XML Rücksicht genommen werden. Die Reihenfolge der Elemente spielt in GML weitgehend keine Rolle. Die einzige Ausnahme bilden Aggregationen (siehe Kapitel 4.2.2) bei der Definition von geometrischen Objekten (Portele, 2007). Hier spielt die Reihenfolge der Elemente innerhalb der Aggregation eine Rolle. Generell muss das mehrfache Vorkommen von Elementen bei der Schlüsselvergabe (z.B. durch Einführung eines Indexes) beachtet werden. Das bezieht sich aber nur auf GML-Properties mit gleichem Namen und Namespace in der gleichen Hierarchieebene, da damit auch die Sonderfälle abgedeckt sind, bei denen die Reihenfolge eine Rolle spielt.

Verfügbarkeit von Datentypen

Die Datentypen von GML-Properties mit textuellem Inhalt müssen in ErgoAI zur Verfügung stehen, damit arithmetische Operationen oder Vergleichsoperationen (z.B. Einschränkung auf einen bestimmten Datumsbereich) möglich sind. Die Datentypinformationen sind grundsätzlich in den XML-Schemas von GML und den domänenspezifischen GML-Applikationsmodellen vorhanden. Da die Transformation jedoch generisch und schemalos erfolgen muss, kann auf diese Informationen nicht zugegriffen werden. Es muss daher eine andere Möglichkeit gefunden werden, um die Datentypen der Textfelder für die Verwendung in ErgoAI zu ermitteln.

Zuordnung zu den Ursprungsdokumenten

Da die Zuordnung von GML-Properties zu GML-Objekten in XML durch die Hierarchie der Elemente implizit gegeben ist, muss ein Mechanismus gefunden werden, mit dem diese Zuordnung auch in ErgoAI möglich ist. Da GML-Objekte mit gleicher globaler ID in unterschiedlichen Dokumenten vorkommen können, ist eine lokale Zuordnung der GML-Properties zu diesen GML-Objekten nicht ausreichend, da dadurch der Kontext zum Rest des Dokuments verloren gehen würde. Es muss daher ein Mechanismus gefunden werden, mit dem GML-Objekte und GML-Properties (und in weiterer Folge auch zugehörige Attribute) eindeutig zu ihren Ursprungsdokumenten zugeordnet werden können.

Exkurs: GeoSpatial Reasoning

Abschließend erfolgt ein kurzer Exkurs, wie man die Rechenlogik für ein GeoSpatial Reasoning im Rahmen von ErgoAI umsetzen könnte. Ein wichtiges Thema im Umgang mit GML-Daten ist das Rechnen mit geometrischen Formen. Dafür sind teilweise komplexe Rechenoperationen (z.B. Ermittlung von Schnittpunkten oder Abständen zwischen zwei Formen in einem bestimmten Koordinatenraum) notwendig, die in logischen Programmiersprachen wie ErgoAI nur umständlich umsetzbar sind. Eine Möglichkeit wäre das Parsen der gesamten geometrischen Objekte als Strings zusätzlich zu der normalen Transformation und die Verwendung der eingebauten ErgoAI-Funktionen um geometrische Rechenoperationen umzusetzen. Einen solchen Ansatz verwendet Dieter Steiner in seiner Dissertation am Institut für Wirtschaftsinformatik – Data & Knowledge Engineering an der Johannes Kepler Universität Linz für das SemNOTAM Projekt (zum Zeitpunkt dieser Arbeit noch nicht veröffentlicht).

Eine Alternative wäre es, die geometrischen GML-Objekte überhaupt nicht nach ErgoAI durchzuschleusen, da das Rechnen damit in ErgoAI ohnehin sehr umständlich ist und dies nur unnötige Hardware-Ressourcen beanspruchen würde. Man könnte stattdessen über eine der ErgoAI-Schnittstellen ein Service anbinden (z.B. Java Programm oder Webservice), das diese Rechenlogik übernimmt. Diesem Service könnte man stattdessen nur die Referenz auf das geometrische GML-Objekt (Datei und `gml:id`) übergeben und das Service könnte sich dann das Objekt direkt aus den XML-Ursprungsdokumenten laden. Für Java gibt es beispielsweise bereits die Open Source GeoTools Bibliothek (https://www.geotools.org/), die das Rechnen mit geografischen Daten ermöglicht.

Kapitel 10

Zugriffsmöglichkeit 1: Transformation in ErgoAI-Fakten und Laden in ErgoAI

In diesem Kapitel wird die direkte Transformation der GML-Dokumente aus dem XML-Format in F-Logic Fakten behandelt. Die GML-Dokumente werden zunächst mit Hilfe eines Mapping-Algorithmus transformiert und die resultierenden Dokumente werden als ErgoAI-Dokumente (*.ergo) abgespeichert. Diese können durch das Laden in Module in Wissensdatenbanken verwendet werden. In den folgenden Unterkapiteln wird zunächst der vorgeschlagene Ansatz für die Transformation und anschließend das Laden und die Abfrage in ErgoAI anhand von Beispielen besprochen.

10.1 Transformation von GML-Daten nach ErgoAI

Die Transformation der GML-Daten in die ErgoAI-Syntax erfolgt durch einen Mapping-Algorithmus, der in Java implementiert wurde. Dieser wurde rekursiv implementiert und macht sich die Eigenschaften des Object-Property-Modells zunutze. Die GML-Dokumente werden zunächst in ein Document Object Model (DOM) geparst. Die GML-Elemente (Objekte, Properties, Attribute) werden dann in korrespondierende ErgoAI-Objekte transformiert. Die resultierenden ErgoAI-Objekte werden in der Frame Syntax dargestellt. Um ErgoAI-Objekte referenzieren zu können, müssen diese eine eindeutige ID bekommen. Die Zusammensetzung der IDs für GML-Objekte und GML-Properties wird bei der Erklärung der Transformation (siehe Tabelle 10.1) aufgeschlüsselt.

Für die Vergabe von IDs von GML-Objekten und GML-Properties werden in ErgoAI Function Terms verwendet. Diese erhalten die Felder der zusammengesetzen IDs als Argumente und ermöglichen somit die Zerteilung von IDs in atomare Teile, die auch wieder einzeln abgefragt werden können. Diese können vom Benutzer frei definiert werden. Für die IDs von GML-Objekten und GML-Properties wurden die Funktionen `qoid` (qualified object id) und `qpid` (qualified property id) eingeführt. Für das Herauslösen von Namespaces, sofern vorhanden, wurde die Funktion `cn` (complex name) definiert. Eine Alternative zu Function Terms wäre die Konkatenation mit Trennzeichen (z.B. Bindestrich oder Raute) der einzelnen Felder, aus denen sich die gesamte ID zusammensetzt. Da in ErgoAI jedoch sehr viele Sonderzeichen reserviert sind, bietet sich diese Option nicht an. Außerdem wäre die Abfrage auf einzelne Teile der IDs nur sehr umständlich (z.B. über Substring-Funktionen) möglich. Da in GML diverse Sonderzeichen (z.B. Punkte, Bindestriche) für IDs verwendet werden, die in ErgoAI reserviert sind, müssen diese Strings in ErgoAI unter einfache Hochkommas gesetzt werden, damit der ErgoAI-Compiler diese verarbeiten kann.

Tabelle 10.1: Mapping von GML-Elementen zu ErgoAI-Objekten

XML-Element	ErgoAI-Objekt
GML-Objekt	

`<⟨objectNS⟩:⟨objectName⟩`
`gml:id="⟨objectID⟩">`
`  <⟨propertyNS⟩:⟨propertyName⟩>`
`  </⟨propertyNS⟩:⟨propertyName⟩>`
`</⟨objectNS⟩:⟨objectName⟩>`
	`qoid(⟨filename⟩,⟨objectID⟩)[`
	`  elementtype->cn(⟨objectNS⟩,⟨objectName⟩),`
	`  cn(⟨propertyNS⟩,⟨propertyName⟩) ->`
	`  qpid(⟨filename⟩,⟨objectID⟩,⟨propertyNS⟩,`
	`  ⟨propertyName⟩,⟨Count⟩)`
	`].`

Beispiel	

`<message:AIXMBasicMessage gml:id="M1">`
`  <message:hasMember>`
`  </message:hasMember>`
`</message:AIXMBasicMessage>`
	`qoid(F1,M1)[`
	`  elementtype->cn('message','AIXMBasicMessage'),`
	`  cn('message','hasMember') ->`
	`  qpid('F1','M1','message','hasMember',1)`
	`].`

GML-Text-Property	

`<⟨Object⟩>`
`  <⟨propertyNS⟩:⟨propertyName⟩>`
`  ⟨String⟩`
`  </⟨propertyNS⟩:⟨propertyName⟩>`
`</⟨Object⟩>`
	`qpid(⟨filename⟩,⟨ElternID⟩,⟨propertyNS⟩,`
	`⟨propertyName⟩,⟨Count⟩)[`
	`  value -> '⟨String⟩'`
	`].`

Beispiel	

`<event:EventTimeSlice gml:id="ETS1">`
`  <event:encoding>DIG</event:encoding>`
`</event:EventTimeSlice>`
	`qpid('F1','ETS1','event','encoding',1)[`
	`  value -> 'DIG'`
	`].`

GML-Objekt-Property	

`<⟨Object⟩>`
`  <⟨propertyNS⟩:⟨propertyName⟩>`
`  <⟨childObject⟩>`
`  </⟨childObject⟩>`
`  </⟨propertyNS⟩:⟨propertyName⟩>`
`</⟨Object⟩>`
	`qpid(⟨filename⟩,⟨ElternID⟩,⟨propertyNS⟩,`
	`⟨propertyName⟩,⟨Count⟩)[`
	`  cn(⟨childObjectNS⟩,⟨childObjectName⟩)`
	`  ->`
	`  qoid(⟨childObjectFilename⟩,⟨childObjectID⟩)`
	`].`

Beispiel	

`<event:Event gml:id="EV1">`
`  <event:timeSlice>`
`  <event:EventTimeSlice gml:id="ETS1">`
`  </event:EventTimeSlice>`
`  </event:timeSlice>`
`</event>`
	`qpid('F1','EV1','event','timeSlice',1)[`
	`  cn(event,EventTimeSlice) ->`
	`  qoid('F1','ETS1')`
	`].`

XML-Element	ErgoAI-Objekt
GML-Attribut	
1 <⟨Object⟩/⟨Property⟩ ⟨attributeNS⟩:⟨attributeName⟩> 2 </⟨Object⟩>	1 ⟨objectID⟩/⟨propertyID⟩[2 attribute (cn(⟨attributeNS⟩,⟨attributeName⟩)) -> '⟨attributeValue⟩' 3].
Beispiel	
1 <event:textNOTAM xsi:nil="true" nilReason="inapplicable"/>	1 qpid('F1','ID_HDL_E011','event','textNOTAM',1)[2 attribute (cn('nilReason')) -> 'inapplicable', 3 attribute (cn('xsi','nil')) -> 'true' 4].

Der Ansatz für die Transformation der GML-Daten in die ErgoAI-Syntax erfolgt basierend auf der Dissertation von Dieter Steiner. In seiner Dissertation entwickelte er im Rahmen des SemNOTAM-Projekts einen Ansatz für das Mapping zwischen AIXM-Daten und ObjectLogic, der sich ebenfalls an den Eigenschaften des Object-Property-Modells orientiert. Außerdem ist ObjectLogic syntaktisch sehr ähnlich wie ErgoAI aufgebaut. Ein wesentlicher Unterschied zu seinem Ansatz ist jedoch, dass in seinem Ansatz keine eigenen Objekte (im ObjectLogic Sinn) für GML-Properties erzeugt werden, sondern diese stattdessen direkt in die darüberliegenden GML-Objekte geschachtelt sind. Das hat den Vorteil, dass bei der Abfrage die Navigationswege etwas kürzer sind. Es hat aber den Nachteil, dass bei der Zuordnung von Attributen zu Properties Codeduplizierung erfolgt, da er für jedes Property-Attribut einen eigenen Eintrag erzeugt, der das gesamte GML-Property enthält. Aus diesem Grund und aufgrund der besseren Übersichtlichkeit wurden die GML-Properties in dieser Arbeit in eigene ErgoAI-Objekte ausgelagert.

Um die Navigation zu vereinfachen, wurden der Transformation abschließend zwei Regeln hinzugefügt, mit denen die Property-Objekte übersprungen werden können. Dadurch bleiben die Vorteile (bessere Strukturierung, keine Codeduplizierung bei Properties) der eigenen ErgoAI-Objekte für GML-Properties erhalten und es besteht gleichzeitig die Möglichkeit einer direkteren Navigation. Das ist vor allem dann nützlich, wenn man direkt von einem GML-Objekt auf ein anderes GML-Objekt oder direkt auf den Wert von Text-Properties zugreifen will, ohne die Attribute der Properties zu benötigen. Auf die Umsetzung dieser Navigationsregeln wird in Kapitel 10.2 genauer eingegangen.

Eine Übersicht über die Transformation von GML-Elementen zu ErgoAI-Objekten wird in Tabelle 10.1 dargestellt. GML-Objekte erhalten in ErgoAI ihre `gml:id` als ID, da diese eindeutig ist. Zusätzlich wird der Name des Dokuments, aus dem das GML-Objekt stammt, Teil der ID um später eine Zuordnung zu diesen zu ermöglichen. Die Bezeichnung des XML-Elements (z.B. AIXMBasicMessage) wird dem ErgoAI-Objekt als Typattribut (`elementtype`) mitgegeben. Die resultierenden ErgoAI-Objekte enthalten die in den XML-Dokumenten geschachtelten Properties jedoch nicht direkt, sondern verweisen nur auf diese, da GML-Properties als eigene ErgoAI-Objekte dargestellt werden. Das ist deshalb der Fall, da GML-Properties ebenfalls Attribute enthalten können und da dadurch eine bessere Übersichtlichkeit gewahrt wird. Eine Schachtelung der GML-Properties in

die zugehörigen GML-Objekte wäre jedoch auch möglich (siehe Ansatz von Dieter Steiner). Bei dieser Variante müsste jedoch für jedes Property-Attribut ein eigener Eintrag erzeugt werden, der das gesamte GML-Property enthält. Das würde zusätzlich zur schlechteren Übersichtlichkeit zu Codeduplizierung führen.

Für die Generierung von eindeutigen IDs für GML-Properties wird die ID des übergeordneten GML-Objekts (Dokumentname, `gml:id`) verwendet. Der Name des Properties wird ebenfalls Teil der ID und auch der Namespace des Properties muss Teil der ID sein, um Konflikte von Properties mit gleichem Namen zu vermeiden. Da Properties mit gleichem Namen und gleichem Namespace auf der selben Hierarchieebene in XML vorkommen können, wird zusätzlich ein Zähler zur Unterscheidung benötigt. Beim Verweis auf ein GML-Property in einem GML-Objekt muss der zusammengesetzte Schlüssel zur eindeutigen Identifizierung des Properties verwendet werden. Das ErgoAI-Objekt für das Property selbst erhält dann ebenfalls diesen Schlüssel als ID. Bei Objekt-Properties erfolgt die Referenzierung auf das geschachtelte GML-Objekt analog indem auf die `gml:id` und den Dokumentnamen des geschachtelten GML-Objekts referenziert wird. Dieser Vorgang für die Erstellung und Referenzierung von GML-Objekten und GML-Properties anhand des Object-Property-Modells wechselt sich solange ab, bis die unterste Ebene (Text-Properties) erreicht ist. Auch Text-Properties werden als eigene ErgoAI-Objekte modelliert. Diese enthalten aber ihren Inhalt direkt als textuelles Attribut. XML-Attribute können dadurch bei GML-Objekten, Text-Properties und Objekt-Properties direkt in den korrespondierenden ErgoAI-Objekten angegeben werden.

Bei Text-Properties müssen die Datentypen angegeben werden, um später arithmetische Operationen und Vergleichsoperationen zu ermöglichen. Da XML-Schemainformationen, aufgrund der für diese Arbeit definierten Anforderungen, nicht verwendet werden können, werden die Datentypen direkt bei der Transformation der GML-Dokumente durch Parsingversuche ermittelt. Es wird somit versucht, ob ein String-Wert beispielsweise in einen Integerwert oder in ein Datum geparst werden kann. Ist das Parsing in einen Datentyp erfolgreich, wird angenommen, dass es sich bei dem ursprünglichen Wert auch tatsächlich um diesen Datentyp gehandelt hat. Dieses Vorgehen ist zwar nicht ideal, aber es stellt eine Möglichkeit dar um ohne XML Schema auszukommen. Folgende Datentypen werden unterstützt:

- Boolean
- Integer
- Double
- Long
- DateTime (nach ISO 8106)

Sollte das Parsing für die unterstützten Werte nicht erfolgreich sein, wird der Wert weiterhin als String behandelt. Diese Datentypen werden so auch in ErgoAI unterstützt und wurden in ErgoAI zum Teil auch analog dazu implementiert, wie sie in XML definiert sind (z.B. Datums- und Zeitangaben). Die Angabe von Datentypen erfolgt in ErgoAI syntaktisch in der Form `"<Wert>"^^\<Datentyp>` (z.B. `"2019-03-11T13:00:00Z"^^\dateTime`).

Exkurs: Alternative Darstellung von GML-Properties

Bei der Umsetzung der Transformation wurde neben der bereits vorgestellten Variante auch eine alternative Darstellung von GML-Properties in Betracht gezogen. Bei dieser Darstellung wird der ursprüngliche Schlüssel der Properties angepasst. Anstatt des bisher eingeführten Schlüssels (Dateiname/ID des übergeordneten GML-Objekts, Namespace, Name, Index) könnte man eine Nummerierung der Properties innerhalb eines Dokuments einführen. Der Schlüssel setzt sich dann nur noch aus dem Dateinamen des Ursprungsdokuments und dessen Nummer innerhalb des Dokuments zusammen. Der Dateiname des Ursprungsdokuments muss für eine eindeutige Zuordnung vorhanden bleiben. Bei Objekt-Properties erfolgt der Verweis in den ErgoAI-Objekten auf die Kind-Objekte anhand eines neu eingeführten `object` Attributs, bei Text-Properties erfolgt der Verweis auf den Wert weiterhin über das `value` Attribut. Die Alternative Darstellung von GML-Properties wird in Listing 10.2 anhand eines Ausschnitts aus einer AIXM-Nachricht (Listing 10.1) veranschaulicht.

Listing 10.1: Alternative Darstellung von GML-Properties Beispieldokument

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <message:AIXMBasicMessage
   xmlns:message="http://www.aixm.aero/schema/5.1.1/message"
   xmlns:gml="http://www.opengis.net/gml/3.2"
3  xmlns:aixm="http://www.aixm.aero/schema/5.1.1"
   xmlns:event="http://www.aixm.aero/schema/5.1/event" gml:id="M1">
4      <message:hasMember>
5          <event:Event gml:id="uuid.6fdfbe36-0e6e-4058-9bca-f2275c97e6c8">
6              <event:timeSlice>
7                  <event:EventTimeSlice gml:id="ID_HDL_E011">
8                      <aixm:interpretation>BASELINE</aixm:interpretation>
9                  </event:EventTimeSlice>
10             </event:timeSlice>
11         </event:Event>
12     </message:hasMember>
13 </message:AIXMBasicMessage>
```

Durch die Nummerierung gehen zwar keine Informationen verloren und die Darstellung der resultierenden ErgoAI-Objekte wird kompakter, sie bringt aber auch einen Nachteil mit sich. In der ursprünglichen Transformation ist bei Objekt-Properties der Name und der Namespace der Kind-Objekte Teil des referenzierenden Attributs. Dadurch kann bei Abfragen in der Navigation direkt über das Attribut auf bestimmte Objekte (z.B. `Event` oder `EventTimeSlice`) eingeschränkt werden. In dieser Variante muss dafür das `elementtype` Attribut ausgelesen und über eine String-Vergleichsfunktion geprüft werden. Dieser Fall tritt bei der Navigation in Abfragen jedoch häufig auf und dabei funktioniert die Formulierung von Abfragen in der ursprünglichen Variante einfacher. Ein weiterer Vorteil dieser Variante ist auch, dass dabei die Reihenfolge der Properties innerhalb eines Dokuments erhalten bleibt. Da in GML die Reihenfolge der Elemente bis auf wenige Sonderfälle (siehe Kapitel 9) jedoch keine Rolle spielt, war das kein ausschlaggebender Grund für die Bevorzugung dieser Variante. Aus diesen Gründen wurde die Darstellung der GML-Properties in der ursprünglichen Variante umgesetzt und diese Variante wird zur Vollständigkeit nur als Alternative angeführt.

Listing 10.2: Alternative Darstellung von GML-Properties in ErgoAI

```
qoid ('property_alternative','M1')[
   elementtype -> cn('message','AIXMBasicMessage'),
   attribute (cn ('gml','id')) -> 'M1',
   attribute (cn ('xmlns','aixm')) -> 'http://www.aixm.aero/schema/5.1.1',
   attribute (cn ('xmlns','event')) -> 'http://www.aixm.aero/schema/5.1/event',
   attribute (cn ('xmlns','gml')) -> 'http://www.opengis.net/gml/3.2',
   attribute (cn ('xmlns','message')) ->
   'http://www.aixm.aero/schema/5.1.1/message',
   cn ('message','hasMember') -> qpid ('property_alternative',1)
].

qpid ('property_alternative',1)[
   cn('event','Event' -> qoid
   ('property_alternative','uuid.6fdfbe36-0e6e-4058-9bca-f2275c97e6c8')
].

qoid ('property_alternative','uuid.6fdfbe36-0e6e-4058-9bca-f2275c97e6c8')[
   elementtype -> cn('event','Event'),
   attribute (cn ('gml','id')) -> 'uuid.6fdfbe36-0e6e-4058-9bca-f2275c97e6c8',
   cn ('event','timeSlice') -> qpid ('property_alternative',2)
].

qpid ('property_alternative',2)[
   object -> qoid ('property_alternative','ID_HDL_E011')
].

qoid ('property_alternative','ID_HDL_E011')[
   elementtype -> cn('event','EventTimeSlice'),
   attribute (cn ('gml','id')) -> 'ID_HDL_E011',
   cn ('aixm','interpretation') -> qpid ('property_alternative',3)
].

qpid ('property_alternative',3)[
   value -> "BASELINE"^^\string
].
```

10.2 Laden und Abfrage von GML-Daten in ErgoAI

Für das Laden in ErgoAI wird durch das Transformationsprogramm für jeden Durchlauf auch eine Gesamtdatei (example_kb.ergo) in dem Verzeichnis erstellt, in dem auch die einzelnen ErgoAI-Dateien der transformierten GML-Dokumente abgelegt werden. Diese Datei enthält über den #include Befehl sämtliche GML-Dokumente, die in diesem Durchlauf transformiert wurden. Sie stellt somit die Wissensdatenbank über die transformierten GML-Dokumente dar, auf die in weiterer Folge Regeln und Abfragen formuliert werden können. Listing 10.3 zeigt einen Ausschnitt der example_kb.ergo Datei.

Listing 10.3: Ausschnitt aus der Wissensdatenbank-Datei

```
1 #include "C:\<Pfad_zu_Output>\airmet-A6-1a-TS.ergo"
2 #include "C:\<Pfad_zu_Output>\airmet-translation-failed.ergo"
3 #include "C:\<Pfad_zu_Output>\DN_ACG.UNS_EADD.ergo"
4 #include "C:\<Pfad_zu_Output>\DN_AD.CLS_except_special_flights.ergo"
5 #include "C:\<Pfad_zu_Output>\DN_AD.LIM_limited_weight.ergo"
6 #include "C:\<Pfad_zu_Output>\DN_AD.LIM_prohibit_tgl.ergo"
```

Um Property-Objekte überspringen zu können und damit eine einfachere Navigation zu ermöglichen, wurden der example_kb.ergo Datei die beiden Regeln in Listing 10.4 hinzugefügt.

Listing 10.4: Regeln für das Überspringen von Property-Objekten

```
1 ?PARENT[op(?OPREF)->?qoid(?CHILD_NS,?CHILD_ID)] :-
2 ?PARENT[
3    ?OPREF->qpid(?_PAFILE,?_PAID,?_PRNS,?_PRNAME,?_PRI)[
4       ?_REFERENCE->?qoid(?CHILD_NS,?CHILD_ID)
5    ]
6 ].
7
8 ?PARENT[tp(?TEXTPROPERTY)->?VALUE] :- ?PARENT.?TEXTPROPERTY[value->?VALUE].
```

Für diesen Zweck wurden die Function Terms op (Objekt-Property) und tp (Text-Property) eingeführt. Mit diesen Function Terms werden in den ErgoAI-Objekten der GML-Objekte zusätzliche Hilfsfakten generiert, die direkt auf die Kind-Objekte von Objekt-Properties bzw. auf den Wert von Text-Properties verweisen. In der ersten Regel wird bei Objekt-Properties zum zwischengelagerten ErgoAI-Objekt für das Objekt-Property navigiert und dessen Referenz (ID) auf das Kind-Objekt zurückgegeben. In der zweiten Regel wird der Wert (value) des Text-Properties ausgelesen und direkt zurückgegeben. Die Verwendung dieser Regeln ist auch im nächsten Abschnitt bei der Formulierung von Beispielabfragen auf die Wissensdatenbank eingeflossen und wird dort veranschaulicht.

Im Folgenden wird nun die Verwendung und Formulierung von Abfragen auf die resultierende Wissensdatenbank anhand von einigen Beispielen erläutert. Diese Beispiele beziehen sich auf die AIXM und IWXXM Applikationsschemas und sollen einen Eindruck darüber geben, wie ErgoAI beispielsweise in einem realen Szenario für die Flugsicherheit eingesetzt werden kann. Diese Abfragen können entweder direkt in ErgoAI-Dateien eingebettet werden (mit Präfix ?-) oder über das Abfragefenster des ErgoAI-Studios abgesetzt werden. Für den Zweck der Veranschaulichung wurden folgende Beispielabfragen definiert:

- Abfrage 1: AIXM-Nachrichten, die ein bestimmtes Feature (z.B. AircraftGroundService) betreffen
- Abfrage 2: Alle AIXM Ereignisse, die gerade aktiv sind
- Abfrage 3: Aktuelle Wetter- und Windverhältnisse an einem bestimmten Flughafen (z.B. Flughafen Karlsbad)

AIXM-Nachrichten, die ein bestimmtes Feature (z.B. AircraftGroundService) betreffen

`AircraftGroundServices` bezeichnen in AIXM am Boden befindliche Wartungs- und Unterstützungsservices für Flugzeuge. Die Verfügbarkeit von diesen kann beispielsweise für Flugzeuge im Landeanflug auf einen Flughafen relevant sein. Eine mögliche Abfrage könnte somit die Auflistung aller Flugverkehrsnachrichten sein, die solche Bodenservices betreffen. Listing 10.5 zeigt die Umsetzung dieser Abfrage in ErgoAI.

Listing 10.5: Abfrage von Nachrichten zu Bodenservices in ErgoAI

```
?- qoid(?DATEI_NACHRICHT,?ID_NACHRICHT)[
   elementtype -> cn(?_MTNS,?_MESSAGETYPE),
   op(cn(?_MNS,hasMember))->qoid(?_DATEI_SERVICE,?ID_SERVICE)[
      elementtype -> cn(?_TNS,?TYPE)
   ]
],
?_MESSAGETYPE[equals(AIXMBasicMessage)]@\basetype,
?TYPE[equals(AircraftGroundService)]@\basetype.
```

Für diesen Zweck muss innerhalb der Nachricht zu dem ErgoAI-Objekt navigiert werden, das den Bodenservice beinhaltet. Dieses befindet sich in AIXM immer in einem `hasMember`-Property. Da in dieser Abfrage eventuell vorhandene Attribute der Property-Elemente nicht relevant sind, kann durch die verkürzte Navigation über den op Function Term das Property-Element übersprungen werden und direkt zum Kind-Objekt navigiert werden. In XML findet man den Typ des betroffenen Features direkt im Namen des XML-Elements, in diesem Fall `<aixm:AircraftGroundService>`. Für die Typbezeichnung wurde bei der Transformation ein eigenes `elementtype`-Feld erzeugt, das bei der Abfrage auch wieder ausgegeben werden kann. Um auf AIXM-Nachrichten bzw. Bodenservices einzuschränken, kann der Elementtyp (`AIXMBasicMessage`) bzw. (`AircraftGroundService`) der GML-Objekte verwendet werden. Die Einschränkung auf einen bestimmten Wert erfolgt in ErgoAI über die eingebaute Funktion `\string[|=> equals(\object)|]`. Eine Alternative für die Einschränkung auf einen bestimmten Elementtyp wäre es, anstatt der `?_MESSAGETYPE` und `?TYPE` Variablen den Wert direkt einzusetzen. Dann würde jedoch im Abfrageergebnis dieser Typ nicht erscheinen. In diesem Beispiel interessiert uns jedoch der Typ(`AircraftGroundService`) des abgefragten Services (die Einschränkung auf den Nachrichtentyp wurde aufgrund der Einheitlichkeit dann auch über die `equals` Methode umgesetzt).

Dieses Beispiel zeigt auch den Vorteil der Verwendung von Function Terms für die Definition von zusammengesetzten IDs in ErgoAI. Denn dadurch können bei der Abfrage die einzelnen Teile der IDs separat gebunden und verwendet werden. Für diese Abfrage wurde beispielsweise auf die Namespaces der Elemente keine Rücksicht genommen und diese wurden durch anonyme Variablen ersetzt. In diesem Fall wollen wir nur die Ursprungsdatei, die ID der Nachricht und die ID und den

Typ des betroffenen Services wissen. Das Ergebnis der Abfrage für die Beispiel-Wissensdatenbank ist in Tabelle 10.2 dargestellt.

Listing 10.6: Abfrage von Nachrichten zu Bodenservices in ErgoAI (Alternative)

```
?- qoid(?_DATEI_SERVICE,?ID_SERVICE)=
qoid(?DATEI_NACHRICHT,?ID_NACHRICHT).op(cn(?_MNS,hasMember)),
qoid(?_DATEI_SERVICE,?ID_SERVICE)[
   elementtype->cn(?_TNS,?TYPE)
],qoid(?DATEI_NACHRICHT,?ID_NACHRICHT)[elementtype->cn(?_MTNS, ?_MESSAGETYPE)],
?_MESSAGETYPE[equals(AIXMBasicMessage)]@\basetype,
?TYPE[equals(AircraftGroundService)]@\basetype.
```

ErgoAI bietet auch eine Punktnotation als verkürzte Schreibweise an. Diese ist vor allem dann sinnvoll, wenn lange Navigationswege zu einem Element notwendig sind und die Zwischenelemente im Abfrageergebnis nicht von Interesse sind. Dadurch kann die Definition und Bindung an anonyme Variablen vermindert werden. Die Punktnotation kann auch mit der regulären Frame-Syntax vermischt werden. Außerdem können Zwischenschritte über den = Operator an Variablen gebunden werden, von denen weiternavigiert werden kann. Listing 10.6 zeigt eine alternative Schreibweise für die Abfrage aus Listing 10.5, bei der eine gemischte Schreibweise aus Punktnotation und Frame-Syntax verwendet wird.

Tabelle 10.2: Abfrageergebnis von Nachrichten zu Bodenservices in ErgoAI

?DATEI_NACHRICHT	?ID_NACHRICHT	?ID_SERVICES	?TYPE
'DN_ACG.UNS_EADD'	M00001	'uuid.9f46fdb7-02d5-410b-92bd-b0b85fe177ea'	AircraftGroundService

Alle AIXM-Ereignisse, die gerade aktiv sind

Für den Flugverkehr ist es generell relevant, eine Übersicht über sämtliche Ereignisse zu haben, die den Flugverkehr bzw. die Flugsicherheit beeinflussen können. AIXM-Nachrichten verfügen durch das AIXM-Zeitmodell über einen Zeitstempel, der den Gültigkeitszeitraum der Nachricht angibt. Anhand von diesem kann ermittelt werden, ob eine Nachricht gerade zeitlich relevant ist. ErgoAI bietet über seine Vergleichsoperatoren die Möglichkeit, Datums- und Zeitangaben zu vergleichen und bietet somit beispielsweise die Möglichkeit für die zeitliche Eingrenzung von Nachrichten. Listing 10.7 zeigt wie eine solche Abfrage in ErgoAI auf die transformierten GML-Dokumente erfolgen kann. Eine solche Abfrage könnte in einem realen Flugsicherheitsszenario natürlich noch um weitere Aspekte verfeinert werden um eine effektive Filterung von Flugverkehrsnachrichten zu ermöglichen.

In AIXM werden Nachrichten in `TimeSlices` verpackt, die die zeitliche Gültigkeit definieren. Für Zeitintervalle sind diese Informationen in `TimePeriod` Elemente verschachtelt, die über ein `beginPosition` und `endPosition` Feld verfügen. Diese Felder markieren den Beginn bzw. das Ende des Gültigkeitszeitraums des Ereignisses. Um auf diese Werte einschränken zu können, muss zunächst in ErgoAI über das jeweilige `TimeSlice` und dessen `TimePeriod` Element zu diesen Feldern navigiert werden. Dies erfolgt wieder über den verkürzten Navigationsweg durch die generierten Hilfsfakten (op Function Term), da auch in dieser Abfrage keine Attribute von Interesse sind. Da es sich bei den beiden Feldern um Text-Properties handelt, kann der `tp` Function Term verwendet werden, um deren Werte direkt auszulesen.

Listing 10.7: Abfrage von aktiven AIXM-Ereignissen in ErgoAI

```
?- ?_NACHRICHT[
   elementtype -> cn(?_MTNS,?_MESSAGETYPE),
   op(cn(?_MNS,hasMember))->qoid(?DATEI_EVENT,?ID_EVENT)[
      elementtype -> cn(?_MEMNS,?_MEMBERTYPE),
      op(cn(?_EVTSNS,timeSlice))->?_EVTSO[
         op(cn(?_VTNS,validTime))->?_TP[
            tp(cn(?_BPNS,beginPosition))->?BEGINN,
            tp(cn(?_EPNS,endPosition))->?ENDE
         ]
      ]
   ]
],
?_MESSAGETYPE[equals(AIXMBasicMessage)]@\basetype,
?_MEMBERTYPE[equals(Event)]@\basetype,
\dateTime[now -> ?_NOW]@\basetype,
?_NOW[lessEq(?ENDE)]@\basetype,
?BEGINN[lessEq(?_NOW)]@\basetype.
```

Bei der Navigation muss neben dem Elementtyp des Wurzelelements (`AIXMBasicMessage`) auch auf den Elementtyp (`Event`) eingeschränkt werden, da AIXM-Nachrichten nicht nur Events als Mitglied haben können. Diese Einschränkung erfolgt aufgrund der besseren Strukturierung wieder über die `equals` Methode. Anhand der `<`, `=<`, `>`, `>=` Vergleichsoperatoren, die auch auf Datums- und Zeitangaben (`\date`, `\time`, `\dateTime`) anwendbar sind, lässt sich die zeitliche Einschränkung herstellen. Aus logischer Sicht muss der Beginn des Ereignisses vor dem aktuellen Zeitstempel und das Ende des Ereignisses nach dem aktuellen Zeitstempel liegen. Dafür ist es notwendig, dass die Datentypen der transformierten Werte bekannt sind. Für die Ermittlung des aktuellen Zeitstempels bietet ErgoAI die eingebaute Funktion `\dateTime[now -> ?DATETIME]`. Das Ergebnis der Abfrage für die Beispiel-Wissensdatenbank für das aktuelle Tagesdatum ist in Tabelle 10.3 dargestellt.

Tabelle 10.3: Abfrageergebnis von allen aktiven Ereignissen in ErgoAI

?DATEI_EVENT	?ID_EVENT	?BEGINN	?ENDE
DN_AD.CLS_except_special_flights	uuid.1d713318-a022-4f0f-808a-8eea31b3e411	2018-05-11T10:00:00	2021-05-11T18:00:0
DN_RTE.OPN_one_direction	uuid.ace72405-4255-4426-985e-bde51ce90f79	2018-09-13T11:00:00	2021-09-13T20:00:00
DN_AD.LIM_limited_weight	uuid.a1a51296-bc95-4e22-a41f-dc8163e8dab0	2019-04-10T11:00:00	2021-04-15T13:00:00

Aktuelle Wetter- und Windverhältnisse an einem bestimmten Flughafen

Dieses Beispiel zeigt die Abfrage der aktuellen Wetter- und Windverhältnisse aus einem routinemäßigen IWXXM-Wetterbericht (METAR) für den Flughafen Karlsbad. Ein Teil eines METAR-Berichts ist der betroffene Flughafen, der neben seinem Namen über einen eindeutigen Identifikationscode (`designator`) identifiziert werden kann. Für den Flughafen Karlsbad ist das der Code „LKKV“. Der erste Teil der Abfrage beinhaltet nur die Navigation zu diesem Feld und die Bindung des Wertes an die `?DESIGNATORT` Variable. Die Einschränkung auf einen bestimmten Wert erfolgt wiederum über die eingebaute Funktion `\string[|=> equals(\object)|]`. Um den Wert für die

Anzeige ohne Datentypinformation zu formatieren, wurde die Funktion `\string[|rawValue => \string|]` verwendet.

Listing 10.8: Abfrage von Wetter- und Windverhältnissen am Flughafen Karlsbad in ErgoAI

```
?- ?_NACHRICHT[
   op(cn(?_MNS,aerodrome))->?_AERODROME[
      op(cn(?_AEONS,timeSlice))->?_AETS[
         tp(cn(?_AEOTS,designator))->?_DESIGNATORT
]]],
?_DESIGNATORT[equals("LKKV"^^\string)]@\basetype,
?_DESIGNATORT[rawValue->?DESIGNATOR]@\basetype,
?_NACHRICHT[
   op(cn(?_MNS,observation)) ->?_OBSERVATION[
      cn(?_AIRTONS,airTemperature)->?_AIRT[
         value ->?AIRT, attribute (uom)->?AIRTUOM
      ],
      cn(?_DEWTONS,dewpointTemperature)->?_DEWTO[
         value ->?DEWT, attribute (uom)->?DEWTUOM
      ],
      op(cn(?_SFWPNS,surfaceWind))->?_SFW[
         cn(?_SFWONS,meanWindDirection)->?_MWDO[
            value->?MWD, attribute (uom)->?MWDUOM
         ],
         cn(?_MWSONS,meanWindSpeed)->?_MWSO[
            value->?MWS, attribute (uom)->?MWSUOM
]]]].
```

Die eigentlichen Angaben zu Wetter- und Windverhältnissen befinden sich in IWXXM in dem `MeteorologicalAerodromeObservation` Element. Im zweiten Teil der Abfrage wird zu diesem Element navigiert und es werden Angaben zur Lufttemperatur, Kondensationstemperatur, Windrichtung und Windgeschwindigkeit abgefragt. In den ursprünglichen IWXXM-Dokumenten wurden die Maßeinheiten für diese Werte als XML-Attribute angegeben. In den transformierten ErgoAI-Objekten findet man diese Attribute ebenfalls bei den korrespondierenden Objekten zu den ursprünglichen Text-Properties. Diese wurden über den Function-Term `attribute (<Wert>)` abgebildet.

Tabelle 10.4: Abfrageergebnis der Wetter- und Windverhältnisse am Flughafen Karlsbad in ErgoAI

?DESIGNATOR	?AIRT	?AIRTUOM	?DEWT	?DEWTUOM	?MWD	?MWDUOM	?MWS	?MWSUOM
LKKV	27	Cel	10	Cel	210	deg	2.6	m/s

Die Navigation zu den relevanten GML-Objekten kann hier wieder über den verkürzten Navigationsweg erfolgen. Da in dieser Abfrage jedoch auch Attribute von Text-Properties ausgelesen werden, muss explizit zu deren Property-Objekten navigiert werden. Dieses Beispiel zeigt damit auch, wie auf Property-Objekte weiterhin zugegriffen werden kann, falls deren Attribute von Interesse sind. Damit bei der Abfrage zur Vereinfachung nicht auf Namespaces Rücksicht genommen werden muss, wurden diese durch anonyme Variablen ersetzt. Die gesamte Abfrage ist in Listing 10.8 dargestellt. Das Ergebnis der Abfrage ist in Tabelle 10.4 dargestellt.

Kapitel 11

Zugriffsmöglichkeit 2: Transformation in relationales Datenmodell und Zugriff über SQL-Schnittstelle

Eine Möglichkeit für den Zugriff auf GML-Daten aus ErgoAI ist der Zugriff auf eine relationale Datenbank mit Hilfe der ErgoAI SQL-Schnittstelle. Dafür wird sowohl ein geeignetes relationales Datenmodell, als auch ein Algorithmus zur Transformation von GML-Daten in ein relationales Schema benötigt. Für die Transformation in ein relationales Schema gibt es zwei Möglichkeiten. Eine Möglichkeit wäre der Entwurf eines spezifischen Datenmodells, bei dem alle Features und sonstige Komponenten eines GML-Applikationsschemas ausmodelliert werden. Das würde beispielsweise im Fall von AIXM bedeuten, dass für alle Komponenten (z.B. Flughäfen, Landebahnen, Navigationshilfen, usw.) eigene Tabellen angelegt werden und diese entsprechend dem GML-Applikationsschema miteinander verknüpft werden. Das hat jedoch den Nachteil, dass dieses Datenmodell nur für ein bestimmtes GML-Applikationsschema gültig ist und für jedes weitere Applikationsschema extra angelegt werden muss. Das würde in einem relativ großen Datenmodell resultieren und verhältnismäßig viel Aufwand bedeuten. Aus diesem Grund wird stattdessen ein generisches Datenmodell vorgeschlagen, das sich den Umstand zu nutze macht, dass GML dem Object-Property-Modell (vgl. Kapitel 4.1) folgt. In diesem Datenmodell werden sämtliche Komponenten entweder als Features oder Properties von Features behandelt, die optional Attribute enthalten können.

Das generische relationale Datenmodell und der entwickelte Ansatz für die Transformation von GML-Daten in ein relationales Schema, werden im Folgenden vorgestellt. Auf Basis der transformierten Daten gibt es zwei Möglichkeiten um auf diese aus ErgoAI zuzugreifen. Bei der ersten Variante werden Ad-Hoc SQL-Abfragen geschrieben, die ein spezifisches ResultSet für eine bestimmte Problemstellung zurückgeben. Bei der zweiten Variante werden generische SQL-Statements geschrieben, die aus den in der relationalen Datenbank gespeicherten GML-Daten korrespondierende ErgoAI-Objekte analog zur direkten Transformation erstellen. In dieser Variante erfolgt die Abfrage der GML-Daten nicht über spezifische SQL-Statements, sondern über die ErgoAI-Syntax wie wir es bereits von der direkten Transformation kennen. Das entwickelte Datenmodell kann als Basis für beide Zugriffsvarianten über die SQL-Schnittstelle verwendet werden.

11.1 Generisches relationales Datenmodell für GML-Daten

Das Datenmodell basiert auf der Annahme, dass sich GML-Daten an das Object-Property-Modell halten. Das heißt, ein Objekt darf als Kind-Element nicht direkt ein weiteres Objekt enthalten, sondern muss ein Property-Element dazwischen geschachtelt haben und umgekehrt. Das generische relationale Datenmodell wird in Abbildung 11.1 dargestellt.

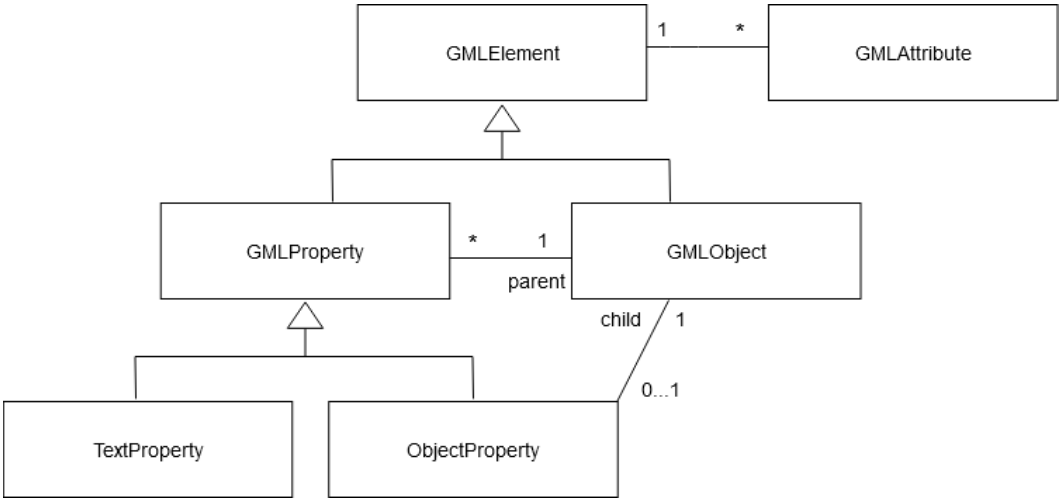


Abbildung 11.1: Generisches Datenmodell für GML-Daten

GMLElement ist die generische Superklasse von GML-Objekten und GML-Properties. Jedes GML-Element erhält als Surrogatschlüssel eine ID bestehend aus dem Dokumentnamen des ursprünglichen XML-Dokuments und einer fortlaufenden Nummer (Nummerierung innerhalb eines Dokuments). Diese Nummer wird von GML-Objekten und GML-Properties referenziert, um die Verbindung zu GML-Attributen herstellen zu können. Die Klasse dient zur Vereinfachung des Datenmodells, da sowohl GML-Objekte, als auch GML-Properties Attribute enthalten können. Aufgrund der unterschiedlichen Primärschlüssel müssten die Attributstabellen für GML-Objekte und GML-Properties aufgespaltet werden. Durch die Einführung der generischen Superklasse kann jedoch auf den Surrogatschlüssel der Superklasse anstatt auf die Primärschlüssel der spezifischen Klassen referenziert werden. Tabelle 11.1 zeigt eine Übersicht und Kurzbeschreibung der notwendigen Spalten für die GMLElement-Tabelle (der Primärschlüssel der folgenden Tabellen ist immer hervorgehoben).

Tabelle 11.1: GMLElement-Tabelle

Spalte	**Datentyp**	**Beschreibung**
element_id	VARCHAR2	Surrogatschlüssel für GML-Objekte und GML-Properties

GMLObject ist die Hauptklasse für GML-Features. Ein GML-Objekt kann mehrere Text-Properties und maximal ein Objekt-Property beinhalten. Der Primärschlüssel setzt sich durch den Namen des Ursprungsdokuments und seine eigene ID zusammen. Die `document_id` wird benötigt, um GML-Objekte ihrem Ursprungsdokument zuordnen zu können. Die IDs von GML-Objekten kommen aus dem `gml:id` Attribut (häufig auf Basis von UUIDv4). Der Namespace bezeichnet den XML-Namespace des GML-Objekts aus dem Ursprungsdokument. Der Objektname wird aus dem XML-Elementnamen übernommen und beinhaltet die Art des Elements (z.B. AIXMBasicMessage,

Event, EventTimeSlice, TimePeriod).Tabelle 11.2 zeigt eine Übersicht und Kurzbeschreibung der notwendigen Spalten für die GMLObject-Tabelle.

Tabelle 11.2: GMLObject-Tabelle

Spalte	**Datentyp**	**Beschreibung**
document_id	VARCHAR2	Name des Ursprungsdokuments eines GML-Objekts
object_id	VARCHAR2	Global eindeutige ID eines GML-Objekts
element_id	VARCHAR2	ID des generischen GML-Elements
object_ns	VARCHAR2	Namespace des GML-Objekts
object_name	VARCHAR2	Name des GML-Objekts (XML-Elementname)

GMLProperty ist die Hauptklasse für Properties eines GML-Objekts. Diese werden immer genau einem GML-Objekt zugeordnet. Properties können entweder einfache Text-Properties sein oder weitere GML-Objekte beinhalten. Um diese Mehrdeutigkeit in einem relationalen Schema abbilden zu können, wurden für beide Fälle eigene Tabellen angelegt. Die Tabelle `Text-Property` beinhaltet sämtliche textuellen Properties und die Tabelle `Objekt-Property` beinhaltet sämtliche Properties, die ein GML-Objekt geschachtelt haben. Tabelle 11.3 und Tabelle 11.4 zeigen eine Übersicht und Kurzbeschreibung der notwendigen Spalten für diese Tabellen.

Tabelle 11.3: Text-Property-Tabelle

Spalte	**Datentyp**	**Beschreibung**
parent_document_id	VARCHAR2	Name des Ursprungsdokuments des übergeordneten GML-Objekts
parent_id	VARCHAR2	Global eindeutige ID des übergeordneten GML-Objekts
property_name	VARCHAR2	Name des Properties (XML-Elementname)
property_index	INTEGER	Zähler für die Anzahl von Properties mit gleicher ID (des Eltern-Elements) innerhalb einer hierarchischen Ebene
property_ns	VARCHAR2	Namespace des Properties
element_id	VARCHAR2	ID des generischen GML-Elements
property_value	VARCHAR2	Textueller inhalt des Properties
property_datatype	VARCHAR2	Datentyp des Inhalts des Properties

Um GML-Properties eindeutig identifizieren zu können, wird der Primärschüssel des übergeordneten GML-Objekts Teil des Primärschlüssels des Properties. Das Property selbst kann über seinen Namen, seinen Index und seinen Namespace innerhalb eines GML-Objekts eindeutig identifiziert werden. Der Property-Index wird als Teil des Primärschlüssels benötigt, da innerhalb eines GML-Objekts mehrere Properties mit gleichem Namen und gleichem Namespace vorkommen können. Der

Tabelle 11.4: Objekt-Property-Tabelle

Spalte	Datentyp	Beschreibung
parent_document_id	VARCHAR2	Name des Ursprungsdokuments des übergeordneten GML-Objekts
parent_id	VARCHAR2	Global eindeutige ID des übergeordneten GML-Objekts
property_name	VARCHAR2	Name des Properties (XML-Elementname)
property_index	INTEGER	Zähler für die Anzahl von Properties mit gleicher ID (des Eltern-Elements) innerhalb einer hierarchischen Ebene
property_ns	VARCHAR2	Namespace des Properties
element_id	VARCHAR2	ID des generischen GML-Elements
object_document_id	VARCHAR2	Name des Ursprungsdokuments des geschachtelten GML-Objekts
object_id	VARCHAR2	Global eindeutige ID des geschachtelten GML-Objekts

Property-Namespace wird als Teil des Primärschlüssels benötigt, da innerhalb eines GML-Objekts mehrere Properties mit gleichem Namen vorkommen können, die jedoch aus unterschiedlichen Namespaces stammen.

Der gesamte Primärschlüssel der beiden Property-Tabellen setzt sich somit aus dem Primärschlüssel des übergeordneten GML-Objekts und den notwendigen Spalten zur eindeutigen Identifikation des Properties selbst zusammen. Bei Properties, die ein GML-Objekt geschachtelt haben, muss eine eindeutige Referenz auf das GML-Objekt gegeben sein. Diese Referenz muss wiederum aus dem Primärschlüssel der GMLObject-Tabelle bestehen, damit das Objekt eindeutig identifiziert werden kann. Textuelle Properties enthalten statt einer Objektreferenz direkt den textuellen Inhalt des Properties. Außerdem wird bei textuellen Properties der Datentyp des Werts abgespeichert um später Vergleichsoperationen in ErgoAI zu ermöglichen. Die Ermittlung der Datentypen erfolgt analog zur direkten Transformation über Parsingversuche.

Die GMLAttribute-Tabelle wird dazu verwendet, eventuell vorhandene XML-Attribute von GML-Objekten und GML-Properties zu speichern. Um eine Zuordnug zu den Objekten und Properties zu ermöglichen, wird auf den Surrogatschlüssel der GMLElement-Tabelle referenziert, von wo aus dann eine Verbindung zu den zugehörigen GML-Objekten bzw. GML-Properties erfolgen kann. Die Attribute selbst können über ihren Namen und ihren Namespace eindeutig identifiziert werden. Der gesamte Primärschlüssel der GMLAttribute-Tabelle setzt sich somit aus dem Surrogatschlüssel des zugehörigen GMLElements und den Spalten, die zur eindeutigen Identifizierung des Attributs selbst notwendig sind, zusammen. Eine Index-Spalte ist hier nicht notwendig, da laut XML-Spezifikation Attribute mit gleichem Namen und gleichem Namespace innerhalb eines Elements nicht mehrfach vorkommen können. Der GML-Standard sieht bis auf wenige Ausnahmen die Angabe eines Namespaces bei Attributen nicht vor. Für Attribute ohne Namespace wird daher ein Defaultwert eingefügt. Tabelle 11.5 zeigt eine Übersicht und Kurzbeschreibung der notwendigen Spalten für die GMLAttribute-Tabelle.

Tabelle 11.5: GMLAttribute-Tabelle

Spalte	Datentyp	Beschreibung
attribute_name	VARCHAR2	Name des Attributs (XML-Attributname)
attribute_ns	VARCHAR2	Namespace des Attributs
element_id	VARCHAR2	ID des generischen GML-Elements
attribute_value	VARCHAR2	Textueller Inhalt des Attributs

Da in GML-Dokumenten auch Namespaces mit unterschiedlichen Namen vorkommen können, die jedoch auf denselben Namespace verweisen, wurde für den Abgleich eine eigene Namespace-Tabelle angelegt. Diese ist in Abbildung 11.2 dargestellt. Die Identifikation von identischen Namespaces mit unterschiedlichen Namen erfolgt anhand der URI der Namespaces. Die Namespace-Prüfung wurde in den Mapping-Algorithmus integriert und erfolgt für jedes XML-Element des ursprünglichen GML-Dokuments. Für jedes Element wird vor dem Einfügen in das relationale GML-Datenmodell dessen Namespace gegen die Namespace-Tabelle geprüft.

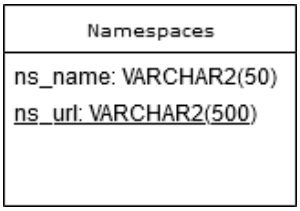


Abbildung 11.2: Namespace-Tabelle

Die Namespace-Tabelle wurde mit einigen Namespaces, die häufig in GML-Dokumenten vorkommen, manuell vorbefüllt. Ist in der Namespace-Tabelle bereits ein Namespace mit der gleichen URI wie die des Namespaces des aktuell behandelten Elements vorhanden, wird der bereits gespeicherte Namespace aus der Namespace-Tabelle für das Einfügen des Elements in der Datenbank herangezogen. Ist der Namespace noch nicht in der Namespace-Tabelle vorhanden, wird dieser in die Namespace-Tabelle eingefügt und für die Speicherung des aktuellen Elements verwendet. Die Namespace-Tabelle wird somit laufend automatisch beim Einfügen von neuen GML-Elementen erweitert.

11.2 Transformation von GML-Daten in ein relationales Datenmodell

Die Transformation von GML-Daten in ein relationales Datenmodell wurde ebenfalls in Java entwickelt. Die GML-Dokumente werden zunächst in ein Document Object Model (DOM) geparst und auf Basis dieses Models werden dann Insert-Statements für das Einfügen in die entsprechenden Datenbanktabellen erzeugt. Für das Parsen der GML-Dokumente wurden folgende APIs verwendet:

- javax.xml.*
- org.w3c.dom.*
- org.xml.sax.*
- com.sun.org.apache.xerces.*

Die Transformation wurde als rekursiver Algorithmus implementiert, der das geparste Document Object Model des GML-Dokuments durchläuft. Der Ausgangspunkt ist immer ein GML-Objekt. Die Unterscheidung zwischen Objekt- und Property-Knoten ergibt sich automatisch aufgrund der aktuellen Rekursionsebene, da sich Objekte und Properties immer abwechseln. In beiden Fällen werden außerdem zugehörige Attribute berücksichtigt, falls diese vorhanden sind.

Aufgrund von Foreignkey-Constraints des Datenmodells, spielt die Reihenfolge beim Einfügen eine Rolle. GML-Objekte und Text-Properties können immer sofort eingefügt werden. Objekt-Properties können erst eingefügt werden, wenn zuvor das referenzierte Kind-GML-Objekt eingefügt wurde, da Objekt-Properties den Primärschlüssel des GML-Objekts referenzieren. Bevor Attribute eingefügt werden können, müssen die zugehörigen GML-Objekte bzw. GML-Properties in der Datenbank vorhanden sein.

Abbildung 11.3 zeigt den Ablauf der Transformation. Die Abbildung zeigt das Einfügen von GML-Objekten und GML-Properties. Diese wurden als eigene UML-Aktivitätsdiagramme modelliert, die sich gegenseitig aufrufen. Der Ablauf startet immer in der Aktivität „GML-Objekt einfügen“. Das GML-Objekt wird zunächst in die Datenbank eingefügt und anschließend werden dessen Attribute eingefügt, falls diese vorhanden sind. GML-Objekte haben im Normalfall Properties und somit Kind-Knoten. Falls dies der Fall ist, werden die GML-Properties des GML-Objekts an die Aktivität „GML-Property einfügen“ übergeben und die Transformation geht in die nächste Rekursionsebene. Beim Einfügen von GML-Properties wird zunächst ermittelt, ob es sich um ein Text-Property oder Objekt-Property handelt. Text-Properties können sofort in die Datenbank eingefügt werden. Bei Objekt-Properties muss zuvor die gesamte Rekursion für das einzufügende Kind-GML-Objekt durchlaufen werden, bevor das Objekt-Property eingefügt werden kann. Dafür wird die Aktivität „GML-Objekt einfügen“ aufgerufen. In beiden Fällen werden anschließend die Attribute der GML-Properties eingefügt, wenn diese vorhanden sind. Die Rekursion endet bei Text-Properties, da diese nur einen textuellen Wert enthalten und keine weiteren GML-Objekte mehr geschachtelt haben können.

Die Transformation beinhaltet außerdem einige Prüfungen und Behandlungen für Sonderfälle. Die Prüfung von gleichen Namespaces und die automatische Erweiterung der Namespace-Tabelle wurde bereits erwähnt. Da Attribute in GML im Normalfall keinen Namespace besitzen, wird für diese Attribute der Default-Namespace 'none' eingefügt. Außerdem wird das Vorhandensein des `gml:id` Attributs bei GML-Objekten geprüft. Falls dieses nicht vorhanden ist, wird mit Hilfe von UUIDv4 eine neue Objekt-ID zur Laufzeit erzeugt.

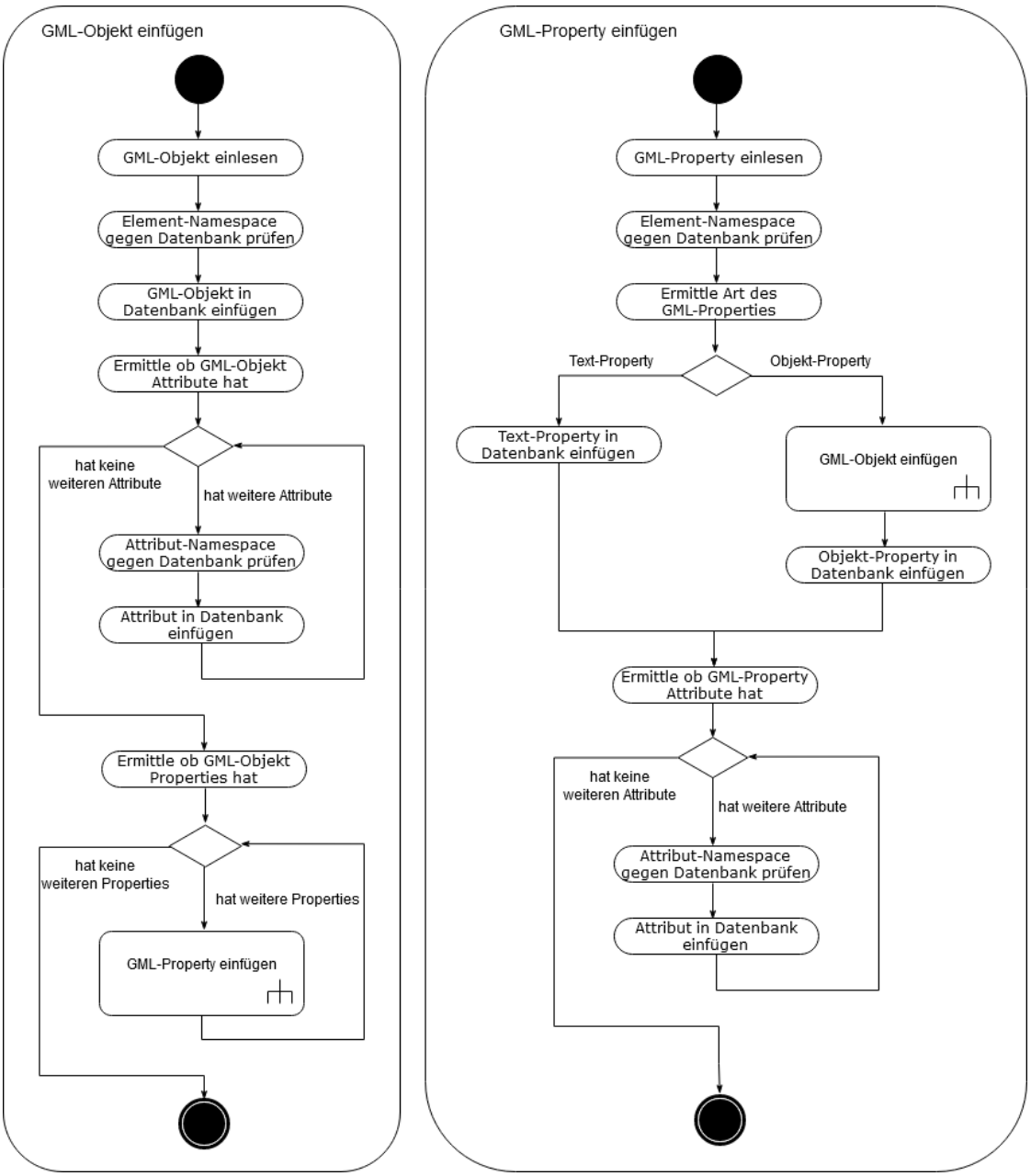


Abbildung 11.3: Ablauf der Transformation von GML-Dokumenten in relationales Schema

11.3 Variante 1: Ad-Hoc SQL-Abfragen auf GML-Daten in der SQL-Datenbank

Bei dieser Variante werden spezifische SQL-Abfragen für eine bestimmte Problemstellung (z.B. alle AIXM-Nachrichten, die ein Bodenservice betreffen) geschrieben und in ErgoAI-Abfragen eingebettet. Der Vorteil dabei ist, dass nicht die gesamte Wissensdatenbank aus der relationalen Datenbank nach ErgoAI geladen werden muss. Es wird immer nur das jeweilige ResultSet an ErgoAI zurückgegeben, das gerade für die jeweilige Problemstellung relevant ist. Der Nachteil bei dieser Variante ist, dass für jede Problemstellung eine eigene SQL-Abfrage geschrieben werden muss und eventuell bereits existierende Abfragen in der ErgoAI-Syntax dafür nicht direkt übernommen werden können.

Damit aus ErgoAI Daten aus einer SQL-Datenbank abgefragt werden können, muss zunächst über

die SQL-Schnittstelle eine Datenbankverbindung etabliert werden. Für die Verbindung muss als erster Parameter ein frei wählbarer Name vergeben werden, auf den später Abfragen referenzieren können. Als zweiter Parameter muss ein Datenquellenname angegeben werden. Wie dieser unter Windows erstellt werden kann, wird im Anhang erläutert. Die letzten beiden Parameter sind der Datenbankbenutzer und das Passwort, mit dem die Verbindung hergestellt werden soll. Der notwendige Befehl zur Herstellung der Datenbankverbindung wird in Listing 11.1 dargestellt.

Listing 11.1: Datenbankverbindung über SQL-Schnittstelle in ErgoAI herstellen

```
1 ?- odbc[open(con,GML12x64,gml,'gml')]@\sql.
```

Sobald die Verbindung aufgebaut ist, können Abfragen über diese gestellt werden. Für die Abfragen kann ebenfalls ein frei wählbarer Name vergeben werden. Anschließend muss die Abfrage selbst als Query-String übergeben werden. Einfache Hochkommas in den Abfragen müssen durch eine Escape-Sequenz (\) gekennzeichnet werden, damit ErgoAI diese verarbeiten kann. Das Ergebnis der Abfrage kann dann an ErgoAI-Variablen gebunden werden (Reihenfolge wie in der Abfrage definiert). Die ad-Hoc Abfrage von GML-Daten über die SQL-Schnittstelle wird im Folgenden anhand der gleichen Abfragen veranschaulicht, die auch in Kapitel 10.2 verwendet wurden.

AIXM-Nachrichten, die ein bestimmtes Feature (z.B. AircraftGroundService) betreffen

Diese Abfrage ähnelt von der Komplexität her der direkten Abfrage der in ErgoAI geladenen Dokumente. Die Einschränkung des Wurzelelements auf AIXM-Nachrichten und die Einschränkung auf einen bestimmten Featuretyp erfolgt in diesem Fall direkt in der SQL-Abfrage. Die analoge Abfrage über die SQL-Schnittstelle ist in Listing 11.2 dargestellt.

Listing 11.2: Abfrage von Nachrichten zu Bodenservices über die ErgoAI SQL-Schnittstelle

```
1  ?- con[query(aircraftgroundservices,['
2  SELECT g1.document_id, g1.object_id ,
3  g2.object_id services_id, g2.object_name type
4  FROM GMLOBJECT g1, GMLOBJECT g2, OBJECTPROPERTY p1
5  WHERE
6  g1.object_name = \'AIXMBasicMessage\' and
7  p1.parent_id = g1.object_id and
8  p1.parent_document_id = g1.document_id and
9  p1.property_name = \'hasMember\' and
10 p1.object_id = g2.object_id and
11 p1.object_document_id = g2.document_id and
12 g2.object_name = \'AircraftGroundService\''],
13 [?DATEI_NACHRICHT,?ID_NACHRICHT,?SERVICES_ID,?TYPE])]@\sql.
```

Alle AIXM-Ereignisse, die gerade aktiv sind

Bei dieser Abfrage ist vor allem der Datentyp der Datumsfelder wichtig, da bei diesen auf einen bestimmten Zeitraum eingeschränkt werden muss. Bei der Abfrage über die SQL-Schnittstelle gibt es mehrere Möglichkeiten, um diese zeitliche Einschränkung umzusetzen. In der ersten Variante passiert die Einschränkung direkt in der SQL-Abfrage über Vergleichsoperatoren in der WHERE-Klausel. Der aktuelle Zeitstempel kann in Oracle über `CURRENT_TIMESTAMP` abgefragt werden. In diesem Fall enthält das ResultSet, das in ErgoAI ankommt, bereits die gefilterten Werte. Diese Abfrage ist in Listing 11.3 dargestellt.

Listing 11.3: Abfrage von aktiven AIXM-Ereignissen über die ErgoAI SQL-Schnittstelle (A)

```
?- con[query(active_events,['
SELECT g.document_id,g.object_id id_event,t1.property_value beginn,
t2.property_value ende
FROM GMLOBJECT g0,GMLOBJECT g,OBJECTPROPERTY p0,OBJECTPROPERTY p1,GMLOBJECT g1,
OBJECTPROPERTY p2,GMLOBJECT g2,TEXTPROPERTY t1,TEXTPROPERTY t2
WHERE
g0.object_name = \'AIXMBasicMessage\' and p0.parent_id = g0.object_id and
p0.parent_document_id = g0.document_id and
p0.property_name = \'hasMember\' and p0.object_id = g.object_id and
p0.object_document_id = g.document_id and
g.object_name = \'Event\' and p1.parent_id = g.object_id and
p1.parent_document_id = g.document_id and
p1.property_name = \'timeSlice\' and p1.object_id = g1.object_id and
p1.object_document_id = g1.document_id and
g1.object_name = \'EventTimeSlice\' and p2.parent_id = g1.object_id and
p2.parent_document_id = g1.document_id and
p2.property_name = \'validTime\' and p2.object_id = g2.object_id and
p2.object_document_id = g2.document_id and
g2.object_name = \'TimePeriod\' and t1.parent_id = g2.object_id and
t1.parent_document_id = g2.document_id and
t1.property_name = \'beginPosition\' and
to_timestamp(t1.property_value, \'YYYY-MM-DD"T"HH24:MI:SS"Z"\')
<= CURRENT_TIMESTAMP and
t2.parent_id = g2.object_id and
t2.parent_document_id = g2.document_id and
t2.property_name = \'endPosition\' and
to_timestamp(t2.property_value, \'YYYY-MM-DD"T"HH24:MI:SS"Z"\')
>= CURRENT_TIMESTAMP'],
[?DATEI_EVENT,?ID_EVENT,?BEGINN,?ENDE])]@\sql.
```

In der zweiten Variante passiert die zeitliche Filterung erst in ErgoAI. Das von der SQL-Datenbank an ErgoAI übermittelte ResultSet enthält somit noch alle Ereignisse, die in der Datenbank gespeichert sind, unabhängig von zeitlichen Kriterien. Für die Filterung müssen die Datumswerte in ErgoAI jedoch zunächst in den \dateTime Datentyp umgewandelt werden, da sämtliche Werte in der Datenbank nur als Strings gespeichert werden. Dies erfolgt über die integrierte \dateTime[toType(\symbol) => \dateTime] Methode. Um die Datentypsicherheit zu gewährleisten, kann der Datentyp des jeweiligen Feldes ebenfalls abgefragt werden. In ErgoAI kann dann über ein If-Then-Else Konstrukt sichergestellt werden, dass der String-Wert des Datumsfeldes nur dann konvertiert wird, wenn es sich dabei auch um einen gültigen Datumsstring von Typ \dateTime handelt. Diese Variante wird in Listing 11.4 dargestellt.

Listing 11.4: Abfrage von aktiven AIXM-Ereignissen über die ErgoAI SQL-Schnittstelle (B)

```
?- con[query(active_events,['
SELECT g.document_id datei_event,g.object_id id_event,t1.property_value as beginn,
t1.property_datatype as beginndt,t2.property_value as ende,
t2.property_datatype as endedt
FROM GMLOBJECT g0, GMLOBJECT g, OBJECTPROPERTY p0, OBJECTPROPERTY p1, GMLOBJECT g1,
OBJECTPROPERTY p2, GMLOBJECT g2, TEXTPROPERTY t1, TEXTPROPERTY t2
WHERE
g0.object_name = \'AIXMBasicMessage\' and p0.parent_id = g0.object_id and
p0.parent_document_id = g0.document_id and
p0.property_name = \'hasMember\' and p0.object_id = g.object_id and
p0.object_document_id = g.document_id and
g.object_name = \'Event\' and p1.parent_id = g.object_id and
p1.parent_document_id = g.document_id and
p1.property_name = \'timeSlice\' and p1.object_id = g1.object_id and
p1.object_document_id = g1.document_id and
g1.object_name = \'EventTimeSlice\' and p2.parent_id = g1.object_id and
p2.parent_document_id = g1.document_id and
p2.property_name = \'validTime\' and p2.object_id = g2.object_id and
p2.object_document_id = g2.document_id and
g2.object_name = \'TimePeriod\' and t1.parent_id = g2.object_id and
t1.parent_document_id = g2.document_id and
t1.property_name = \'beginPosition\' and t2.parent_id = g2.object_id and
t2.parent_document_id = g2.document_id and
t2.property_name = \'endPosition\''],
[?DATEI_EVENT,?ID_EVENT,?_BEGINNSTRING, ?_BEGINNTYPE, ?_ENDESTRING,
?_ENDETYPE])]@\sql,
\dateTime[now -> ?_NOW]@\basetype,
\if (?_BEGINNTYPE[equals('^^\\dateTime')]@\basetype)
\then (\dateTime[toType(?_BEGINNSTRING) -> ?BEGINN]@\basetype)
\else (?BEGINN = "9999-12-31T00:00:00"^^\dateTime),
\if (?_ENDETYPE[equals('^^\\dateTime')]@\basetype)
\then (\dateTime[toType(?_ENDESTRING) -> ?ENDE]@\basetype)
\else (?ENDE = "9999-12-31T00:00:00"^^\dateTime),
?_NOW[lessEq(?ENDE)]@\basetype,
?BEGINN[lessEq(?_NOW)]@\basetype.
```

In beiden Varianten muss jedoch innerhalb der SQL-Abfrage zunächst zu den relevanten Feldern navigiert werden. Aufgrund der generischen Speicherung anhand des Object-Property-Modells in der Datenbank passiert das hauptsächlich durch abwechselnde Verknüpfung der Objekt- und Propertytabellen und durch Einschränkung auf bestimmte Elemente (z.B. AIXMBasicMessage, Event, timeSlice) in der WHERE-Klausel der Abfrage. Aufgrund dieser Tatsache können, je nachdem wie tief (in der ursprünglichen XML-Hierarchie) navigiert werden muss, unter Umständen sehr lange Abfragen mit vielen Joins notwendig sein. Bei dieser Abfrage wird unabhängig von der gewählten Variante bereits ersichtlich, dass diese über die SQL-Schnittstelle im Vergleich zur direkten Abfrage in ErgoAI deutlich länger wird.

Aktuelle Wetter- und Windverhältnisse an einem bestimmten Flughafen

Für diese Abfrage muss im SQL-Statement zunächst zum designator Feld des aerodrome Elements navigiert werden, um auf den Flughafen einschränken zu können. Des Weiteren muss zu der meteorologischen Beobachtung navigiert werden, um die Temperatur- und Windverhältnisse ermitteln zu können.

Listing 11.5: Abfrage von Wetter- und Windverhältnissen am Flughafen Karlsbad über die ErgoAI SQL-Schnittstelle

```
?- con[query(weather,['SELECT t1.property_value designator,
t2.property_value as airt, a1.attribute_value as airtuom,
t3.property_value as dewt, a2.attribute_value as dewtuom,
t4.property_value as mwd, a3.attribute_value as mwduom,
t5.property_value as mws, a4.attribute_value as mwsuom
FROM GMLOBJECT g,OBJECTPROPERTY p1,GMLOBJECT g1,OBJECTPROPERTY p2,GMLOBJECT g2,
TEXTPROPERTY t1,OBJECTPROPERTY p3,GMLOBJECT g3,TEXTPROPERTY t2, TEXTPROPERTY t3,
GMLATTRIBUTE a1,GMLATTRIBUTE a2,OBJECTPROPERTY p4, GMLOBJECT g4, TEXTPROPERTY t4,
TEXTPROPERTY t5, GMLATTRIBUTE a3, GMLATTRIBUTE a4
WHERE g.object_name = \'METAR\' and
p1.parent_id = g.object_id and p1.parent_document_id = g.document_id and
p1.property_name = \'aerodrome\' and
p1.object_id = g1.object_id and p1.object_document_id = g1.document_id and
g1.object_name = \'AirportHeliport\' and
p2.parent_id = g1.object_id and p2.parent_document_id = g1.document_id and
p2.property_name = \'timeSlice\' and
p2.object_id = g2.object_id and p2.object_document_id = g2.document_id and
g2.object_name = \'AirportHeliportTimeSlice\' and
t1.parent_id = g2.object_id and t1.parent_document_id = g2.document_id and
t1.property_name = \'designator\' and t1.property_value = \'LKKV\' and
p3.parent_id = g.object_id and p3.parent_document_id = g.document_id and
p3.property_name = \'observation\' and
p3.object_id = g3.object_id and p3.object_document_id = g3.document_id and
g3.object_name = \'MeteorologicalAerodromeObservation\' and
t2.parent_id = g3.object_id and t2.parent_document_id = g3.document_id and
t2.property_name = \'airTemperature\' and
a1.element_id = t2.element_id and a1.attribute_name = \'uom\' and
t3.parent_id = g3.object_id and t3.parent_document_id = g3.document_id and
t3.property_name = \'dewpointTemperature\' and
a2.element_id = t3.element_id and a2.attribute_name = \'uom\' and
p4.parent_id = g3.object_id and p4.parent_document_id = g3.document_id and
p4.property_name = \'surfaceWind\' and
p4.object_id = g4.object_id and p4.object_document_id = g4.document_id and
g4.object_name = \'AerodromeSurfaceWind\' and
t4.parent_id = g4.object_id and t4.parent_document_id = g4.document_id and
t4.property_name = \'meanWindDirection\' and
a3.element_id = t4.element_id and a3.attribute_name = \'uom\' and
t5.parent_id = g4.object_id and t5.parent_document_id = g4.document_id and
t5.property_name = \'meanWindSpeed\' and
a4.element_id = t5.element_id and a4.attribute_name = \'uom\''],
[?DESIGNATOR,?AIRT,?AIRTUOM,?DEWT,?DEWTUOM,?MWD,?MWDUOM,?MWS,?MWSUOM])]@\sql.
```

Um die Maßeinheiten zu ermitteln, muss zusätzlich eine Verknüpfung zu den Attributstabellen hergestellt werden, da diese in den Ursprungsdokumenten als Attribute angegeben sind. Die Verknüpfung zu den Attributstabellen kann jedoch über die zusätzlich eingeführte `element_id` erfolgen. Da in dieser Abfrage zu mehreren GML-Properties navigiert werden muss, die in mehreren Hierarchieebenen geschachtelt sind, wird trotz der vereinfachten Verknüpfung der Attribute bereits ersichtlich, dass Abfragen über die SQL-Schnittstelle schon bei verhältnismäßig einfachen Problemstellungen sehr lange und unübersichtlich werden können.

11.4 Variante 2: Generische SQL-Abfragen für die Erstellung von ErgoAI-Fakten

Bei dieser Variante werden generische SQL-Statements geschrieben, mit denen aus den GML-Daten aus der SQL-Datenbank Fakten in ErgoAI erstellt werden. Die SQL-Statements für die Erstellung der ErgoAI-Fakten wurden in einer solchen Weise geschrieben, dass die Abfrage auf die GML-Daten in ErgoAI analog zur Abfrage bei der direkten Transformation gestellt werden können. Da die gleichen Abfragen aus der direkten Transformation in dieser Variante verwendet werden können, werden diese hier nicht nochmal angeführt. Das Abfrageergebnis ist das gleiche wie bei der direkten Transformation. Der Vorteil dieser Variante ist, dass keine spezifischen SQL-Abfragen mehr geschrieben werden müssen, sobald die generischen SQL-Statements für das Laden der GML-Daten entwickelt wurden (und das ist ein einmaliger Prozess). Der Nachteil im Vergleich zu Variante 1 ist, dass dabei die gesamten GML-Daten in ErgoAI geladen werden müssen, sofern man die gesamte Wissensdatenbank berücksichtigen möchte. Durch eine Anpassung der SQL-Statements, mit einer Einschränkung auf bestimmte Dateinamen, könnten hier aber auch nur bestimmte Dokumente geladen werden. Die Anlage der Fakten in ErgoAI erfolgt über non-transactional Insert-Statements, wodurch die generierten Fakten direkt in der Wissensdatenbank gespeichert werden können. Non-transactional Statements werden in ErgoAI immer geschrieben, auch wenn folgende Updates eventuell fehlschlagen (ähnlich wie die Autocommit Funktionalität in SQL-Datenbanken). Non-Transactional Insert-Statements haben in ErgoAI die folgende Form:

```
insop {literals [| query ]}
```

Im Folgenden wird beispielhaft anhand von Text-Properties demonstriert, wie das generische Laden von GML-Objekten, GML-Properties und deren Attributen aus der SQL-Datenbank umgesetzt werden kann. Die gesamte Umsetzung findet man im Dokument `SQLGenericLoad.ergo`. Die in Kapitel 10.2 vorgestellten Regeln zur Erzeugung von Hilfsfakten zur vereinfachten Navigation wurden auch in dieser Variante hinzugefügt und können somit analog zur direkten Transformation verwendet werden. Um die Text-Properties in ErgoAI anzulegen, sind folgende Schritte notwendig:

- Anlage der ErgoAI-Objekte für die Text-Properties
- Referenzierung der Text-Properties im jeweiligen ErgoAI-Objekt des übergeordneten GML-Objekts
- Ermittlung der Attribute der Text-Properties

Für die Erstellung der ErgoAI-Objekte der Text-Properties wird wie bei der direkten Transformation der Function Term `qpid()` verwendet, der für die Vergabe von IDs von Properties definiert wurde.

Der Schlüssel (Dateiname und gml:id des übergeordneten GML-Objekts und Namespace, Name und Index des Properties) für die eindeutige Identifizierung des Text-Properties ist der gleiche wie bei der direkten Transformation und muss aus der SQL-Datenbank ermittelt werden. Das erstellte ErgoAI-Objekt bekommt den textuellen Wert des Text-Properties als value Attribut gesetzt.

Eine Besonderheit bei Text-Properties ist, dass hier auch der Datentyp des Inhalts ermittelt werden muss. Diese Information ist in der SQL-Datenbank in der Spalte property_datatype vorhanden. Um den Textinhalt mit dem korrekten Datentyp zu laden, muss die entsprechende Parsingfunktion (toType) des Datentyps in ErgoAI aufgerufen werden. Diese Funktion steht für alle primitiven Datentypen, die auch durch das Transformationsprogramm in das relationale Datenmodell unterstützt werden, zur Verfügung. Über die Datentypinformation in der SQL-Datenbank und über ein if-then Konstrukt in ErgoAI kann die Datentypsicherheit gewährleistet werden.

Listing 11.6: Abfrage von Text-Properties über die SQL-Schnittstelle

```
?- con[query(load_textproperty,['
SELECT t.parent_document_id, t.parent_id, t.property_name,
t.property_index, t.property_ns, t.property_value, t.property_datatype
FROM textproperty t
'],[?DATEI_PARENT,?ID_PARENT,?PROPERTY_NAME,?PROPERTY_INDEX,?PROPERTY_NS,
?PROPERTY_VALUE,?PROPERTY_DATATYPE])]@\sql,
insert{textproperty(?DATEI_PARENT,?ID_PARENT,?PROPERTY_NAME,?PROPERTY_INDEX,
?PROPERTY_NS,?PROPERTY_VALUE,?PROPERTY_DATATYPE)}.
```

Durch das Parsing in einen Datentyp wird aus einer Variable eine typisierte Variable. Da an typisierte Variablen nur noch Werte dieses Datentyps gebunden werden können, muss hierfür ein Hilfsprädikat verwendet werden. Im ersten Schritt werden die Daten der Text-Properties über die SQL-Schnittstelle abgefragt und zunächst noch mit Wert und Datentyp getrennt in das Hilfsprädikat geladen, das in Listing 11.6 dargestellt ist. Im zweiten Schritt wird dann erst das Parsing in den korrekten Datentyp durchgeführt und das ErgoAI-Objekt für das Text-Property erstellt. Dadurch muss die Abfrage auf die GML-Daten in der SQL-Datenbank nur einmal erfolgen. Das dafür notwendige Insert-Statement ist in Listing 11.7 dargestellt.

Listing 11.7: Erstellung von Text-Properties über die SQL-Schnittstelle (Objekt)

```
qpid (?DATEI_PARENT,?ID_PARENT,?PROPERTY_NS,?PROPERTY_NAME,?PROPERTY_INDEX)
[value ->?TYPEDVALUE] :-
textproperty(?DATEI_PARENT,?ID_PARENT,?PROPERTY_NAME,?PROPERTY_INDEX,
?PROPERTY_NS,?PROPERTY_VALUE,?PROPERTY_DATATYPE),
\if (?PROPERTY_DATATYPE[equals('^^\\boolean')]@\basetype)
  \then (\boolean[toType(?PROPERTY_VALUE) -> ?TYPEDVALUE]@\basetype),
<\if() \then() Statements fuer uebrige Datentypen>.
```

Nachdem die ErgoAI-Objekte für die Text-Properties erstellt wurden, müssen diese in den ErgoAI-Objekten ihrer übergeordneten GML-Objekte referenziert werden. Dabei werden Fakten für die GML-Objekte mit den Referenzen auf die Text-Properties erzeugt. Die Identifizierung der GML-Objekte erfolgt wieder über den Dateinamen und ihre gml:id mit Hilfe des qoid() Function Terms. Diese Informationen sind in den Property-Tabellen bereits vorhanden, da sie auch Teil des Schlüssels von Properties sind. Die Referenzierung erfolgt über den Namespace und Namen des

Properties (Function Term cn()), die auf die ID des im vorigen Schritt erstellten ErgoAI-Objekts des Text-Properties verweisen. Das dafür notwendige Insert-Statement ist in Listing 11.8 dargestellt.

Listing 11.8: Erstellung von Text-Properties über die SQL-Schnittstelle (Referenz)

```
?- con[query(load_textproperty_reference,['
SELECT t.parent_document_id, t.parent_id, t.property_name,
t.property_index, t.property_ns
FROM textproperty t
'],[?DATEI_PARENT,?ID_PARENT,?PROPERTY_NAME,?PROPERTY_INDEX,?PROPERTY_NS])]@\sql,
insert{qoid (?DATEI_PARENT,?ID_PARENT)[cn (?PROPERTY_NS,?PROPERTY_NAME) ->
qpid (?DATEI_PARENT,?ID_PARENT,?PROPERTY_NS,?PROPERTY_NAME,?PROPERTY_INDEX)]}.
```

Abschließend müssen noch die Attribute zu den Text-Properties ermittelt werden. Dabei wird in den Text-Properties für jedes Attribut ein Fakt erzeugt, der den Wert des Attributs enthält. Hierfür wird wieder der Function Term cn() verwendet. Hier ist aber zu beachten, dass Attribute laut GML-Standard bis auf wenige Ausnahmen ohne Namespace angegeben werden. Es müssen in ErgoAI aber beide Fälle abgedeckt werden. Bei der direkten Transformation wurde dies direkt im Transformationsprogramm durch die unterschiedliche Parametrisierung (mit und ohne Namespace) des cn() Function Terms umgesetzt.

Listing 11.9: Erstellung von Text-Properties über die SQL-Schnittstelle (Attribute)

```
?- con[query(load_textproperty_attributes,['
SELECT t.parent_document_id, t.parent_id, t.property_name,
t.property_index, t.property_ns, a.attribute_ns, a.attribute_name,
a.attribute_value
FROM textproperty t, gmlattribute a
WHERE t.element_id = a.element_id AND
a.attribute_ns <> \'none\'
'],[?DATEI_PARENT,?ID_PARENT,?PROPERTY_NAME,?PROPERTY_INDEX,?PROPERTY_NS,
?ATTRIBUTE_NS,?ATTRIBUTE_NAME,?ATTRIBUTE_VALUE])]@\sql,
insert{qpid (?DATEI_PARENT,?ID_PARENT,?PROPERTY_NS,?PROPERTY_NAME,?PROPERTY_INDEX)
[attribute (cn (?ATTRIBUTE_NS,?ATTRIBUTE_NAME)) -> ?ATTRIBUTE_VALUE]}.
```

Da für die Speicherung von Attributen ohne Namespace in der SQL-Datenbank der Platzhalter 'none' vergeben wurde, kann dieser Mechanismus mit zwei Insert-Statements und entsprechenden SQL-Abfragen mit und ohne Namespace in dieser Variante ebenfalls nachgestellt werden (das Beispiel enthält nur die Variante mit Namespace). Das dafür notwendige Insert-Statement ist in Listing 11.9 dargestellt.

Kapitel 12

Zugriffsmöglichkeit 3: Direkter Zugriff auf GML-Dokumente

Bei dieser Variante wird der in ErgoAI integrierte XML-Parser für das Laden der GML-Dokumente verwendet. Da der ErgoAI XML-Parser die Anforderungen und Problemstellungen dieser Arbeit nicht vollständig erfüllen bzw. lösen kann, wurde diese Zugriffsmöglichkeit nicht vollständig implementiert. Es wird stattdessen die Funktionalität dex XML-Parsers anhand eines Beispieldokuments veranschaulicht und auf dessen Unzulänglichkeiten hingewiesen. Das Laden von XML-Dokumenten erfolgt in ErgoAI über folgenden Befehl:

```
?InDoc[load_xml(?Module) -> ?Warn]@\xml
```

Der Parameter `?InDoc` gibt den Pfad zum XML-Dokument in Form eines Dateipfades oder einer Url an. Der Parameter `?Module` gibt an in welches Modul das Dokument geladen werden soll und der Parameter `?Warn` ist ein Objekt, in das ggf. anfallende Warnungen gespeichert werden.

Listing 12.1: XML-Parser Beispieldokument

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <message:AIXMBasicMessage
   xmlns:message="http://www.aixm.aero/schema/5.1.1/message"
   xmlns:gml="http://www.opengis.net/gml/3.2"
3  xmlns:aixm="http://www.aixm.aero/schema/5.1.1"
   xmlns:event="http://www.aixm.aero/schema/5.1/event" gml:id="M1">
4      <message:hasMember>
5          <event:Event gml:id="uuid.6fdfbe36-0e6e-4058-9bca-f2275c97e6c8">
6              <gml:identifier codeSpace="urn:uuid:">
7              6fdfbe36-0e6e-4058-9bca-f2275c97e6c8
8           </gml:identifier>
9              <event:timeSlice>
10                 <event:EventTimeSlice gml:id="ID_HDL_E011">
11                     <aixm:interpretation>BASELINE</aixm:interpretation>
12                 </event:EventTimeSlice>
13             </event:timeSlice>
14         </event:Event>
15     </message:hasMember>
16 </message:AIXMBasicMessage>
```

Als Beispieldokument wird die AIXM-Nachricht `Parser_Test.xml` verwendet. Dieses Dokument beinhaltet einen vereinfachten Ausschnitt einer Nachricht zur vorübergehenden Nichtverfügbarkeit eines Bodenservices für Flugzeuge. Dieser Ausschnitt ist in Listing 12.1 dargestellt. Der Befehl für das Laden dieses Dokuments sieht (abhängig vom Speicherort) folgendermaßen aus:

```
?- 'C:\\OracleData\\transformXML\\Parser_Test.xml'[load_xml(main) -> ?Warn]@\xml.
```

Der XML-Parser führt dabei eigenständig eine Transformation in korrespondierende ErgoAI-Objekte durch. Dieser Prozess kann nur in beschränktem Umfang über Parametrisierung beeinflusst werden und der Systementwickler hat somit kaum Kontrolle über die erstellten ErgoAI-Objekte. Ein wesentlicher Punkt ist hier die Navigierbarkeit zwischen den erstellten ErgoAI-Objekten. Hierfür sieht der ErgoAI XML-Parser neben dem Standardmodus die Möglichkeit vor, Navigationslinks zwischen den ErgoAI-Objekten anfordern zu können. Im Standardmodus ist die Navigation einfacher und es werden weniger Objekte angelegt, dafür ist aber die Navigierbarkeit eingeschränkt (z.B. kein Link auf das Eltern-Element). Im Modus mit Navigationslinks erzeugt der XML-Parser zusätzliche Navigationsattribute, mit denen eine erweiterte Navigation in der XML-Hierarchie möglich ist. Dadurch wird aber eine Vielzahl an Fakten generiert, die Ressourcen benötigen und die Wissensdatenbank unübersichtlich machen. Die generierten ErgoAI-Fakten können über folgende Abfrage ermittelt werden:

```
?_X[?_Y->?_Z]@main, ?Z = $ {?_X[?_Y->?_Z]}
```

Der generierte Output wird für den XML-Parser und die selbst entwickelte Transformation in Listing 12.2 und Listing 12.3 gegenübergestellt. Im Modus ohne Navigationslinks werden für dieses Dokument durch den XML-Parser 17 Fakten erzeugt, bei der Transformation durch das selbst entwickelte Transformationsporgramm (siehe Kapitel 10) sind es 19 Fakten (ohne Regeln für einfachere Navigation). Die Anzahl der generierten Fakten ist annähernd gleich, der Unterschied ergibt sich nur durch das explizite Anführen des Elementtyps von GML-Objekten bei der selbst entwickelten Transformation. Im Modus mit Navigationslinks würde der ErgoAI XML-Parser für dieses kurze Beispieldokument bereits 103 Fakten erzeugen, die zusätzliche Ressourcen beanspruchen und in vielen Fällen überhaupt nicht benötigt werden. Wenn man die abgeleiteten Fakten für die einfachere Navigation bei der selbst entwickelten Transformation noch mitberücksichtigt, sind es für dieses Beispieldokument im Vergleich nur 23 Fakten.

Der größte Vorteil der selbst entwickelten Transformation im Vergleich zum ErgoAI XML-Parser ist, dass diese das Object-Property-Modell berücksichtigt. Erst dadurch erreicht man die Flexibilität einer kompakten Navigation bei einer gleichzeitigen Repräsentation der GML-Objekte, GML-Properties und deren Attribute mit verhältnismäßig wenigen Fakten. Durch die Einführung von zusätzlichen Navigationsregeln zum Überspringen von Properties kann so bei vielen Abfragen eine vereinfachte Navigation erreicht werden, wie es bereits in Kapitel 10.2 verdeutlicht wurde. Bei der generischen Transformation durch den XML-Parser ist die Definition von solchen Regeln in einer sinnvollen Weise nicht möglich, da dieser nicht zwischen GML-Objekten und GML-Properties unterscheidet.

Listing 12.2: XML-Parser Beispieldokument Output durch direkte Transformation

```
qoid('Parser_Test',M1)[
   elementtype->cn(message,AIXMBasicMessage),
   attribute(cn(gml,id))->M1,
   attribute(cn(xmlns,aixm))->'http://www.aixm.aero/schema/5.1.1',
   attribute(cn(xmlns,event))->'http://www.aixm.aero/schema/5.1/event',
   attribute(cn(xmlns,gml))->'http://www.opengis.net/gml/3.2',
   attribute(cn(xmlns,message))->'http://www.aixm.aero/schema/5.1.1/message',
   cn(message,hasMember)->qpid('Parser_Test',M1,message,hasMember,1),
   op(cn(message,hasMember))->
   qoid('Parser_Test','uuid.6fdfbe36-0e6e-4058-9bca-f2275c97e6c8')
].

qpid('Parser_Test',M1,message,hasMember,1)[
   cn(event,Event)->qoid('Parser_Test','uuid.6fdfbe36-0e6e-4058-9bca-f2275c97e6c8')
].

qoid('Parser_Test','uuid.6fdfbe36-0e6e-4058-9bca-f2275c97e6c8')[
   elementtype->cn(event,Event),
   attribute(cn(gml,id))->'uuid.6fdfbe36-0e6e-4058-9bca-f2275c97e6c8',
   cn(gml,identifier)->
   qpid('Parser_Test','uuid.6fdfbe36-0e6e-4058-9bca-f2275c97e6c8',gml,identifier,1),
   tp(cn(gml,identifier))->"6fdfbe36-0e6e-4058-9bca-f2275c97e6c8"^^\string,
   cn(event,timeSlice)->
   qpid('Parser_Test','uuid.6fdfbe36-0e6e-4058-9bca-f2275c97e6c8',event,timeSlice,1),
   op(cn(event,timeSlice))->qoid('Parser_Test',ID_HDL_E011)
].

qpid('Parser_Test','uuid.6fdfbe36-0e6e-4058-9bca-f2275c97e6c8',gml,identifier,1)[
   attribute(cn(codeSpace))->'urn:uuid:',
   value->"6fdfbe36-0e6e-4058-9bca-f2275c97e6c8"^^\string
].

qpid('Parser_Test','uuid.6fdfbe36-0e6e-4058-9bca-f2275c97e6c8',event,timeSlice,1)[
   cn(event,EventTimeSlice)->qoid('Parser_Test',ID_HDL_E011)
].

qoid('Parser_Test',ID_HDL_E011)[
   elementtype->cn(event,EventTimeSlice),
   attribute(cn(gml,id))->ID_HDL_E011]@main,
   cn(aixm,interpretation)->qpid('Parser_Test',ID_HDL_E011,aixm,interpretation,1),
   tp(cn(aixm,interpretation))->"BASELINE"^^\string
].

qpid('Parser_Test',ID_HDL_E011,aixm,interpretation,1)[
   value->"BASELINE"^^\string
].
```

Listing 12.3: XML-Parser Beispieldokument Output durch Parser

```
obj1[
   'message:AIXMBasicMessage'->obj2
].

obj2[
   attribute('xmlns:message')->'http://www.aixm.aero/schema/5.1.1/message',
   attribute('xmlns:gml')->'http://www.opengis.net/gml/3.2',
   attribute('xmlns:aixm')->'http://www.aixm.aero/schema/5.1.1',
   attribute('xmlns:event')->'http://www.aixm.aero/schema/5.1/event',
   attribute('gml:id')->M1,
   'message:hasMember'->obj3
].

obj3[
   'event:Event'->obj4
].

obj4[
   attribute('gml:id')->'uuid.6fdfbe36-0e6e-4058-9bca-f2275c97e6c8',
   'gml:identifier'->obj5,
   'event:timeSlice'->obj6
].

obj5[
   attribute(codeSpace)->'urn:uuid:',
   \text->'6fdfbe36-0e6e-4058-9bca-f2275c97e6c8'
].

obj6[
   'event:EventTimeSlice'->obj7
].

obj7[
   attribute('gml:id')->ID_HDL_E011,
   'aixm:interpretation'->obj8
].

obj8[
   \text->BASELINE
].
```

Die selbst entwickelte Transformation mag aufgrund der langen UUIDs von GML-Objekten auf den ersten Blick unübersichtlich erscheinen, der Vorteil der einfacheren Navigation wird aber bei der Formulierung von Abfragen deutlich. Um dies zu veranschaulichen, soll in dem angeführten Beispieldokument zur AIXM-Nachricht die Interpretation des EventTimeSlices ausgelesen werden. Im Ergebnis sollen die ID der Nachricht, die ID des Events und die Interpretation des EventTimeSlices aufscheinen. Die dafür notwendigen Abfragen sind in Listing 12.4 und Listing 12.5 dargestellt.

Listing 12.4: Ermittlung der TimeSlice Interpretation bei selbst entwickelter Transformation

```
?- qoid(?_DAT_MSG,?ID_MSG)[
   elementtype -> cn(?_MTNS,?_MSGTYPE),
   op(cn(?_MNS,hasMember))->qoid(?_DAT_EVT,?ID_EVT)[
      elementtype -> cn(?_MEMNS,?_MEMTYPE),
      op(cn(?_EVTSNS,timeSlice))->?_EVTSO[
         tp(cn(?_INS,interpretation))->?INTERPRETATION
      ]
   ]
],?_MSGTYPE[equals(AIXMBasicMessage)]@\basetype,
?_MEMTYPE[equals(Event)]@\basetype.
```

Hier wird ersichtlich, dass die Navigation bei der selbst entwickelten Transformation schon in einem einfachen Beispiel deutlich kürzer wird. Dieser Effekt verstärkt sich, je tiefer in die Hierarchie des Ursprungsdokuments navigiert werden muss. Abgesehen von der längeren Navigation ist es beim XML-Parser auch ein Problem, dass dieser die Namespaces nicht separat speichert. In diesem verkürzten Beispiel ist das nicht relevant, da hier gezielt auf EventTimeSlices eingeschränkt wird. AIXM-Nachrichten können aber auch noch weitere Mitglieder haben und es kann auch Situationen geben, in denen man nicht nur auf eine bestimmte Art von TimeSlice einschränken möchte. In Situationen, in denen der Namespace keine Rolle spielen soll, muss beispielsweise mit Regular Expression gearbeitet werden, um den Namespace zu ignorieren.

Listing 12.5: Ermittlung der TimeSlice Interpretation bei Transformation durch XML-Parser

```
?- ?_OBJ1[
   'message:AIXMBasicMessage'->?_OBJ2[
      attribute('gml:id')->?ID_MSG,
      'message:hasMember'->?_OBJ3[
         'event:Event'->?_OBJ4[
            attribute('gml:id')->?ID_EVT,
            ?_P2->?_TS[
               ?_P3->?_OBJ5[
                  'aixm:interpretation'->?_OBJ6[
                     \text->?INTERPRETATION
                  ]
               ]
            ]
         ]
      ]
   ]
].
```

Außerdem generiert der XML-Parser für die Identifizierung der ErgoAI-Objekte generische und nicht aussagekräftige IDs der Form obj + <Laufnummer>. Die gml:id wird zwar als Attribut zu GML-Objekten gespeichert, aber nicht für die Identifizierung verwendet, wie es der GML-Standard vorsieht. In diesem Beispiel wird auch ersichtlich, dass durch den XML-Parser die Verbindung zum Ursprungsdokument verloren geht, da dieses nicht mitgespeichert wird.

Ein weiteres Problem ist, dass beim XML-Parser die Datentypinformation verloren geht. XML-Elemente mit textuellem Inhalt werden vom XML-Parser lediglich als Textattribut (gekennzeichnet durch `\text`) gespeichert. Sämtliche Abfragen, bei denen Datentypinformationen der Textwerte (die nicht tatsächlich nur Strings sind) benötigt werden, sind bei der Transformation durch den integrierten XML-Parser somit nicht möglich. Die Abfrage 'Alle AIXM-Ereignisse, die gerade aktiv sind' könnte in dieser Variante beispielsweise nicht durchgeführt werden, da hier eine Einschränkung auf Datumswerte erfolgen muss.

Kapitel 13

Evaluierung

Dieses Kapitel behandelt die Evaluierung der Anforderungen und Ziele, die in Kapitel 8 für die Umsetzung der Zugriffsmöglichkeiten definiert wurden. Die Erfüllung der Anforderungen wird für jeden Ansatz gesondert betrachtet und die Ansätze werden anhand der Anforderungserfüllung verglichen. Für die Erreichung der Ziele werden die Performance der Zugriffsmöglichkeiten, sowie die Wartbarkeit der Transformationsprogramme und die Benutzerfreundlichkeit bei der Formulierung von Abfragen betrachtet, wo es relevant ist.

13.1 Anforderungen

In diesem Abschnitt wird die Erfüllung der Anforderungen für jede Zugriffsmöglichkeit diskutiert und gegenübergestellt. Tabelle 13.1 zeigt eine Gegenüberstellung der Zugriffsmöglichkeiten anhand der in Kapitel 8.1 definierten Anforderungen.

Tabelle 13.1: Evaluierung der Anforderungen der Zugriffsmöglichkeiten

Anforderung	**Z1**	**Z2 (V1)**	**Z2 (V2)**	**Z3**
A1 - Das Ursprungsdokument muss vollständig abgebildet werden	✓	✓	✓	✗
A1.1 - Es muss zu sämtlichen Elementen des Ursprungsdokuments navigiert werden können	✓	✓	✓	✓
A1.2 - Namespaces müssen berücksichtigt werden	✓	✓	✓	✓
A1.3 - Datentypen der Werte müssen ermittelt werden können	✓	✓	✓	✗
A2 - Es muss eine Zuordnung der transformierten XML-Elemente zu ihren Ursprungsdokumenten möglich sein	✓	✓	✓	✗
A3 - Die Transformation muss generisch sein	✓	✓	✓	✓
A4 - Die Transformation muss schemalos sein	✓	✓	✓	✓

Zugriffsmöglichkeit 1: Transformation in ErgoAI-Fakten und Laden in ErgoAI

Bei der direkten Transformation der GML-Daten in ErgoAI-Fakten können alle Anforderungen erfüllt werden. Das ursprüngliche GML-Dokument wird direkt in ErgoAI vollständig abgebildet. GML-Objekte und GML-Properties werden dabei als eigene ErgoAI-Objekte abgebildet. Durch die Vergabe von eindeutigen Schlüsseln können GML-Objekte und GML-Properties aufeinander referenzieren und es kann zwischen diesen navigiert werden. Die Attribute werden bei den GML-Objekten bzw. GML-Properties direkt gespeichert. Die Namespaces werden bei der Transformation berücksichtigt und werden mit Hilfe von Function Terms in ErgoAI separat vom Namen des Elements gespeichert, damit diese auch wieder einzeln abgefragt werden können. Die Datentypen von GML-Properties mit textuellem Inhalt werden bei der Transformation über Parsing-Versuche ermittelt und der ErgoAI-Syntax entsprechend an die Werte angehängt. Durch die Einbeziehung des Dateinamens in die IDs von GML-Objekten und GML-Properties können diese ihrem Ursprungsdokument eindeutig zugeordnet werden. Die Transformation funktioniert generisch für alle GML-Dokumente unabhängig von domänenspezifischen Applikationsschemas, da es auf dem Object-Property-Modell aufbaut, welches dem GML-Standard zugrunde liegt. Die Transformation ist schemalos, da sie rein auf Dokumentenebene arbeitet und ohne XML-Schemainformationen auskommt.

Zugriffsmöglichkeit 2: Transformation in relationales Datenmodell und Zugriff über SQL-Schnittstelle

Bei der Transformation in ein relationales Datenmodell und beim Zugriff über die ErgoAI SQL-Schnittstelle können ebenfalls alle Anforderungen in beiden Varianten erfüllt werden. Das entwickelte relationale Datenmodell für die Speicherung der GML-Daten in der SQL-Datenbank liegt beiden Varianten zugrunde. In diesem Datenmodell kann das Ursprungsdokument vollständig abgebildet werden. GML-Objekte, GML-Properties (Objekt- und Text-Properties getrennt) und deren Attribute werden in eigenen Tabellen gespeichert. GML-Objekte und GML-Properties können über die gleichen Schlüssel wie bei der direkten Transformation miteinander verknüpft werden. Zur Vereinfachung des Datenmodells wurde ein alternativer Schlüssel (`element_id`) erzeugt, um GML-Objekte und GML-Properties einfacher mit ihren Attributen verknüpfen zu können. Für die Speicherung des Namespaces wurden in den Objekt-,Property- und Attributstabellen eigene Spalten eingefügt. Die Ermittlung der Datentypen erfolgt direkt bei der Transformation in das relationale Datenmodell und wird bei Text-Properties in einer eigenen Datentyp-Spalte gespeichert. Der Dateiname des Ursprungsdokuments wird bei GML-Objekten und GML-Properties ebenfalls in einer eigenen Spalte gespeichert und ist auch jeweils Teil des Schlüssels.

Diese Informationen können in der ersten Variante über entsprechende SQL-Statements wieder abgefragt werden. In der zweiten Variante werden diese Informationen in den generischen SQL-Statements verwendet, welche die entsprechenden ErgoAI-Objekte analog zur direkten Transformation erzeugen. Die Transformation in das relationale Datenmodell ist generisch, da das relationale Datenmodell anhand des Object-Property-Modells erstellt wurde. Dadurch ist auch in Variante 2 die Erstellung der korrespondierenden ErgoAI-Objekte generisch, da diese auf dem relationalen Datenmodell basiert. Die Transformation in das relationale Datenmodell ist auch schemalos, da sie wie die direkte Transformation nur auf Dokumentenebene arbeitet und ohne XML-Schemainformationen auskommt.

Zugriffsmöglichkeit 3: Direkter Zugriff auf GML-Dokumente

Beim direkten Laden der GML-Dokumente durch den integrierten ErgoAI XML-Parser können nicht alle Anforderungen erfüllt werden. Der XML-Parser generiert zwar eindeutige IDs für die erstellten ErgoAI-Objekte und ermöglicht dadurch eine eindeutige Referenzierung und Navigation zwischen den Objekten, es geht dabei allerdings die Zuordnung zu den Ursprungsdokumenten verloren. Außerdem werden sämtliche textuellen Werte auch nur als String gespeichert und die Datentypinformationen bei Text-Properties gehen dadurch verloren. Somit können keine Abfragen durchgeführt werden, bei denen die Datentypinformationen benötigt werden. Die Namespaces werden vom XML-Parser berücksichtigt, allerdings werden diese unverändert aus XML übernommen und gemeinsam mit dem Namen des Elements gespeichert. Das ist im Vergleich zu den anderen Ansätzen nicht optimal, da in Abfragen, bei denen eine separate Behandlung des Namespaces gewünscht oder erforderlich ist, mit eingebauten String-Funktionen (z.B. Substring oder Regular Expressions) gearbeitet werden muss. Die Transformation und das Laden der GML-Dokumente erfolgt durch den XML-Parser in einem Schritt. In diesen Prozess hat man als Benutzer und Entwickler nur beschränkte Einsicht und Eingriffsmöglichkeiten. Die Transformation ist allerdings insofern generisch und schemalos, da der XML-Parser beliebige XML-Dokumente verarbeiten kann und aus diesen generische ErgoAI-Objekte ohne die Verwendung von XML-Schemainformationen erstellt.

13.2 Ziele

In diesem Kapitel wird die Erreichung der Ziele, die in Kapitel 8.2 definiert wurden, für jeden Ansatz evaluiert und gegenübergestellt. Die definierten Ziele gliedern sich in die Bereiche Performanceziele, Wartbarkeit der Transformationsprogramme und Benutzerfreundlichkeit bei der Formulierung von Abfragen.

13.2.1 Performance

In diesem Abschnitt wird die Evaluierung der Performance für jede Zugriffsmöglichkeit vorgestellt. Die Messung der Performance muss jeweils für folgende Teilaspekte des Datenzugriffs separat betrachtet werden:

- Transformation
- Laden
- Abfrage

Da nicht jeder Teilaspekt für jede Zugriffsmöglichkeit relevant ist, zeigt Tabelle 13.2 eine Gegenüberstellung, welcher Teilaspekt bei der Messung der Performance für die jeweilige Zugriffsmöglichkeit berücksichtigt wurde. Bei Zugriffsmöglichkeit 3 wird durch den XML-Parser die Transformation und das Laden in einem Schritt durchgeführt. Da die Teilaspekte in diesem Fall nicht separat gemessen werden können und der Zugriff über einen Ladebefehl in ErgoAI erfolgt, wurde dieser Schritt dem Laden zugeordnet. Bei der Zugriffsmöglichkeit 2 Variante 1 werden die GML-Daten nicht in die Wissensdatenbank geladen, sondern immer nur spezifische SQL-Abfragen gestellt und ein spezifisches ResultSet zurückgegeben, daher ist hier das Laden nicht relevant.

Tabelle 13.2: Berücksichtigung von Teilaspekten für die Messung der Performance

	Z1	**Z2 (V1)**	**Z2 (V2)**	**Z3**
Transformation	✓	✓	✓	✗
Laden	✓	✗	✓	✓
Abfrage	✓	✓	✓	✓

Um die Performancemessung für alle Zugriffsmöglichkeiten vergleichbar zu machen, müssen diese auf derselben Grundlage erfolgen. Für die Performancemessung des Transformations- und Ladeschritts wurde ein Testdatenset bestehend aus AIXM-Nachrichten (37 Dokumente) und IWXXM-Nachrichten (26 Dokumente) zusammengestellt. Die AIXM-Nachrichten wurden dem fiktiven DONLON Datenset (https://github.com/aixm/donlon) entnommen und die IWXXM-Nachrichten wurden aus einem Repository des WMO Informationsmanagements (https://github.com/wmo-im/iwxxm/tree/master/IWXXM/examples) entnommen. Für die Performancemessung der Abfragen wurden die gleichen Beispielabfragen verwendet, die auch in den Umsetzungskapiteln zur Veranschaulichung verwendet wurden:

- **Abfrage 1:** AIXM-Nachrichten, die ein bestimmtes Feature (z.B. AircraftGroundService) betreffen
- **Abfrage 2:** Alle AIXM Ereignisse, die gerade aktiv sind
- **Abfrage 3:** Aktuelle Wetter- und Windverhältnisse an einem bestimmten Flughafen (z.B. Flughafen Karlsbad)

Die Messung der Performance erfolgte auf einem Notebook mit 8GB Arbeitsspeicher (DDR4 SDRAM 2400MHz) und einem 6-Kern Prozessor (Intel i7-8750H 2,20Ghz). Es wurden für jeden Teilaspekt (Transformation, Laden, Abfrage) für jeden Ansatz jeweils 10 Durchgänge durchgeführt und es wurden während der Messung, soweit es möglich war, alle nicht benötigten Programme und Prozesse beendet, um äußere Einflussfaktoren zu eliminieren, welche die Messung beeinflussen könnten. Dort wo ein Zugriff auf die Oracle Datenbank notwendig war, ist zu berücksichtigen, dass diese auf dem selben System ausgeführt wurde. Es wurde für alle Durchgänge ein Kaltstart durchgeführt um eventuelle Verfälschungen der Ergebnisse durch Caching zu vermeiden. Für die Messung der Transformationszeiten wurde ein Timer in die entwickelten Transformationsprogramme eingebaut. Für die Messung der Lade- und Abfragezeiten wurden die in ErgoAI integrierte Laufzeitmessung über die ErgoAI IDE herangezogen. Bei der Messung der Ladezeiten wurde auch die Zeit für das Preprocessing und Kompilieren der Wissensdatenbank in ErgoAI berücksichtigt.

Tabelle 13.3: Ergebnis Performancemessung Transformation

	Avg(ms)	**Min(ms)**	**Max(ms)**
Zugriffsmöglichkeit 1	1045	991	1080
Zugriffsmöglichkeit 2	57734	56842	59887

Tabelle 13.4: Ergebnis Performancemessung Laden

	Avg(ms)	**Min(ms)**	**Max(ms)**
Zugriffsmöglichkeit 1	7589	7536	7624
Zugriffsmöglichkeit 2 (V2)	6261	5719	6900
Zugriffsmöglichkeit 3	4618	4543	4687

Tabelle 13.5: Ergebnis Performancemessung Abfrage 1

	Avg(ms)	**Min(ms)**	**Max(ms)**
Zugriffsmöglichkeit 1	67	65	71
Zugriffsmöglichkeit 2 (V1)	45	35	80
Zugriffsmöglichkeit 2 (V2)	65	60	78

Tabelle 13.6: Ergebnis Performancemessung Abfrage 2

	Avg(ms)	**Min(ms)**	**Max(ms)**
Zugriffsmöglichkeit 1	51	47	53
Zugriffsmöglichkeit 2 (V1)	97	93	109
Zugriffsmöglichkeit 2 (V2)	101	94	109

Tabelle 13.7: Ergebnis Performancemessung Abfrage 3

	Avg(ms)	**Min(ms)**	**Max(ms)**
Zugriffsmöglichkeit 1	10	9	11
Zugriffsmöglichkeit 2 (V1)	133	130	139
Zugriffsmöglichkeit 2 (V2)	43	31	47

An dieser Stelle sei generell angemerkt, dass die Performancemessungen im Rahmen dieser Arbeit nur eine grobe Vorstellung geben sollen, wie sich die Zugriffsmöglichkeiten hinsichtlich ihrer Performance unterscheiden. Die Performancemessungen wurden nur für eine Datengröße durchgeführt, nämlich mit der des Testdatensets. Die Skalierbarkeit für größere Datenmengen kann somit aufgrund dieser Messungen nicht eingeschätzt werden. Außerdem fand das komplette Testdatenset im Hauptspeicher Platz, weshalb die Unterschiede zwischen den Zugriffsmöglichkeiten nicht so signifikant ausfielen. Zudem wurden die Zugriffsmöglichkeiten, bei denen ein Datenbankzugriff über die ErgoAI SQL-Schnittstelle erfolgt, nur mit einem Datenbankmanagementsystem getestet. Bei diesen Zugriffsmöglichkeiten hängt die Performance jedoch von vielen Faktoren, wie dem verwendeten Datenbankmanagementsystem (z.B. Version, Konfiguration) an sich und dessen interner Abfrageverarbeitung (z.B. Query Optimizer, Indexstrukturen, Caching), ab.

Im Folgenden wird für jede Zugriffsmöglichkeit jeweils der Durchschnitt, der schnellste und der langsamste Durchlauf der 10 Durchgänge angegeben. Die Messergebnisse für die Transformation und das Laden des Testdatensets in den dabei relevanten Zugriffsmöglichkeiten sind in Tabelle 13.3 und Tabelle 13.4 dargestellt. Die Messung der Transformationszeiten für Zugriffsmöglichkeit 2 betrifft beide Varianten, da diese beide auf dem relationalen Datenmodell aufbauen und wird deshalb für beide Varianten zusammengefasst dargestellt. Die Ergebnisse für die Testabfragen sind in Tabellen 13.5, 13.6 und 13.7 dargestellt. Da die Zugriffmöglichkeit 3 aufgrund der unzureichenden Anforderungserfüllung nicht vollständig implementiert wurde, erfolgte für die Abfragen in dieser Variante keine Performancemessung.

Gesamt betrachtet, von den GML-Ursprungsdokumenten bis zum Abfrageergebnis, ist die Performance bei Zugriffsmöglichkeit 1 für alle drei Testabfragen am besten. Bei Zugriffsmöglichkeit 2 dauerte die Transformation des Testdatensets fast eine Minute im Vergleich zu ca. einer Sekunde bei Zugriffsmöglichkeit 1. Aus diesem Grund ist die Zugriffsmöglichkeit 2 gesamt betrachtet deutlich langsamer, auch wenn Abfrage 1 durch die Ad-Hoc Abfrage in Zugriffsmöglichkeit 2 Variante 1 im Durchschnitt schneller war als in Zugriffsmöglichkeit 1. Die Ladezeiten sind für Zugriffsmöglichkeit 1 und Zugriffsmöglichkeit 2 Variante 2 vergleichbar, wobei Zugriffsmöglichkeit 2 Variante 2 im Durchschnitt etwas schneller war. Bei den Abfragezeiten ist Zugriffsmöglichkeit 1 im Durchschnitt aber schneller als Zugriffsmöglichkeit 2 Variante 2. Nur bei Abfrage 1 wurden annähernd gleiche Zeiten erzielt. Bei Abfrage 1 und Abfrage 2 erzielten die beiden SQL-Varianten ähnliche Zeiten, bei Abfrage 3 war aber Variante 2 deutlich schneller als Variante 1. Auf die Gesamtperformance bezogen ist Zugriffsmöglichkeit 1 aus den genannten Gründen zu bevorzugen.

13.2.2 Wartbarkeit der Transformationsprogramme

In diesem Abschnitt wird die Code-Wartbarkeit der Transformationsprogramme diskutiert. Die Evaluierung bezieht sich auf die direkte Transformation in ErgoAI-Fakten (Zugriffsmöglichkeit 1) und auf die Transformation in ein relationales Schema (Zugriffsmöglichkeit 2). Die Hauptfunktionalität der beiden Transformationsprogramme wurde in jeweils einer Klasse implementiert (`XMLToErgoMapper` bzw. `XMLToRelationalMapper`).

Die Klasse `XMLToErgoMapper` enthält 529 Codezeilen und die Klasse `XMLToRelationalMapper` enthält 885 Codezeilen. Beim `XMLToRelationalMapper` wird für das Einfügen in die SQL-Datenbank im Transformationsprogramm eine Datenbankverbindung aufgebaut, die dafür notwendigen Insert-Statements sind als Klassenvariablen in der `XMLToRelationalMapper` Klasse eingebettet. Beim

`XMLToErgoMapper` wurden für die einfachere Konvertierung der GML-Daten in die ErgoAI-Syntax Java-Objekte (`GMLElement`, `GMLObject`, `GMLProperty`, `GMLObjekt-Property`, `GMLText-Property`, `GMLAttribute`) für die Repräsentation der ErgoAI-Objekte angelegt. Wo es möglich war wurde hier das Vererbungskonzept angewendet um Codeduplizierung zu vermeiden. Diese Klassen beinhalten zusätzlich zur `XMLToErgoMapper` Klasse 790 Codezeilen.

Die komplexeste Methode bei beiden Transformationsprogrammen stellt die `traverseRecursive` Methode dar, die die Hauptfunktionalität der Transformation implementiert. Diese Methode wurde als rekursiver Algorithmus implementiert, in dem das Document Object Model (DOM) der GML-Dokumente durchgegangen wird und die entsprechenden Operationen für die Transformation durchgeführt werden (siehe Kapitel 10.1 und Kapitel 11.2). Es wurde darauf geachtet, dass die Rekursion gut nachvollziehbar ist, indem die Abzweigungen und Abbruchbedingungen entsprechend kommentiert sind. Sämtliche nicht-öffentliche Methoden und Variablen enthalten eine kurze Beschreibung in Form von Java-Kommentaren. Für alle öffentlichen Methoden und Variablen wird eine entsprechende Java-Dokumentation (JavaDoc) bereitgestellt.

Für die Transformationsprogramme wurde außerdem eine Ordnerstruktur (Input, Log, Output) mit eigenen Ordnern für jeden Transformationsdurchgang (anhand eines TimeStamps) erstellt, um den Transformationsvorgang besser nachvollziehen zu können. Für das Logging wurde die in Java integrierte Logging-Funktionalität (`java.util.logging`) verwendet. Es wird für jede Datei eine Logdatei erstellt, die die Transformationsschritte und auftretende Fehler (Exceptions) mitprotokolliert. Außerdem wird für jeden Durchgang eine Gesamtdatei mit allen Fehlern über alle Dokumente des Durchgangs erstellt, um die Fehlersuche bzw. Fehlerbehebung zu erleichtern.

13.2.3 Benutzerfreundlichkeit bei der Formulierung von Abfragen

Bei den verwendeten Beispielabfragen wird bereits ersichtlich, dass die Navigation bei den Abfragen relativ langatmig werden kann. Besonders in Abfrage 3 wird ersichtlich, dass die Navigationswege sehr lange werden können, je tiefer in die Dokumentenstruktur navigiert werden muss. Um diesen Effekt abzufedern, wurden in Zugriffsmöglichkeit 1 und Zugriffsmöglichkeit 2 Variante 2 zusätzliche Navigationsregeln für das Überspringen der ErgoAI-Objekte von GML-Properties eingeführt. Diese können immer dann verwendet werden, wenn die Attribute der Properties für die Abfrage nicht relevant sind.

Beim Ad-Hoc Zugriff über spezifische SQL-Statements und bei der Transformation durch den ErgoAI XML-Parser können diese Regeln nicht umgesetzt werden. Hier leidet die Navigierbarkeit, was besonders bei den Abfragen 2 und 3 in der SQL-Variante 1 ersichtlich wird. Hier sind schon bei Abfragen auf verhältnismäßig hoher XML-Hierarchieebene viele Joins notwendig. Ein großer Vorteil von SQL-Variante 2 ist jedoch, dass die Abfragen im Vergleich zur direkten Transformation nicht angepasst werden müssen und trotzdem ein relationales Datenmodell als Grundlage für den Zugriff auf GML-Daten verwendet werden kann.

Für die Benutzerfreundlichkeit ist es außerdem wichtig, dass die Schlüssel der ErgoAI-Objekte atomar dargestellt werden. Das ist beispielsweise bei Abfragen nützlich, in den der Namespace keine Rolle spielen soll. In den selbst entwikelten Transformationen wurde darauf Rücksicht genommen. Beim XML-Parser ist das nicht der Fall und hier müssen teilweise in ErgoAI eingebaute Funktionen (z.B. Regular Expressions) verwendet werden, um die Schlüssel wieder zu trennen.

Kapitel 14

Fazit und Ausblick

In dieser Arbeit wurde gezeigt, dass der Zugriff auf GML-Daten auf ErgoAI auf verschiedene Wege umsetzbar ist. Es wurde gezeigt, dass eine generische Transformation von GML-Daten in ErgoAI-Fakten und in ein relationales Datenmodell anhand des Object-Property-Modells ohne Informationsverlust möglich ist. Die Transformation durch den integrierten ErgoAI XML-Parser war in diesem Bezug unzureichend, da bei diesem wichtige Informationen wie der Bezug zum Ursprungsdokument oder Datentypinformationen verloren gingen. Außerdem bietet die Berücksichtigung des Object-Property-Modells durch die selbst entwickelten Transformationsprogramme Vorteile gegenüber der Transformation durch den ErgoAI XML-Parser. Erst dadurch konnte eine kompakte Transformation durchgeführt werden, die in vielen Situationen einfachere Abfragen mit kürzeren Navigationswegen ermöglicht. Dieser Effekt wurde durch die Einführung von zusätzlichen Navigationsregeln für das Überspringen von GML-Properties verstärkt. Ohne die Berücksichtigung des Object-Property-Modells wäre die Definition dieser Regeln nicht möglich gewesen.

Es wurden auch die Vorteile der Verwendung eines relationalen Datenmodells gegenüber der direkten Transformation in ErgoAI-Fakten gezeigt. Durch spezifische Ad-Hoc Abfragen auf die SQL-Datenbank muss nicht die gesamte Wissensdatenbank in ErgoAI geladen werden. Dadurch können manche Abfragen schneller durchgeführt werden, als beim direkten Laden der gesamten Wissensdatenbank. Dieser Effekt wurde jedoch durch die langen Transformationszeiten im Vergleich zur direkten Transformation gedämpft. Hier besteht für die Zukunft sicher noch Verbesserungspotential. Wenn die Transformationszeiten in das relationale Datenmodell noch weiter reduziert werden können, könnte diese Zugriffsmöglichkeit besonders bei großen Wissensdatenbanken in vielen Situationen vorteilhafter sein als die direkte Transformation. Bei der Erstellung der ErgoAI-Fakten über die generischen SQL-Statements konnte eine ähnliche Performance für das Laden und auch teilweise ähnliche Performance bei den Abfragen beobachtet werden. Solange die Wissensdatenbank im Hauptspeicher Platz findet, unterscheiden sich die beiden Varianten für den Zugriff über die ErgoAI SQL-Schnittstelle somit nicht nennenswert. Sobald die Wissensdatenbank die Größe des verfügbaren Arbeitsspeichers überschreitet, werden jedoch spezifische Ad-Hoc Abfragen eine valide Option, um ohne zusätzliche Hardwareressourcen auszukommen. Alternativ könnten auch die generischen SQL-Statements angepasst werden, um jeweils nur die benötigten Teile der Wissensdatenbank in ErgoAI zu laden. Der Zugriff über die ErgoAI SQL-Schnittstelle ist auch dann eine valide Option, falls die GML-Daten nicht im XML-Format vorliegen, sondern aus bestimmten Gründen vorab in eine SQL-Datenbank geladen werden müssen oder sollen.

Der Fokus dieser Arbeit lag auf dem Zugriff auf GML-Daten, um diese für deduktive Datenbanksysteme basierend auf F-Logic oder dessen Erweiterungen zugänglich zu machen. Diese Arbeit bildet die Grundlage für die Erstellung von deduktiven Datenbanken, die geografische Daten auf Basis des GML-Standards verarbeiten können. Als durchgängiges Anwendungbeispiel wurden die GML-Applikationsschemas AIXM und IWXXM verwendet, um zu zeigen, dass die vorgestellten Zugriffsmöglichkeiten unabhängig von domänenspezifischen Applikationsschemas eingesetzt werden können. Dadurch können basierend auf diesen Zugriffsmöglichkeiten beispielsweise Systeme für die intelligente Filterung von Nachrichten im Flugverkehr oder generell Systeme entwickelt werden, die auf verschiedene GML-basierte domänenspezifische Datenmodelle zugreifen müssen.

Literatur

Abdelmoty, A. I., Williams, M. H. & Paton, N. W. (1993). Deduction and deductive databases for geographic data handling. *International Symposium on Spatial Databases*, 443–464.

Arni, F., Ong, K., Tsur, S., Wang, H. & Zaniolo, C. (2002). The Deductive Database System LDL++.

Bassiliades, N., Vlahavas, I. & Sampson, D. (2003). Using logic for querying XML data. *Web-Powered Databases* (S. 1–35). IGI Global.

Bohannon, P., Freire, J., Roy, P. & Siméon, J. (2002). From XML schema to relations: A cost-based approach to XML storage. *Proceedings 18th International Conference on Data Engineering*, 64–75.

Boucelma, O. & Colonna, F.-M. (2004). GQuery: a query language for GML. *24th Urban Data Management Symposium, UDMS 2004. Chioggia (Italie), 27–29 Octobre 2004*.

Cacace, F., Ceri, S., Crespi-Reghizzi, S., Fraternali, P., Paraboschi, S. & Tanca, L. (1993). The LOGRES prototype. *ACM SIGMOD Record*, *22*(2), 550–551.

Calejo, M. (2004). InterProlog: Towards a declarative embedding of logic programming in Java. *European Workshop on Logics in Artificial Intelligence*, 714–717.

Chaves, L. G., Filho, J. L. & Andrade, M. V. A. (2006). GML Documents : An approach for Database Schema Design and Storing. *iiWAS2006*, (January).

Córcoles, J. E. & González, P. (2001). A specification of a spatial query language over GML. *Proceedings of the 9th ACM international symposium on Advances in geographic information systems*, 112–117.

Corti, P., Kraft, T. J., Mather, S. V. & Park, B. (2014). *PostGIS cookbook*. Packt Publishing Ltd.

Davidson, S., Fan, W. & Hara, C. (2007). Propagating XML constraints to relations. *Journal of Computer and System Sciences*, *73*(3), 316–361.

Davis, J. R. (1998). IBM's DB2 spatial extender: Managing geo-spatial information within the DBMS. *IBM corporation, May*.

Deutsch, A., Fernandez, M. & Suciu, D. (1999). Storing semistructured data with STORED. *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, 431–442.

Egenhofer, M. J. (1994). Spatial SQL: A query and presentation language. *IEEE Transactions on knowledge and data engineering*, *6*(1), 86–95.

EUROCONTROL & Federal Aviation Administration. (2010). AIXM 5 - Temporality Model, 1–30.

EUROCONTROL & Federal Aviation Administration. (2011). AIXM 5 - Feature Identification and Reference, 1–18.

Faqir, A., Mahmood, A., Qazi, K. & Malik, S. (2019). An Approach to Map Geography Mark-up Language Data to Resource Description Framework Schema. *International Conference on Intelligent Technologies and Applications*, 343–354.

Fernandes, A. A., Dinn, A., Paton, N. W., Williams, M. H. & Liew, O. (1999). Extending a deductive object-oriented database system with spatial data handling facilities. *Information and Software Technology*, *41*(8), 483–497.

Florescu, D. & Kossmann, D. (1999). Storing and querying XML data using an RDMBS. *IEEE data engineering bulletin*, *22*, 3.

Green, C. C. & Raphael, B. (1968). The use of theorem-proving techniques in question-answering systems. *Proceedings of the 1968 23rd ACM national conference*, 169–181.

Han, J., Liu, L. & Xie, Z. (1994). LogicBase: a deductive database system prototype. *Proceedings of the third international conference on Information and knowledge management*, 226–233.

He, G., Shanmugasundaram, J., Tufte, K., Zhang, C., DeWitt, D. & Naughton, J. (2008). Relational databases for querying XML documents: Limitations and opportunities. *Proceedings of VLDB*, 302–314.

Hietanen, E., Lehto, L. & Latvala, P. (2016). Providing geographic datasets as linked data in SDI. *The International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, *41*, 583.

Holupirek, A., Grün, C. & Scholl, M. H. (2007). Visually Exploring and Querying XML with BaseX.

Hughes, J. N., Annex, A., Eichelberger, C. N., Fox, A., Hulbert, A. & Ronquest, M. (2015). Geomesa: a distributed architecture for spatio-temporal fusion. *Geospatial Informatics, Fusion, and Motion Video Analytics V*, *9473*, 94730F.

Hunter, J. & Wooldridge, M. (2016). *Inside MarkLogic Server*. MarkLogic.

International Civil Aviation Organisation. (2007). Annex 3 to the Convention on International Civil Aviation Meteorological Service for International Air Navigation, 1–187.

International Civil Aviation Organisation. (2012). *ICAO Meteorological Information Exchange Model*. Verfügbar 4. Juli 2020 unter http://www.wmo.int/schemas/iwxxm/3.0.0RC1/html/index.htm?goto=5

International Civil Aviation Organisation. (2019). International Civil Aviation Organisation guidelines for the implementation of OPMET data exchange using IWXXM, 1–48.

Jeung, H.-y. & Park, S.-h. (2004). A GML data storage method for spatial databases. *Journal of GIS Association of Korea*, *12*(4), 307–319.

Jiang, H., Lu, H., Wang, W. & Yu, J. X. (2002). Path Materialization Revisited: An Efficient Storage Model for XML Data. *Australasian Database Conference*, *5*.

Kappel, G., Kapsammer, E., Rausch-Schott, S. & Retschitzegger, W. (2000). X-ray-towards integrating XML and relational database systems. *International Conference on Conceptual Modeling*, 339–353.

Kellogg, C. & Travis, L. (1981). Reasoning with data in a deductively augmented data management system. *Advances in Data Base Theory* (S. 261–295). Springer.

Kifer, M. (2018a). A Guide to ErgoAI Packages, 1–86.

Kifer, M. (2018b). ErgoAI Reasoner User's Manual, 1–374.

Kim, W. (2001). XRel: a path-based approach to storage and retrieval of XML documents using relational databases. *ACM Transactions on Internet Technology (TOIT)*, *1*(1), 110–141.

Klettke, M. & Meyer, H. (1999). Managing XML documents in object-relational databases.

Koch, S. & Löwner, M.-O. (2017). Representation of CityGML instance models in BaseX. *Advances in 3D Geoinformation* (S. 63–78). Springer.

Kothuri, R., Godfrind, A. & Beinat, E. (2008). *Pro oracle spatial for oracle database 11g*. Dreamtech Press.

Krupa, K. A. (2005). System and method for converting an XML data structure into a relational database [US Patent 6,915,304].

Kyzirakos, K., Savva, D., Vlachopoulos, I., Vasileiou, A., Karalis, N., Koubarakis, M. & Manegold, S. (2018). GeoTriples: Transforming geospatial data into RDF graphs using R2RML and RML mappings. *Journal of Web Semantics*, *52*, 16–32.

Langran, G. (1992). States, events, and evidence: the principle entities of a temporal GIS. *GIS LIS-INTERNATIONAL CONFERENCE-*, *1*, 416–416.

Lee, D. & Chu, W. W. (2000). Constraints-preserving transformation from XML document type definition to relational schema. *International Conference on Conceptual Modeling*, 323–338.

Lee, D. & Chu, W. W. (2001). CPI: constraints-preserving inlining algorithm for mapping XML DTD to relational schema. *Data & Knowledge Engineering*, *39*(1), 3–25.

Li, Y., Li, J. & Zhou, S. (2004). GML storage: a spatial database approach. *International Conference on Conceptual Modeling*, 55–66.

Liu, Y., Zhong, H. & Wang, Y. (2004). Capturing XML constraints with relational schema. *The Fourth International Conference onComputer and Information Technology, 2004. CIT'04.*, 309–314.

Mao, B., Harrie, L., Cao, J., Wu, Z. & Shen, J. (2014). NoSQL based 3D city model management system. *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, *4*.

May, W. (2003). XPath-Logic and XPathLog: A logic-programming style XML data manipulation language. *arXiv preprint cs/0311038*.

Meier, W. (2002). eXist: An open source native XML database. *Net. ObjectDays: International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World*, 169–183.

Morris, K., Ullman, J. D. & Van Gelder, A. (1986). Design overview of the NAIL! system. *International Conference on Logic Programming*, 554–568.

Murthy, R. & Banerjee, S. (2003). Xml schemas in Oracle XML DB. *Proceedings 2003 VLDB Conference*, 1009–1018.

Nicola, M. & Kumar-Chatterjee, P. (2009). *DB2 pureXML cookbook: master the power of the IBM hybrid data server*. Pearson Education.

Nivalis, P. (2018). A Guide to Flora2 Packages, 1–35.

Open Geospatial Consortium et al. (1999). OpenGIS simple features specification for SQL.

Open Geospatial Consortium. (2006). The OpenGIS ® Abstract Specification Topic 6: Schema for coverage geometry and functions, 1–75.

Open Geospatial Consortium. (2018). *Geography Markup Language*. Verfügbar 10. November 2020 unter https://www.ogc.org/standards/gml

Perry, M. & Herring, J. (2012). OGC GeoSPARQL-A geographic query language for RDF data. *OGC implementation standard*, *40*.

Phipps, G., Derr, M. A. & Ross, K. A. (1991). Glue-Nail: A deductive database system. *Proceedings of the 1991 ACM SIGMOD international conference on Management of data*, 308–317.

Portele, C. (2007). OpenGIS® Geography Markup Language (GML) Encoding Standard.

PostgreSQL Global Development Group. (n. d.). *PostgreSQL 12.3 Documentation*. Verfügbar 18. Juli 2020 unter https://www.postgresql.org/docs/12/datatype-xml.html

Ramakrishnan, R., Srivastava, D., Sudarshan, S. & Seshadri, P. (1993). Implementation of the CORAL deductive database system. *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, 167–176.

Robare, P. (2003). Deductive database architecture for geographic data [US Patent 6,601,073].

Rys, M. (2005). XML and relational database management systems: inside Microsoft® SQL Server™ 2005. *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, 958–962.

Sagonas, K., Swift, T. & Warren, D. S. (1994). XSB as an efficient deductive database engine. *ACM SIGMOD Record*, *23*(2), 442–453.

Schmidt, A., Kersten, M., Windhouwer, M. & Waas, F. (2000). Efficient relational storage and retrieval of XML documents. *International Workshop on the World Wide Web and Databases*, 137–150.

Shimura, T., Yoshikawa, M. & Uemura, S. (1999). Storage and retrieval of XML documents using object-relational databases. *International Conference on Database and Expert Systems Applications*, 206–217.

Shrestha, B. B. (2004). *XML Database Technology and Its Use for GML.* (Magisterarb.). International Institute for Geo-information Science und Earth Observation (ITC).

Taranov, I., Shcheklein, I., Kalinin, A., Novak, L., Kuznetsov, S., Pastukhov, R., Boldakov, A., Turdakov, D., Antipin, K., Fomichev, A. et al. (2010). Sedna: native XML database management system (internals overview). *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 1037–1046.

Tatarinov, I., Viglas, S. D., Beyer, K., Shanmugasundaram, J., Shekita, E. & Zhang, C. (2002). Storing and querying ordered XML using a relational database system. *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, 204–215.

Tsur, S. & Zaniolo, C. (1986). LDL: a logic-based data-language. *VLDB*, *86*, 33–41.

Ullman, J. & Ramakrishnan, R. (1995). A survey of research in deductive database systems. *J. Logic Programming*, 125–149.

Vaghanl, J., Ramamohanarao, K., Kemp, D. B., Somogyi, Z., Stuckey, P. J., Leask, T. S. & Harland, J. (1994). The Aditi deductive database system. *The VLDB Journal*, *3*(2), 245–288.

Van den Brink, L., Janssen, P. & Quak, W. (2013). From geo-data to linked data: automated transformation from GML to RDF. *Linked Open Data-Pilot Linked Open Data Nederland. Deel 2-De Verdieping, Geonovum, 2013, pp. 249-261*.

Van den Brink, L., Janssen, P., Quak, W. & Stoter, J. (2014). Linking spatial data: automated conversion of geo-information models and GML data to RDF. *International Journal of Spatial Data Infrastructures Research*, *9*, 59–85.

Vaz, D., Ferreira, M. & Lopes, R. (2007). Spatial-yap: a logic-based geographic information system. *International Conference on Logic Programming*, 195–208.

Vieille, L., Bayer, P., Küchenhoff, V., Lefebvre, A. & Manthey, R. (1992). The EKS-V1 system. *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, 504–506.

Wang, F., Sha, J., Chen, H. & Yang, S. (2000). Geosql: A spatial query language of object-oriented gis. *Proceedings of the 2nd International Workshop on Computer Science and Informationn Technologies*, 215–219.

Widom, J. (1992). The Starburst rule system: Language design, implementation, and applications. *IEEE Data Engineering Bulletin*.

World Meteorological Organization. (n. d.). *WMO Codes Registry*. Verfügbar 4. Juli 2020 unter http://codes.wmo.int

World Meteorological Organization. (2014). *Guidelines on data modelling for WMO codes* [unpublished].

World Wide Web Consortium. (2006). *Namespaces in XML 1.1 (Second Edition)*. Verfügbar 4. Juli 2020 unter https://www.w3.org/TR/xml-names11/

World Wide Web Consortium. (2008). *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. Verfügbar 7. Juli 2020 unter https://www.w3.org/TR/2008/REC-xml-20081126/

Yang, G., Kifer, M., Wan, H. & Zhao, C. (2008). Flora-2: User's manual, 1–166.

Zhang, W., Ling, T. W., Chen, Z. & Dobbie, G. (2005). Xdo2: A deductive object-oriented query language for xml. *International Conference on Database Systems for Advanced Applications*, 311–322.

Zhang, Y., Wang, K. & Zhou, L. (2000). Spatial Deductive Database. *INTERNATIONAL ARCHIVES OF PHOTOGRAMMETRY AND REMOTE SENSING*, *33*(B4/3; PART 4), 1209–1215.

Zhu, F., Guan, J., Zhou, J. & Zhou, S. (2006). Storing and querying GML in object-relational databases. *Proceedings of the 14th annual ACM international symposium on Advances in geographic information systems*, 107–114.

Zhu, F., Guan, J. & Zhou, S. (2007). Constraints-preserving GML storage in object-relational databases. *Proceedings of the 15th annual ACM international symposium on Advances in geographic information systems*, 1–4.

Anhang

Installationsanleitung

Dies ist eine Anleitung zur Installation/Einrichtung der notwendigen Komponenten für den Prototyp des deduktiven Datenbanksystems für GML Daten in Microsoft Windows:

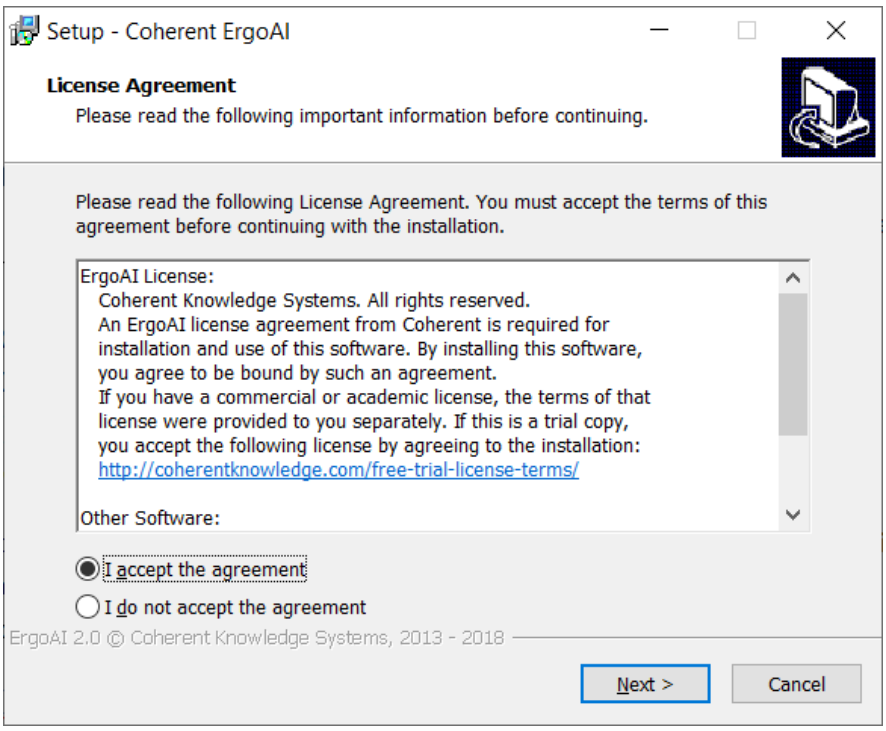
- ErgoAI 2.0 Reasoning Engine (Knowledge Base)
- Docker Desktop 2.1.0.5 (Container für Ausführung der Oracle Datenbank)
- DockerToolbox 19.03.1 (GUI für Containerverwaltung)
- SQL Developer 19.2.1 (Zugriff auf Oracle Datenbank)
- Oracle ExpressEdition 18c Datenbank (Datenhaltung)

Die Software/Installer und das Docker-Image für die Oracle Datenbank müssen vom jeweiligen Hersteller heruntergeladen werden. Die Skripte/Dateien (init_db.sql, tnsnames.ora) für die Initialisierung der Datenbank sind im Abgabe-Archiv enthalten. Die Installation wurde auf einem Windows 10 (64Bit) System getestet.

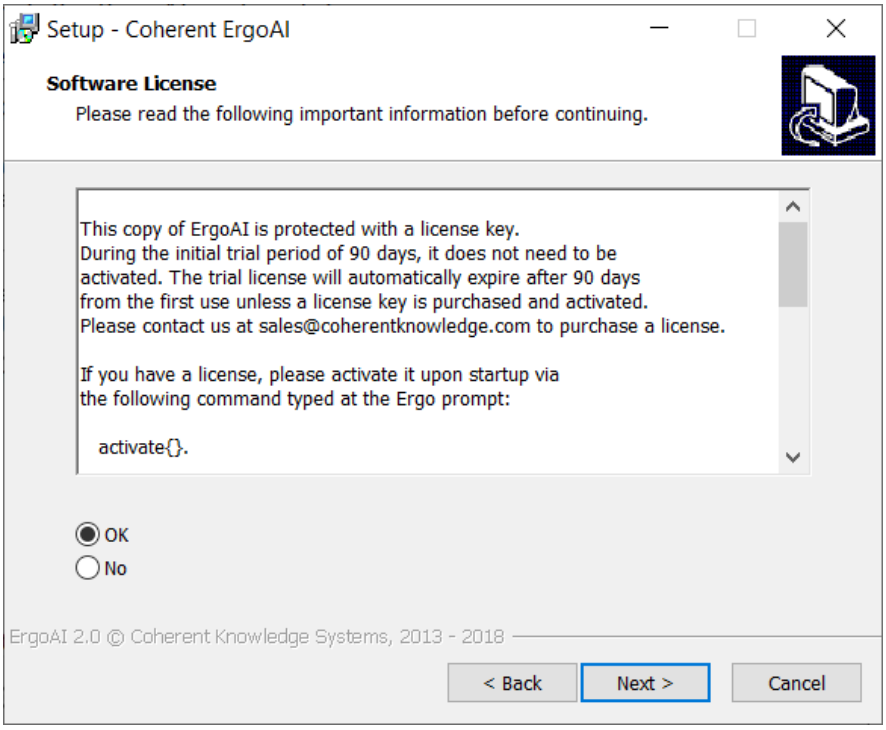
ErgoAI 2.0

Der ErgoAI 2.0 Installer enthält eine 90-Tage Demo Version für die ErgoAI Engine und ErgoAI IDE. Die ErgoAI IDE benötigt Java 8 JRE, die Engine selbst funktioniert aber auch ohne Java. Lizenzen für den akademischen Gebrauch sind auf Anfrage auf http://coherentknowledge.com/academic-license-request-form/ verfügbar.

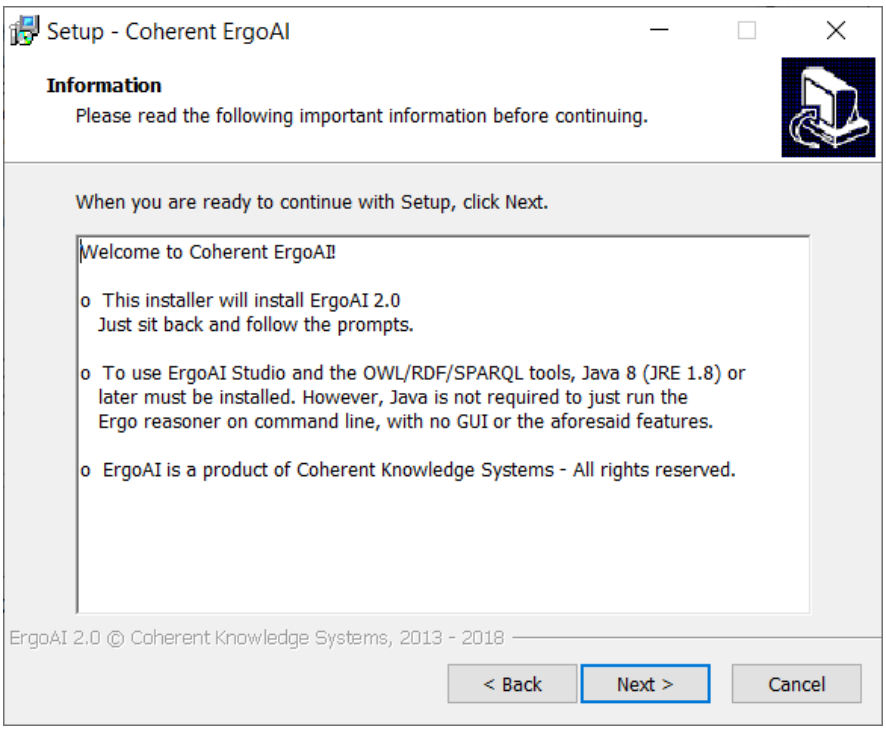
1. ErgoAI Installer (ErgoAI.exe) ausführen
2. Lizenzvereinbarung annehmen und „Next“ klicken



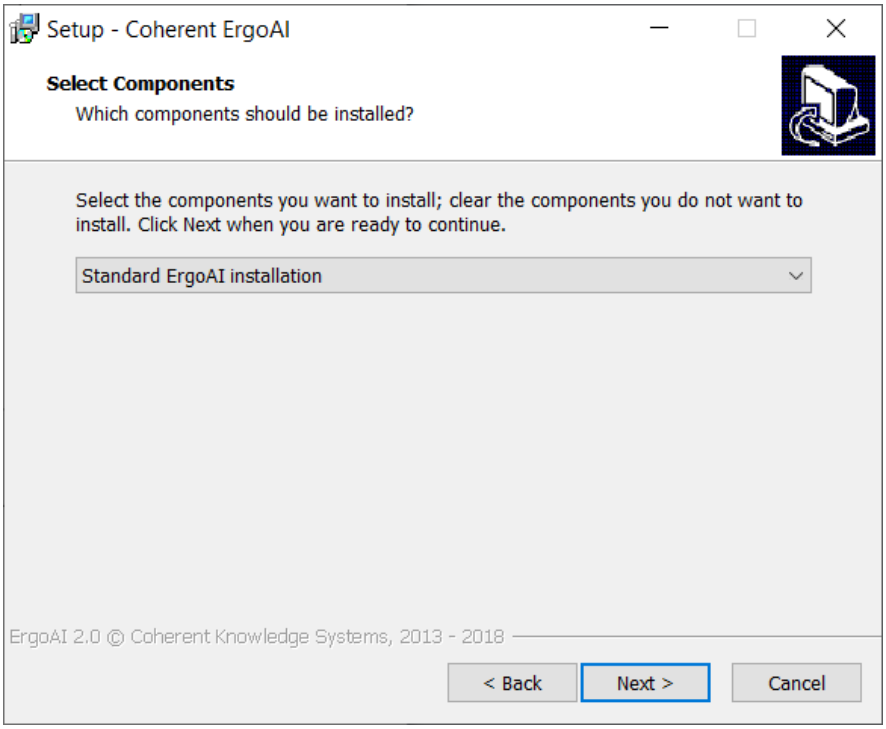
3. Lizenzhinweis mit „OK“ bestätigen und „Next“ klicken



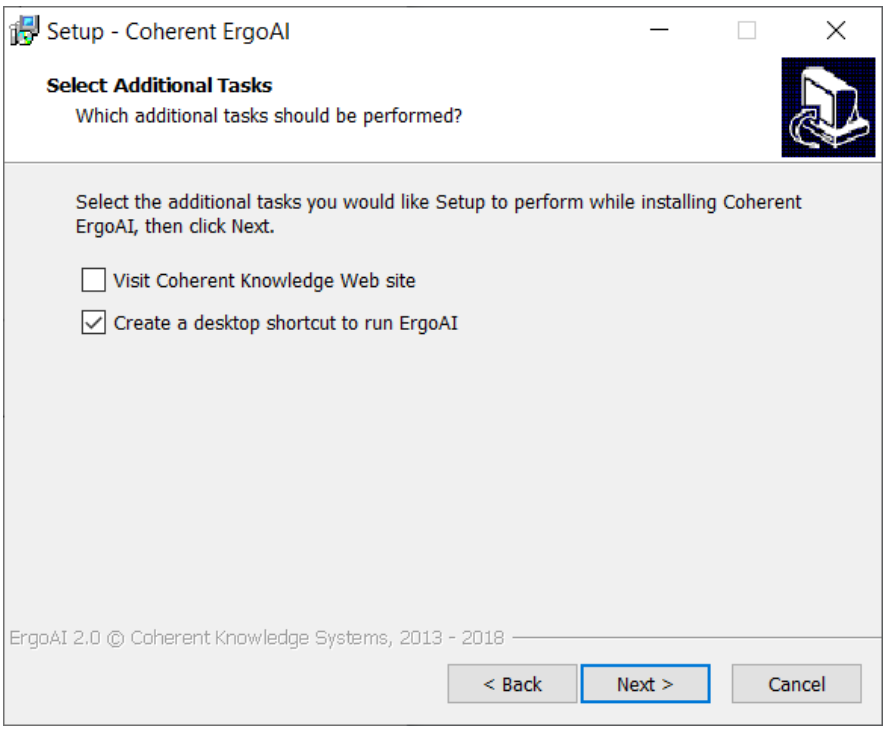
4. „Next“ klicken



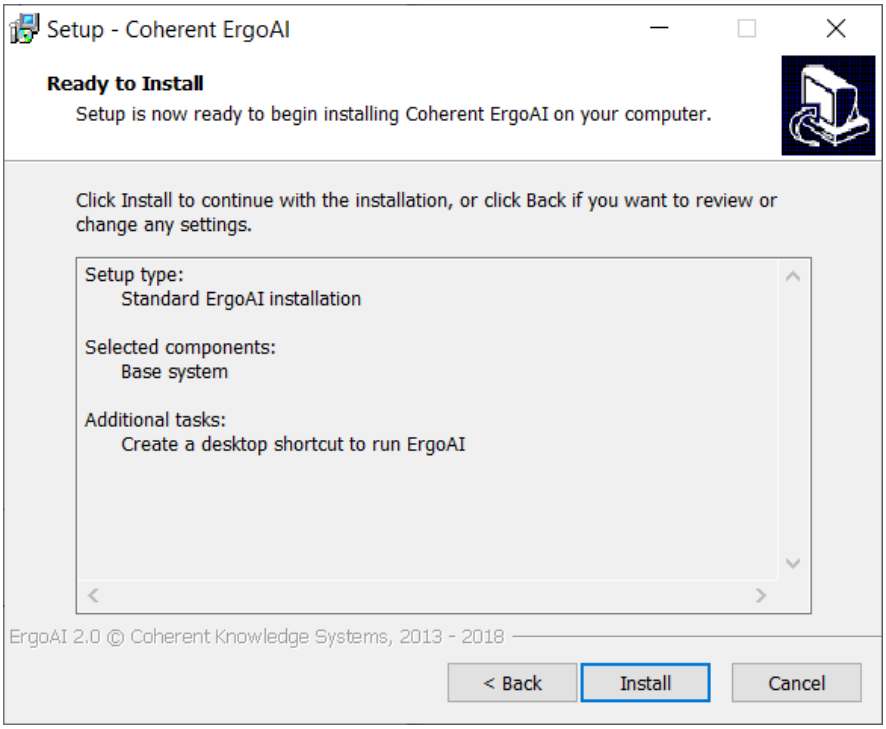
5. Standard ErgoAI Installation auswählen und „Next“ klicken



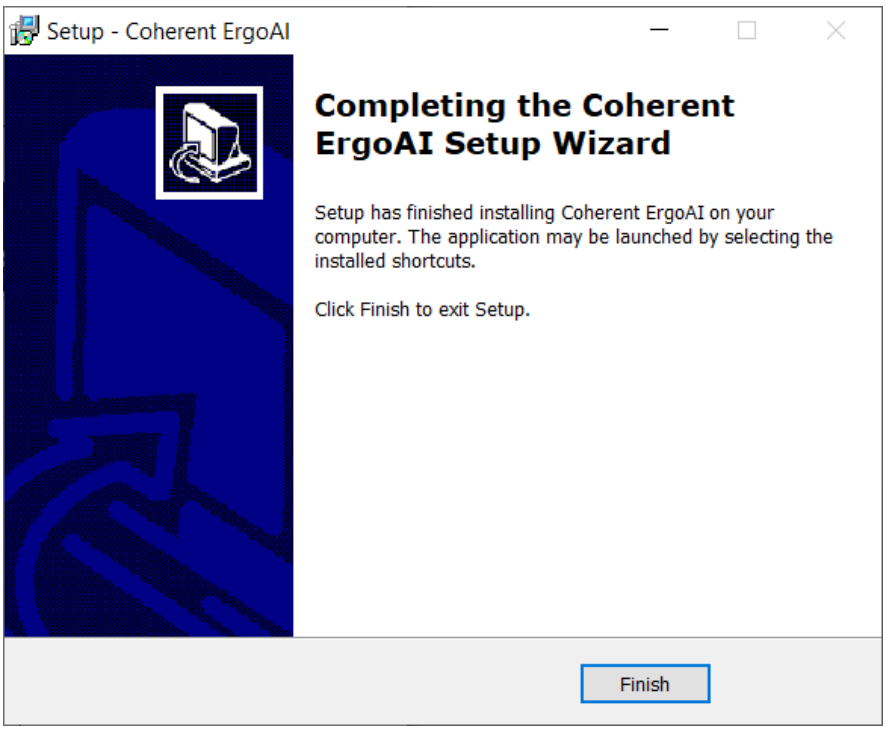
6. „Next“ klicken (Desktop Shortcut wird erstellt)



7. „Install“ klicken

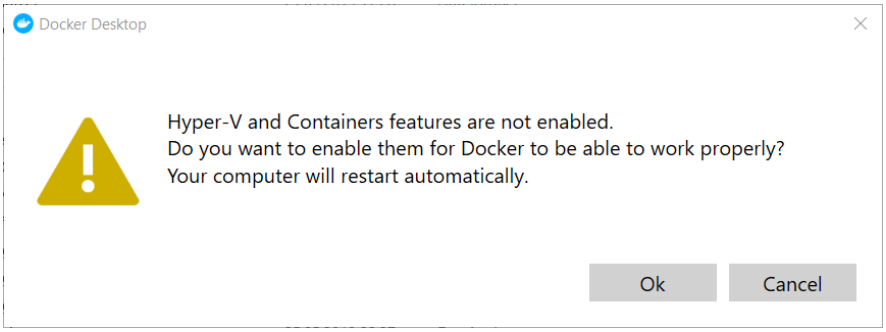


8. Installation abgeschlossen (Verzeichnis `C:\Users\{Benutzer}\Coherent\ErgoAI`)

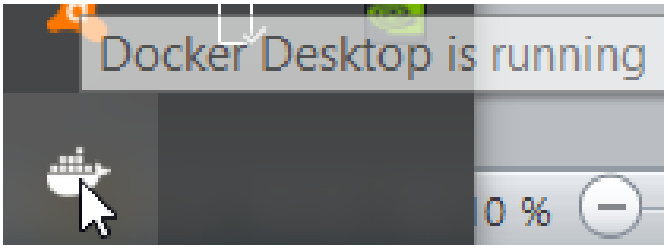


Docker Desktop 2.1.0.5

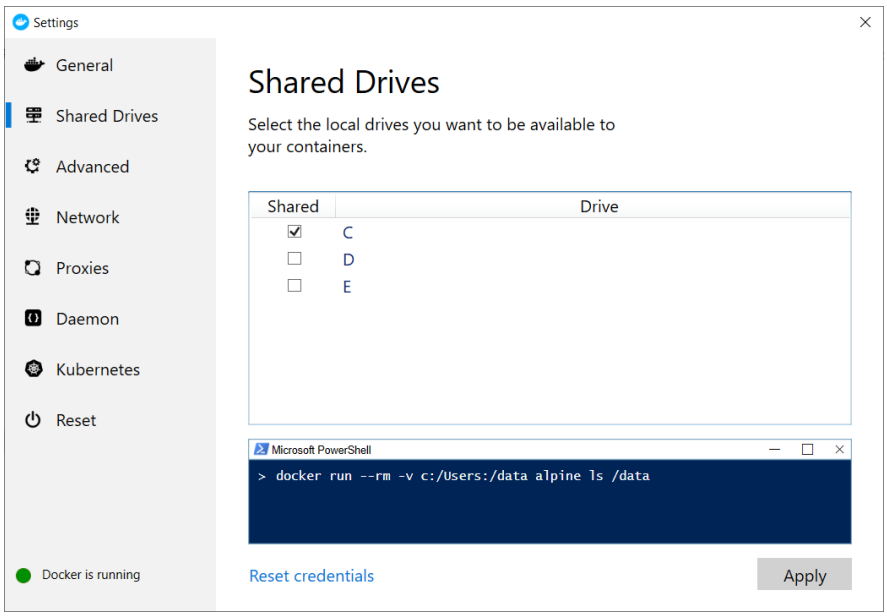
1. Docker Desktop 2.1.0.5 Windows Installer ausführen
2. Es werden die Installationsdateien entpackt und die Installation wird abgeschlossen
3. Man wird von Windows abgemeldet und bei der nächsten Anmeldung kommt folgender Hinweis:



4. Mit „Ok“ bestätigen -> Der Computer wird neu gestartet
5. Docker Desktop läuft nun im Hintergrund



6. Mit rechter Maustaste auf das Symbol klicken (Settings -> Shared Drives) und das Laufwerk für die spätere Ausführung der Oracle DB freigeben (`C:\`)



WICHTIG: Der aktuelle Benutzer benötigt ein Passwort, ansonsten kann das Laufwerk nicht freigegeben werden. Falls das nicht gewünscht ist, wird empfohlen für die gesamte Installation einen eigenen Windows Benutzer anzulegen.

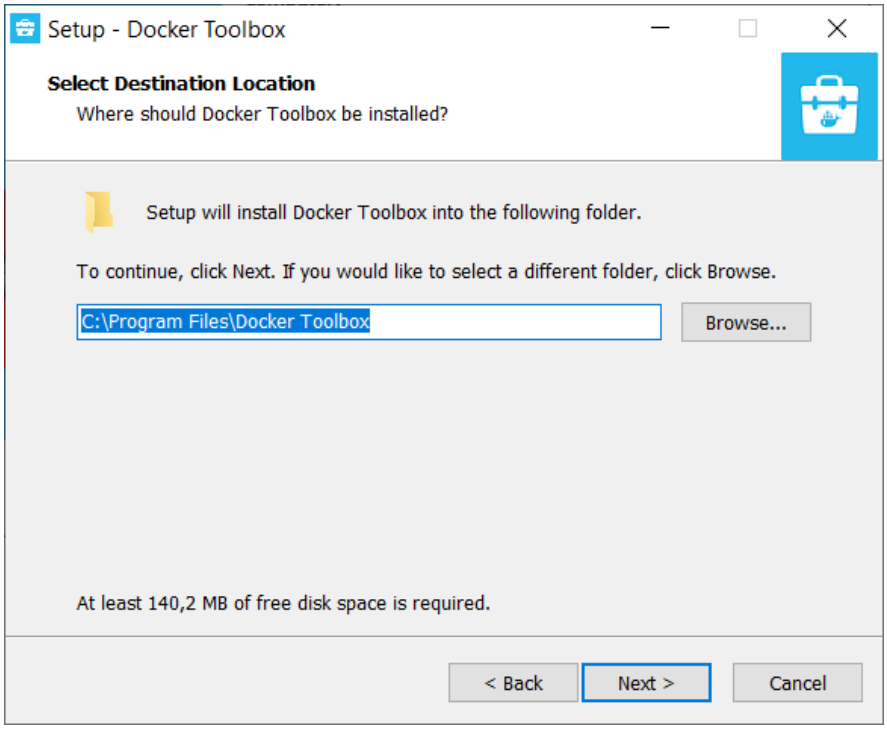
7. „Apply“ klicken und im folgenden Fenster Passwort eingeben und mit „OK“ bestätigen

DockerToolbox 19.03.1

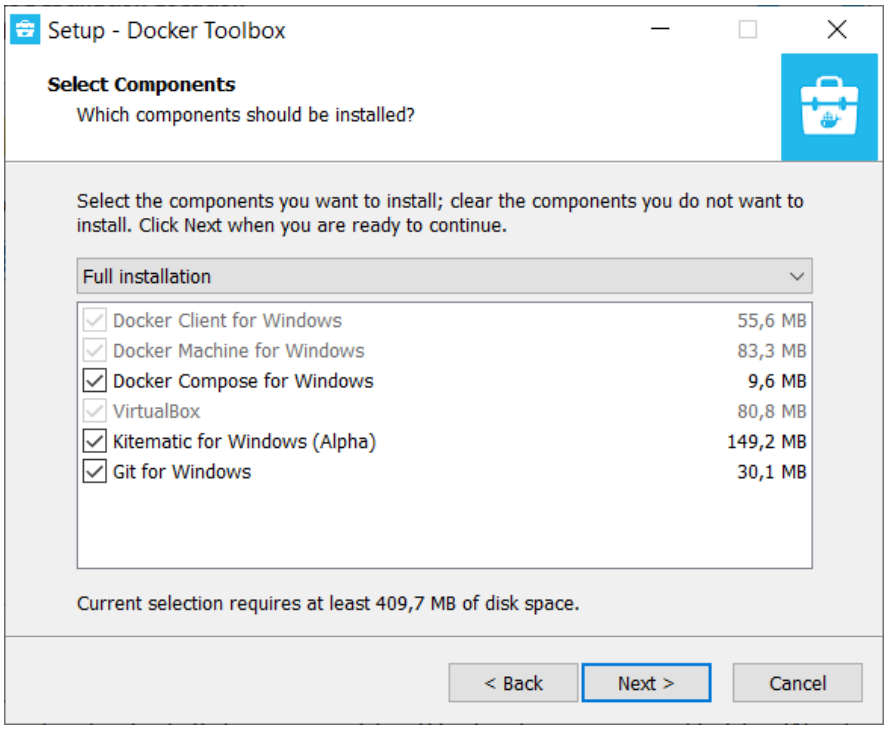
Für die Verwaltung von Docker Containern ist die GUI Kitematic zu empfehlen, diese ist in der Docker Toolbox enthalten.

1. DockerToolbox 19.03.1 Windows Installer ausführen
2. „Next“ klicken (eventuell vorher die Checkbox „Help Docker to improve Toolbox“ deaktivieren)

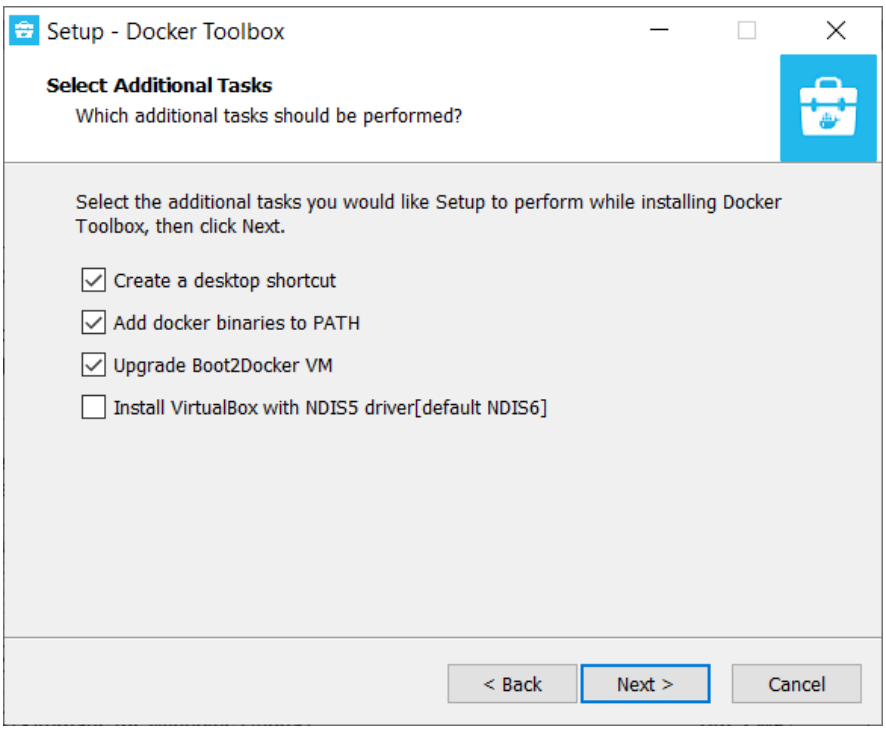
3. Installationsverzeichnis wählen (Standard `C:\Program Files\Docker Toolbox`)



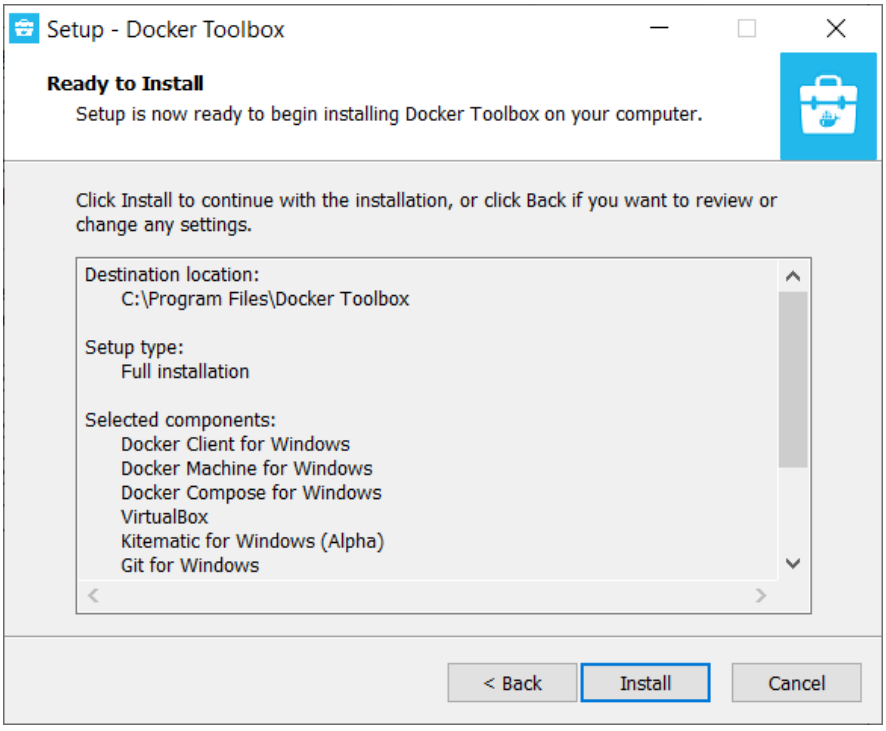
4. Komponenten für Installation auswählen (Docker Compose und Git for Windows optional)



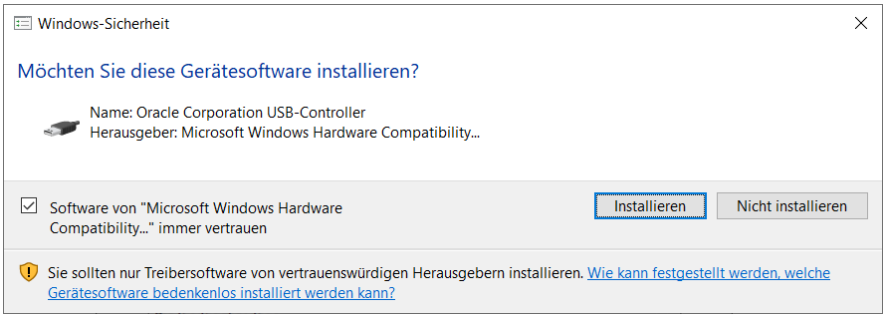
5. Vorbelegte Checkboxen auswählen und „Next“ klicken



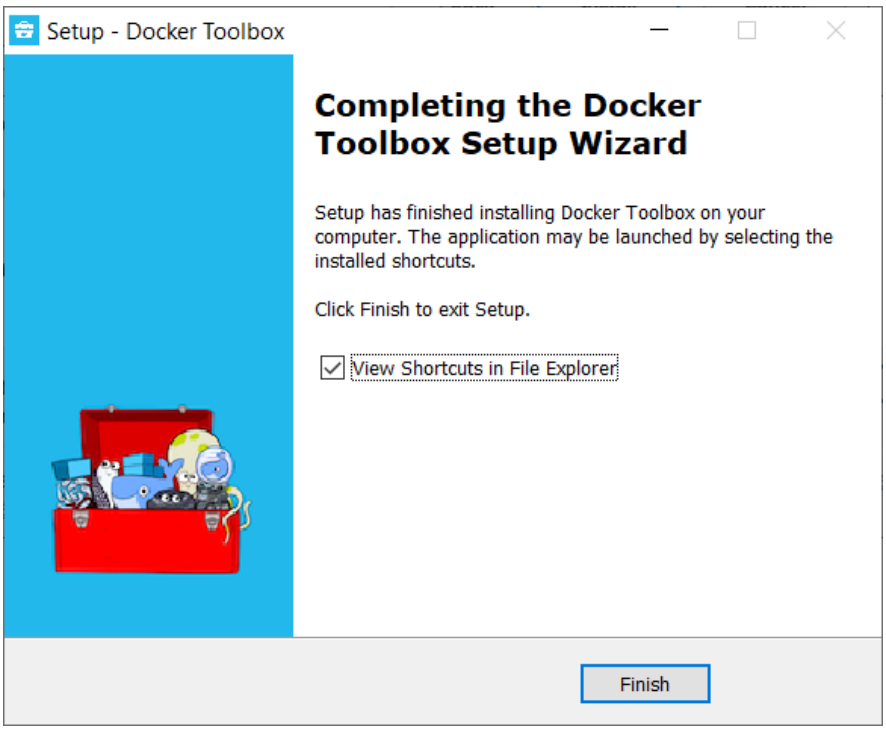
6. „Install“ klicken



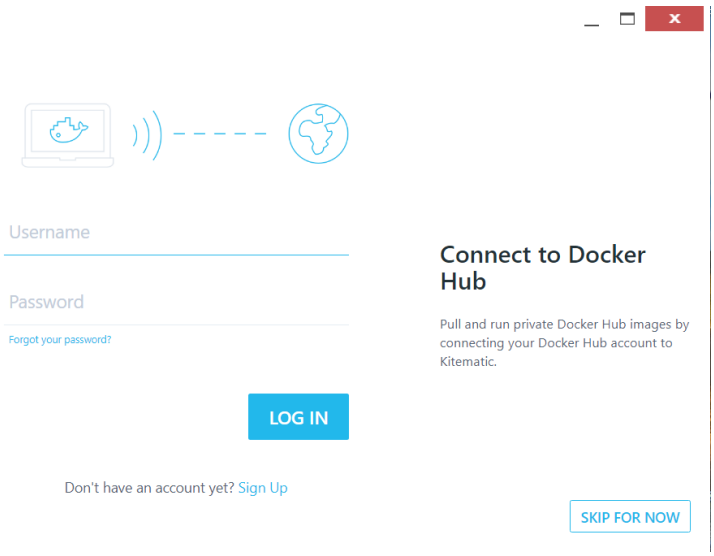
7. Ggf. Hinweis zur Installation von Gerätetreibern bestätigen



8. Installation ist abgeschlossen

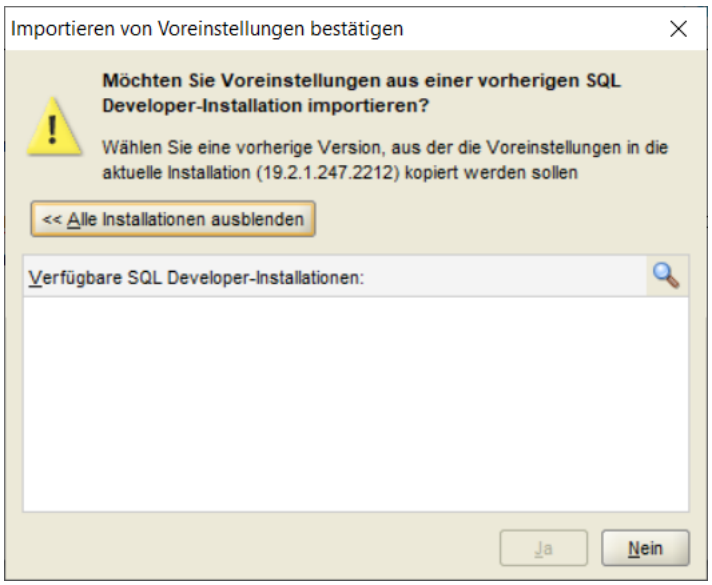


9. Beim ersten Start von Kitematic wird man gefragt ob man sich beim Docker-Hub anmelden will -> Dieser Schritt kann übersprungen werden

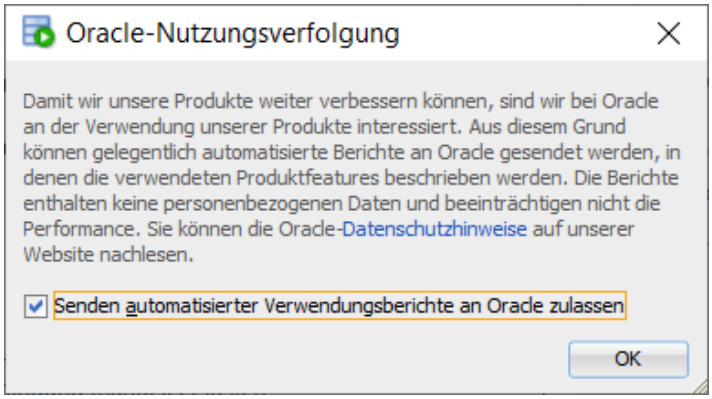


SQL Developer 19.2.1

Der SQLDeveloper muss nicht installiert werden, sondern kann mit der sqldeveloper.exe gestartet werden. Beim ersten Start kommt folgendes Fenster, in dem man Einstellungen aus einer früheren Installation importieren kann:



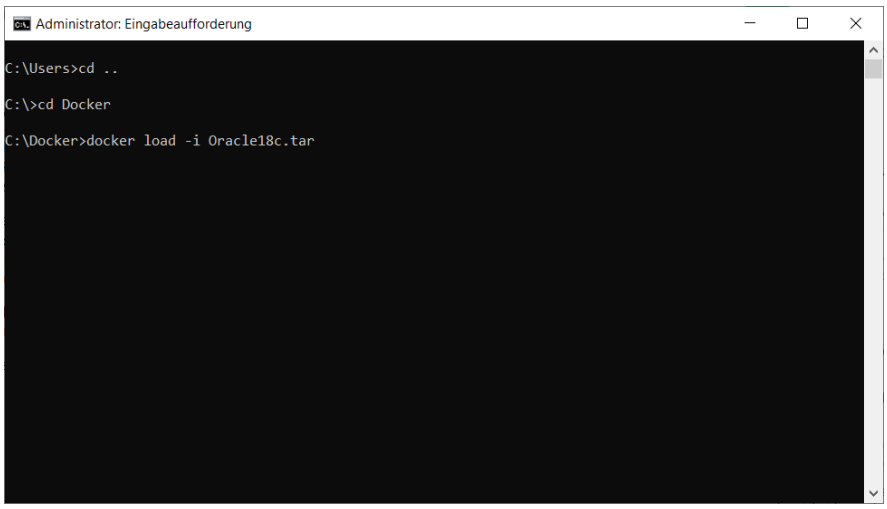
Außerdem wird man gefragt, ob man den Versand von Verwendungsberichten zulassen will. Dies kann deaktiviert werden.



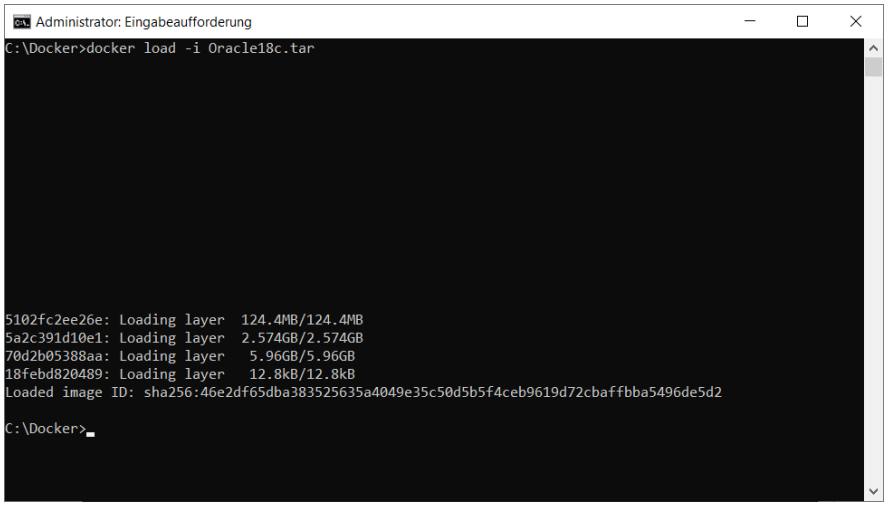
Oracle ExpressEdition 18c

Für eine einfache Installation und aufgrund von einfacher Portierbarkeit wird die Oracle XE 18c Datenbank auf Docker installiert. Die Voraussetzung dafür ist ein funktionierendes Docker Image für die Oracle Datenbank. Aufgrund lizenztechnischer Gründe wird dieses Image nicht mitausgeliefert, sondern muss selbst erstellt werden. Anleitungen dazu findet man im Internet (z.B. im Oracle Blog https://blogs.oracle.com/oraclemagazine/deliver-oracle-database-18c-express-edition-in-containers). Diese Anleitung geht davon aus, dass ein funktionierendes Docker Image für die Oracle XE 18c Datenbank verfügbar ist.

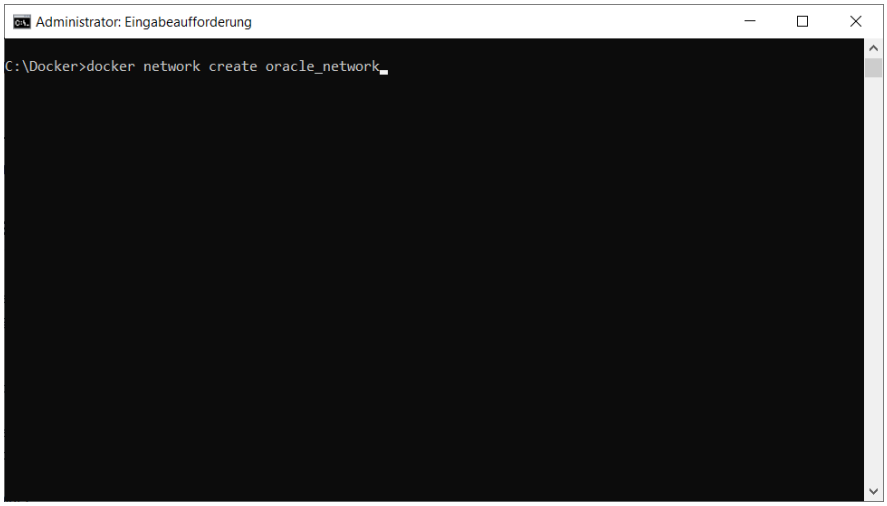
1. Docker Image (in diesem Beispiel Oracle 18c.tar) in Projektverzeichnis kopieren (`C:\Docker`)
2. Eingabeaufforderung oder Powershell öffnen und zu obigem Pfad navigieren
3. Mit folgendem Befehl kann das Docker Image in die Docker Engine geladen werden (kann einige Minuten dauern):



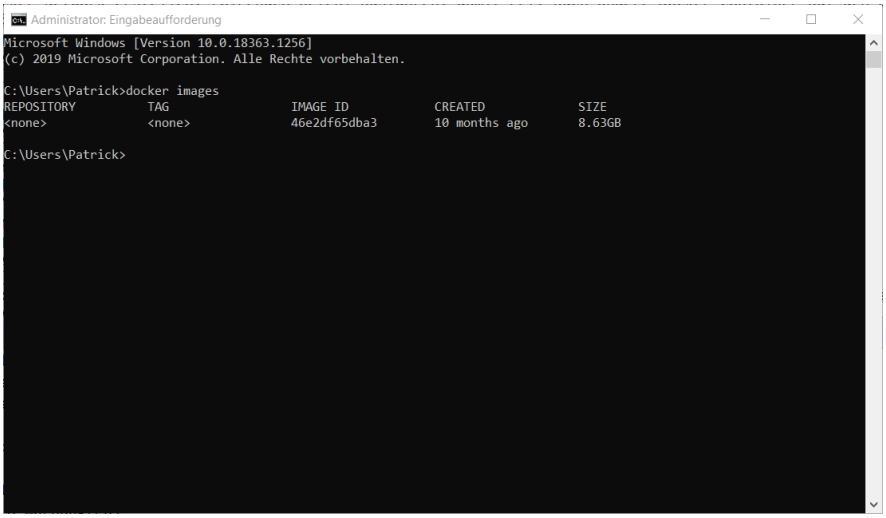
4. Nachdem das Image geladen wurde, sollte die Eingabeaufforderung folgendermaßen aussehen:



5. (Optional) Docker Network erstellen. Folgenden Befehl ausführen, damit ggf. auch andere Docker Container auf den Oracle Container zugreifen können:



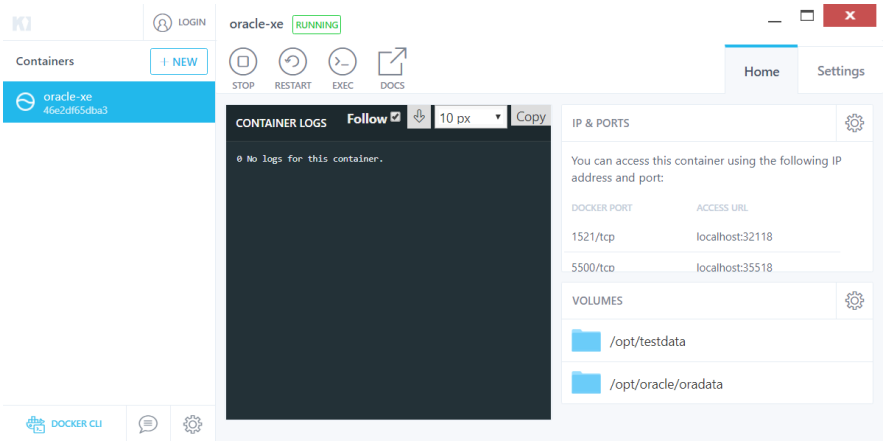
6. Um die Image ID des geladenen Docker Images zu ermitteln kann der `docker images` Befehl verwendet werden:



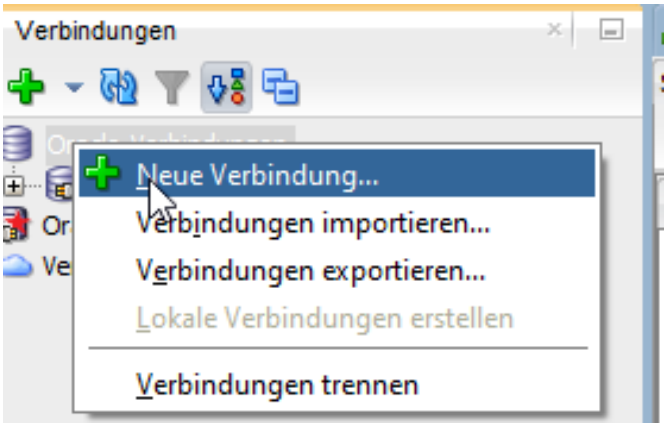
7. Der Docker Container kann nun mit folgendem Befehl gestartet werden, den letzten Parameter (Image ID) entsprechend anpassen:

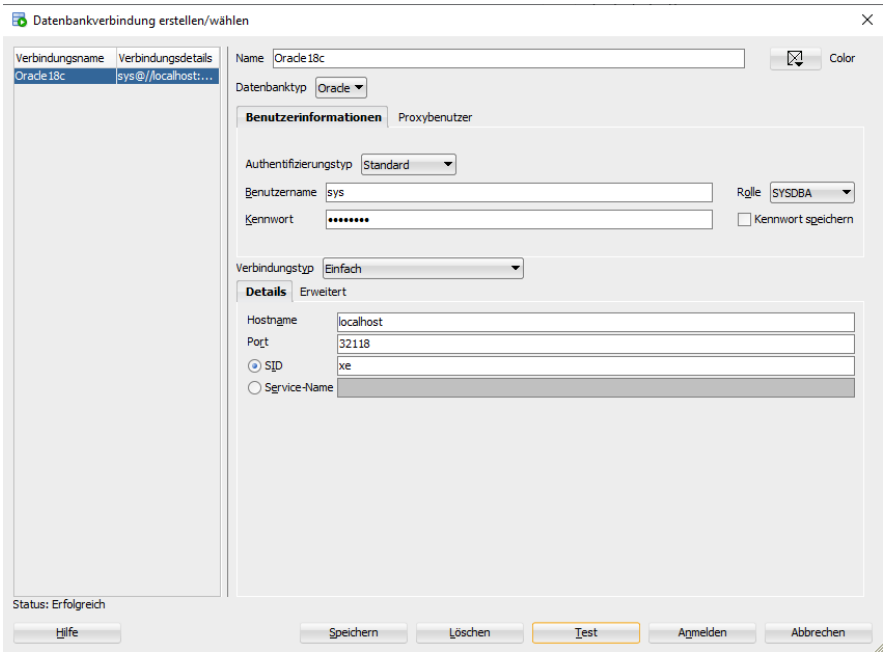
   ```
   docker run -d -p 32118:1521 -p 35518:5500 –name=oracle-xe –volume
   C:\OracleData:/opt/testdata –network=oracle_network 46e2df65dba3
   ```

8. Wenn alles funktioniert hat, sollte das laufende Image in Kitematic mit dem Tag „oracle-xe“ aufscheinen:



9. Um zu prüfen ob die Installation funktioniert hat, nun den SQLDeveloper starten und eine Datenbankverbindung aufbauen (Rechtsklick Oracle Verbindungen -> Neue Verbindung...)

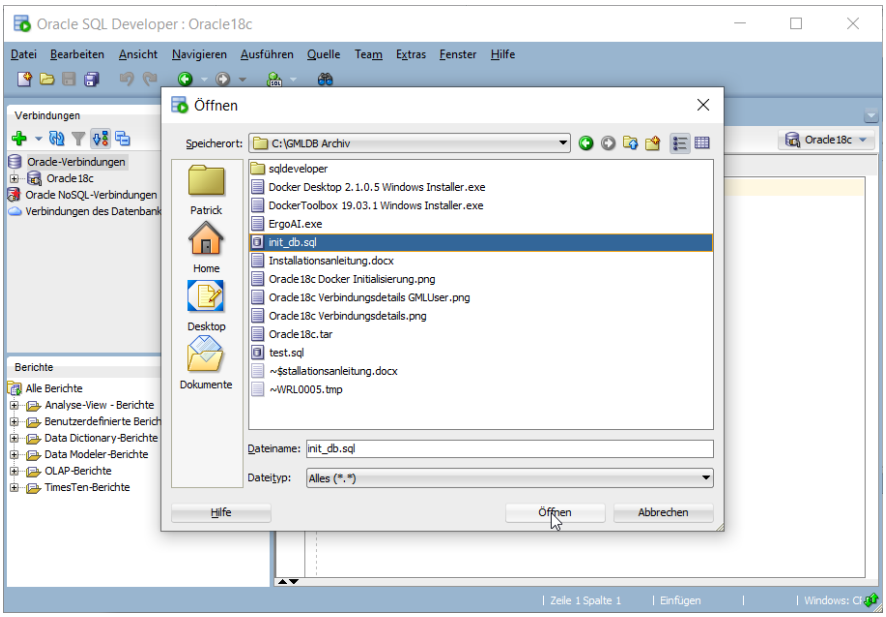




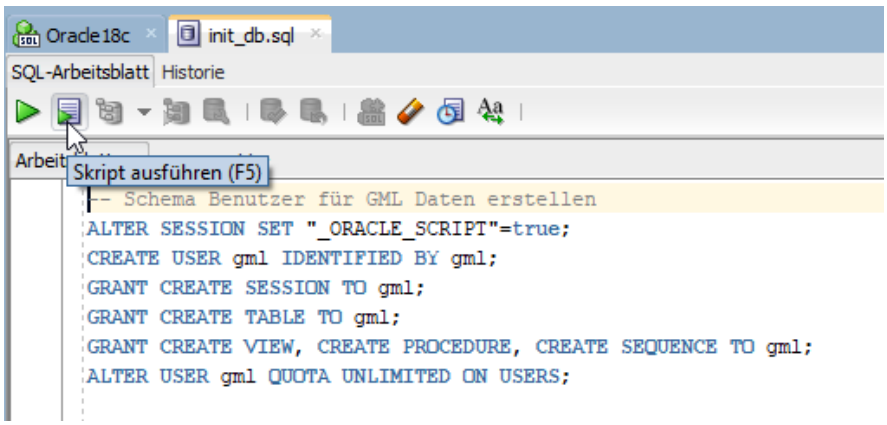
10. Der Verbindungsname kann frei gewählt werden, die restlichen Daten so eingeben wie auf dem Screenshot angegeben. Das **Standardpasswort** für den sys User lautet „**Oracle18**“. Dieses sollte nach der Installation geändert werden. Mit Klick auf den Button „Test“ kann die Verbindung getestet werden -> Status „Erfolgreich“

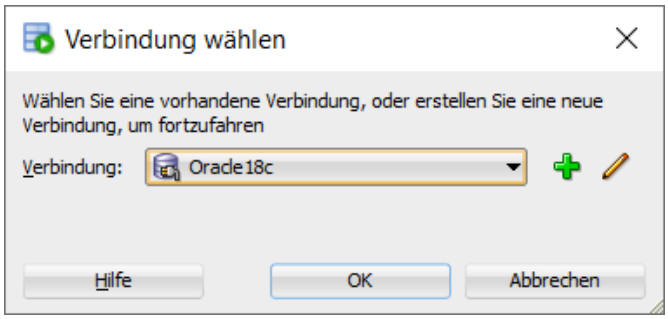
11. Mit Klick auf „Anmelden“ wird die Verbindung zur Datenbank aufgebaut

12. Um ein neues Benutzerschema anzulegen, das Skript `init_db.sql` im SQLDeveloper öffnen (Datei -> Öffnen)

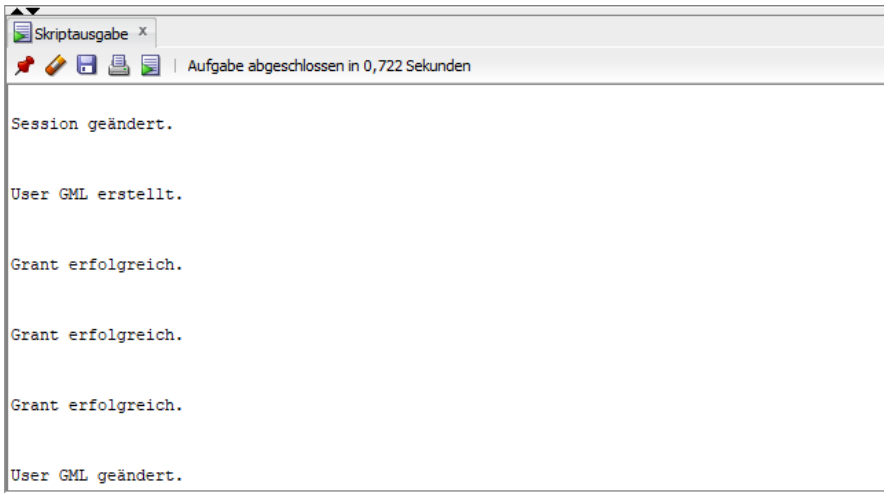


13. Das Skript kann über folgenden Button ausgeführt werden (Verbindung Oracle18c auswählen):



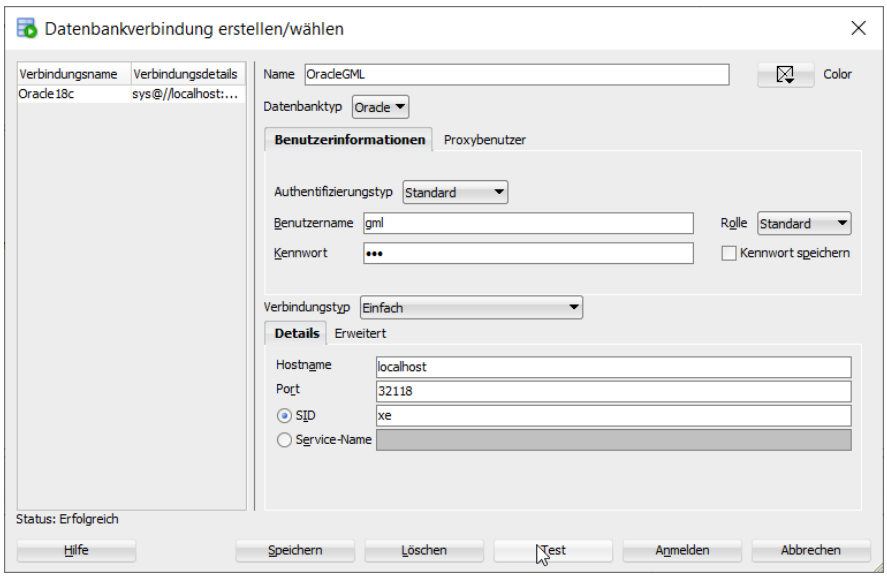


14. Dieses Skript erstellt einen neuen Benutzer und gibt diesem alle notwendigen Berechtigungen für Datenbankoperationen. Außerdem werden die benötigten Tabellen für die Speicherung der GML Daten erstellt. In der Skriptausgabe kann überprüft werden, ob die Ausführung erfolgreich war:



Dieser Benutzer wird dann für die weitere Arbeit auf der Datenbank verwendet!

15. Um sich mit dem Benutzer anzumelden, eine neue Verbindung erstellen und die Daten wie in folgendem Screenshot eingeben. Der Verbindungsname kann wieder frei gewählt werden. Das Standardpasswort für den **gml** Benutzer ist „**gml**“. Dieses sollte nach der Installation geändert werden.



16. Damit aus ErgoAI auf die SQL-Datenbank zugegriffen werden kann, ist es notwendig einen Datenquellen Namen für eine ODBC-Verbindung zu konfigurieren.

17. Voraussetzung für die ODBC-Verbindung sind der Oracle Basic (Light) Instant Client und das dazugehörige ODBC-Treiberpaket. Diese beiden Komponenten können von der Oracle-Webseite kostenlos heruntergeladen werden. Auf der Downloadseite das Betriebssystem, Architektur (32-bit oder 64-bit) und die gewünschte Version wählen. Die Installation wird im Folgenden für den Basic Instant Client 12.2.0.1.0 unter einem Windows 10-Betriebssystem (64-bit) beschrieben.

18. Zunächst das heruntergeladene Instant Client Archiv unter folgendem Pfad entpacken (bei Änderung des Pfades muss dies in den nachfolgenden Schritten berücksichtigt werden):

    `C:\Oracle\instantclient_12_2`

19. Anschließend sämtliche Dateien aus dem heruntergeladenen ODBC-Treiberarchiv in obiges Verzeichnis kopieren, sodass sich die Dateien des ODBC-Treibers mit den Dateien des Instant Clients im **gleichen** Ordner befinden.

20. Anschließend eine Windows-Kommandozeile (als Administrator) öffnen und in genanntem Verzeichnis die `odbc_install.exe` ausführen um den ODBC-Treiber zu installieren (Erfolgsmeldung erscheint in Kommandozeile).

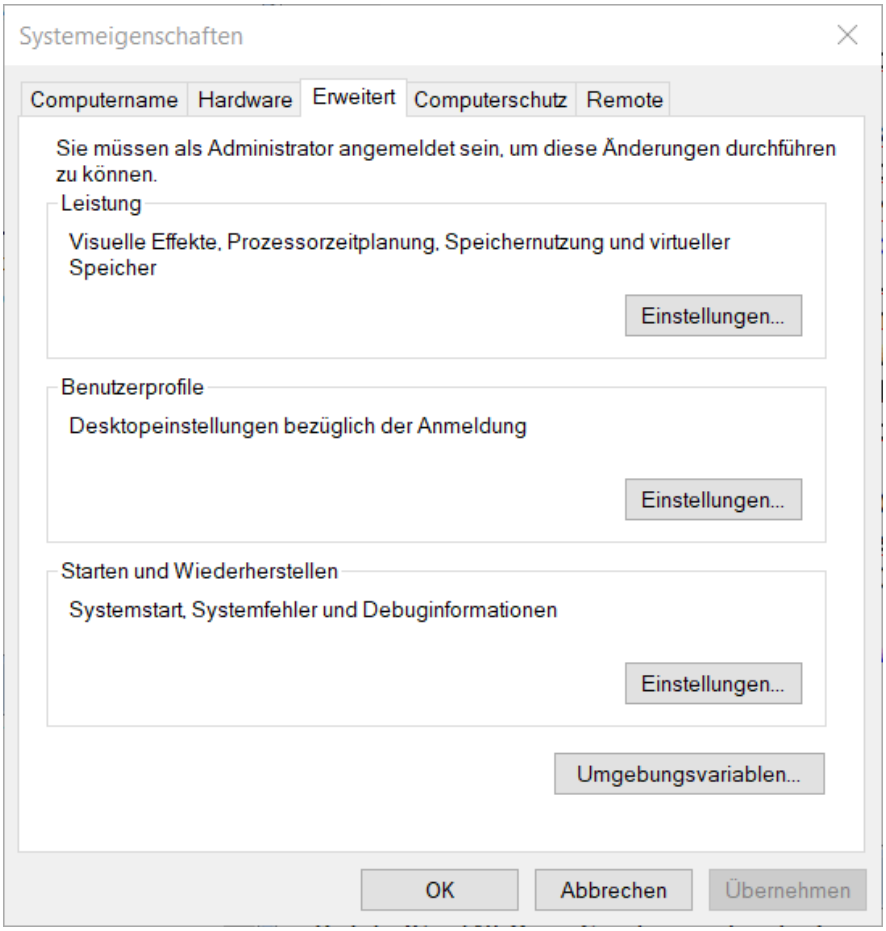
21. Die mitgelieferte tnsnames.ora Datei ebenfalls in den Instant Client Ordner kopieren. Diese Datei enthält alle Informationen (Protokoll, Hostname, Port, SID), die für die eigentliche Datenbankverbindung benötigt werden.

22. Damit vom Windows ODBC-Manager der ODBC-Treiber gefunden und der TNS-Name aufgelöst werden kann, müssen folgende Umgebungsvariablen (Benutzer- oder Systemvariablen) unter Windows gesetzt/ergänzt werden:

    Path: `C:\Oracle\instantclient_12_2` - Pfad zum Instant Client und ODBC-Treiberverzeichnis zur Path-Variable hinzufügen

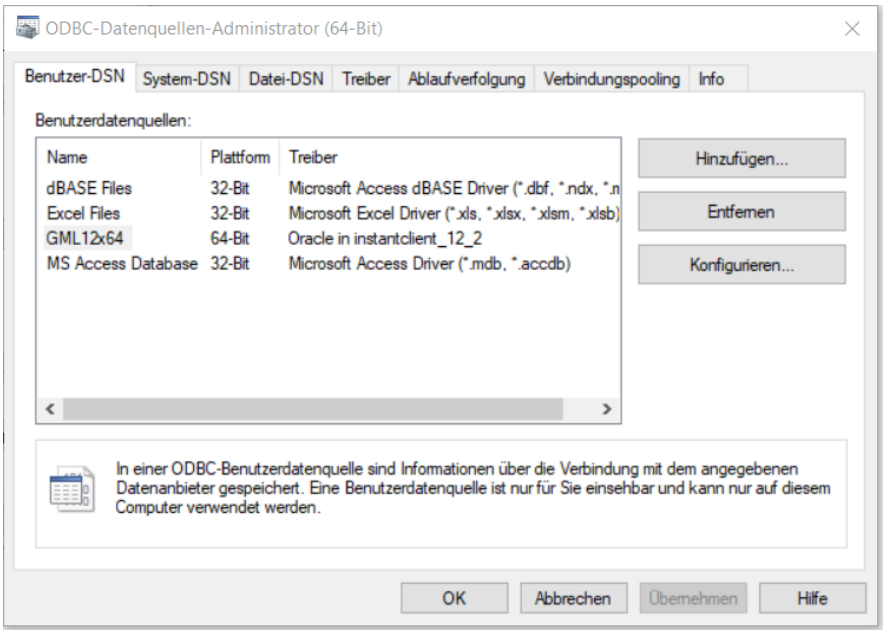
    TNS_ADMIN: `C:\Oracle\instantclient_12_2` - Neue Umgebungsvariable erzeugen (damit tnsnames.ora Datei gefunden wird)

Umgebungsvariablen können unter Windows in den Systemeigenschaften unter dem Erweitert-Reiter bearbeitet werden:

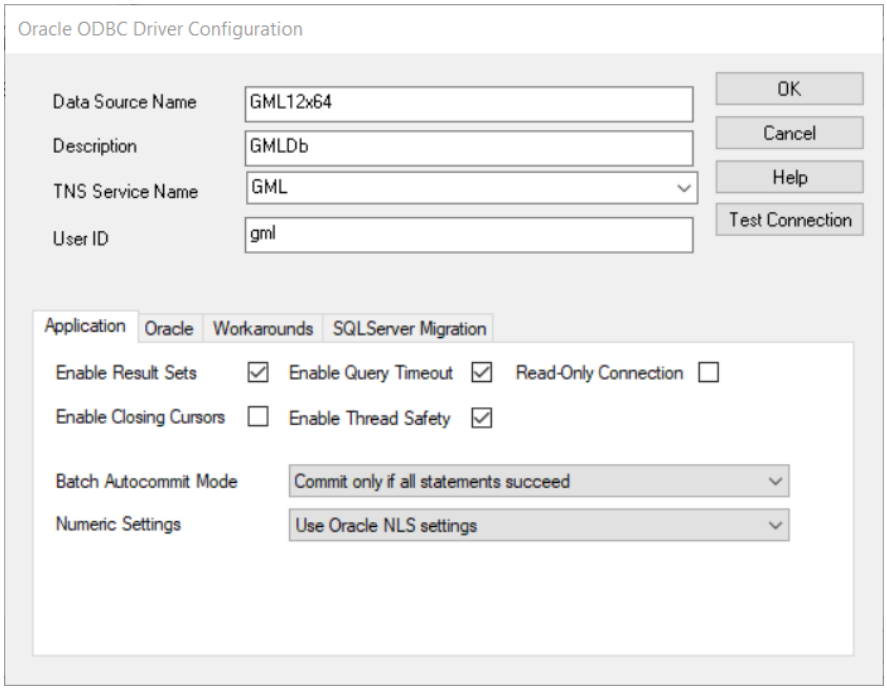


23. Anschließend den Microsoft ODBC-Datenquellen-Administrator (in Windows 10 bereits enthalten) über das Startmenü suchen und öffnen. Diesen gibt es jeweils in zwei eigenständigen Programmen für 32-bit und 64-bit Architekturen - den richtigen auswählen!

24. Einen neuen Benutzer- oder System-DSN hinzufügen und den Oracle instantclient_12_2 auswählen:



25. Im nächsten Fenster kann ein beliebiger Name und eine beliebige Beschreibung für die neue Datenquelle vergeben werden. Als TNS-Namen „GML“ aus dem Drop-Down Menü auswählen (ist in tnsnames.ora hinterlegt) und den Benutzer („gml“) für die Verbindung angeben. Die übrigen Parameter können bei den Standardeinstellungen belassen werden.



26. Bei Eingabe des Passworts für den „gml“-Benutzer kann die Verbindung auch gleich getestet werden: