JOHANNES KEPLER UNIVERSITY LINZ | JKU

Implementation of an Extensible Mapper of Aeronautical Information Exchange Model Data to ObjectLogic

Master's Thesis

to confer the academic degree of

Master of Science

in the Master's Program

Business Informatics

Author:
Ilko Kovačić, BSc.

Submission:
Data & Knowledge Engineering Institute

Thesis Supervisor:
o. Univ.-Prof. DI Dr. Michael Schrefl

Assistant Thesis Supervisor:
Felix Burgstaller, MSc.

Linz, July 2015

Eidesstattliche Erklärung

Ich, Ilko Kovačić, 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.

Signed: ____________________________________

Date: July 15, 2015

Acknowledgements

Many people supported and encouraged me during my studies and especially in the last year. Without them I would not have been able to successfully complete my master thesis. Therefore I would like to express my gratefulness to them in the following paragraphs.

I am grateful to o. Univ.-Prof. DI Dr. Michael Schrefl who gave me the opportunity to be a part of the Semantic NOTAM (SemNOTAM) research project. I express my gratefulness to my assistant thesis supervisor Felix Burgstaller, MSc. for helping me whenever I needed support. Moreover I would like to thank Dieter Steiner, MSc. who provided me an environment to implement my work. Furthermore I would like to thank Margit Brandl for supporting me during the bureaucratic processes.

This thesis would not have been possible without the support of Felix Burgstaller, MSc. who has been an outstanding advisor. Whenever I needed help he provided the right hints and useful information. Moreover I would like to thank him for the time he spent with me in the countless meetings were he gave me guidance in writing and structuring the thesis. Especially I would like to thank him for supporting me after the working hours which cannot be taken for granted.

Finally I would express my gratefulness to my family and to my friends. I would like to thank my girlfriend Marlene who supported me during my whole studies and especially during the completion of my master thesis.

Abstract

The Air Traffic Management (ATM) covers various tasks regarding air traffic control, air traffic flow management and aeronautical information services. This generates a huge amount of information addressing airspace users. The information exchange is based on the Aeronautical Information Exchange Model (AIXM) which is designed to enable the management and distribution of complex and evolving Extensible Markup Language (XML) structures. To avoid an information overflow the Semantic NOTAM (SemNOTAM) project provides intelligent and fine-grained filtering of these information. SemNOTAM is implemented as a rule-based system using ObjectLogic (OL). Therefore, a corresponding OL representation of the information to be filtered is required. This thesis contributes to the SemNOTAM system by implementing a mapper which transforms the XML data to its corresponding OL representation following the object-property model. This OL representation is inserted into the SemNOTAM system and used to conduct queries. A requirements analysis is conducted which analyzes the in- and outputs, exceptions, and constraints on the processing task. Based on the requirements a mapping approach is developed which especially covers the handling of geographical data. Moreover the XML transformation technologies Extensible Stylesheet Language Transformation (XSLT), Simple Application Programming Interface (API) for XML (SAX), Streaming API for XML (StAX) and Document Object Model (DOM) are evaluated with regards to their suitability for the mapper. It is shown that the DOM outperforms other technologies and consequently the mapper is implemented using DOM and Java.

Zusammenfassung

Das Flugverkehrsmanagement umfasst verschiedene Aufgaben von der Flugverkehrskontrolle, Flugverkehrsflussmanagement bis hin zur Verwaltung von aeronautischen Informationsservices. Die dabei generierte enorme Informationsmenge adressiert die Benutzer des Flugraums. Der Informationsaustausch erfolgt anhand des Aeronautical Information Exchange Model (AIXM), welches entwickelt wurde um den Austausch und die Verwaltung von komplexen Extensible Markup Language (XML) Strukturen, deren Entwicklung noch nicht abgeschlossen ist, zu ermöglichen. Um einen Informationsüberfluss bei den Benutzern zu vermeiden, wurde im Semantic NOTAM (SemNOTAM) Projekt eine intelligente und feingranulare Filterung dieser Informationen entwickelt. SemNOTAM baut auf einem regel-basierten System auf welches mittels ObjectLogic (OL), einer formalen Sprache zur Wissensrepräsentation, realisiert ist. Aus diesem Grund ist auch eine OL Repräsentation der Informationen notwendig. Der Beitrag dieser Arbeit zum SemNOTAM System ist die Implementierung eines Mappers der die XML Daten in ihre korrespondierende OL Repräsentation überführt. Die OL Repräsentation wird danach in das SemNOTAM System eingefügt und dient damit als Abfragegrundlage. Um dies zu bewerkstelligen, wird eine Anforderungsanalyse durchgeführt, welche die Ein- und Ausgaben, die Ausnahmen und die Einschränkungen während der Verarbeitung analysiert. Basierend auf den ermittelten Anforderungen wird ein konzeptueller Entwurf entwickelt, welcher insbesondere die Bearbeitung von geographischen Daten umfasst. Des Weiteren wird die Verwendung der XML-Transformations-Technologien Extensible Stylesheet Language Transformation (XSLT), Simple Application Programming Interface (API) for XML (SAX), Streaming API for XML (StAX) und Document Object Model (DOM) evaluiert. Es wird gezeigt, dass die DOM Technologie in Anbetracht der Leistungsfähigkeit den anderen überlegen ist, weshalb auch die Implementierung des Mappers mittels DOM und Java erfolgt.

Contents

Chapter 1

Introduction

Contents

This section gives an introduction to the topic of ATM and the processes within it. Furthermore, the scope of the thesis, its contribution, and finally its structure is provided.

1.1 Preface

The European Organisation for the Safety of Air Navigation (EUROCONTROL) expects to reach the 2008 peak of traffic (10.1 million flights) within and from Europe in 2016 (EUROCONTROL, 2014, p. 25). To manage this traffic volume an efficient and effective ATM is required (EUROCONTROL, 2015d). The ATM consists of the following three main components (EUROCONTROL, 2015d):

1. The **Air Traffic Control** is the process of safely separating aircraft in order to prevent collisions or critical situations. Therefore a separation of the aircraft in the sky as they fly and at the airports where they take off and land is needed.

2. **Air Traffic Flow Management** takes place before the flight and represents a central repository where all flights are analysed and computed. For each flight a flight plan is computed, determining the exact position of an aircraft at any given point in time. This flight plan is used by aeronautical controllers to supervise an aircraft and its flight crew.
3. **Aeronautical Information Services (AIS)** provide, combine and distribute aeronautical information to airspace users. The services provide information regarding administrative or legal matters, safety or navigation related activities, or about technical issues and their updates.

The information provided by AIS is encoded within so called Notices to Airmen (NOTAM). NOTAMs are a loosely structured, text-based representation of aeronautical information which are used by aviation systems and flight personnel such as flight crews or controllers. Flight crews use NOTAMs, provided by AIS, for their pre-flight briefings (EUROCONTROL, 2015c). These pre-flight briefings allow the pilots to get and read the relevant NOTAMs for their flight.

However, flight crews have to face a flood of NOTAMs during their pre-flight briefings. This flood is a consequence of limited filtering capabilities of current Aeronautical Information Management (AIM) systems (EUROCONTROL, 2015b). The loosely structured textual representation of NOTAMs conflicts with the needs of automated systems which require highly structured and standardised data structures (EUROCONTROL, 2010). The problem is that the decision whether a NOTAM is relevant or not mostly requires human interpretation since geographical, navigational, temporal information, and descriptions about events are represented as free text.

To overcome these drawbacks of textual NOTAMs a joint project between EUROCONTROL and the Federal Aviation Administration (FAA) was launched in 2010 (EUROCONTROL & Federal Aviation Administration, 2011b). The aim was to replace textual NOTAMs with digital NOTAMs. Digital NOTAMs represent structured data sets which can be read and interpreted by automated systems. In order to provide an encoding for digital NOTAMs, the existing Aeronautical Information Exchange Model (AIXM) was used and extended (Geospatial Intelligence TWG, 2006). AIXM allows to represent ATM elements such as event scenarios or features like airports, airspaces or routes. The machine-interpretable data structure allows filtering of NOTAMs without the need of human interpretation. Thus the flood of NOTAMs retrieved by flight crews and other flight personnel can be minimized using this automated filtering capabilities. This reduces the information overload and stress of the crew which positively affects their situation awareness.

Services, such as the Federal NOTAM Service and NOTAM Distribution Service (FNS-NDS), already provide filtering of NOTAMs but on a very limited scale. The service allows querying basic geographical information, text, and/or attributes for a point in time (Burgstaller et al., 2015). However these query capabilities are not sufficient enough for complex queries like filtering for a flight plan. To pose such a query, intelligent filtering capabilities are required.

This issue is addressed by the SemNOTAM project (Burgstaller et al., 2015). It aims to provide intelligent and fine-grained filtering and querying of digital NOTAMs (Burgstaller et al., 2015). Therefore a rule- and ontology-based Knowledge-Based Framework (KBF), that uses and extends semantic technologies, is developed. The KBF of the SemNOTAM system is depicted in Figure 1.1.

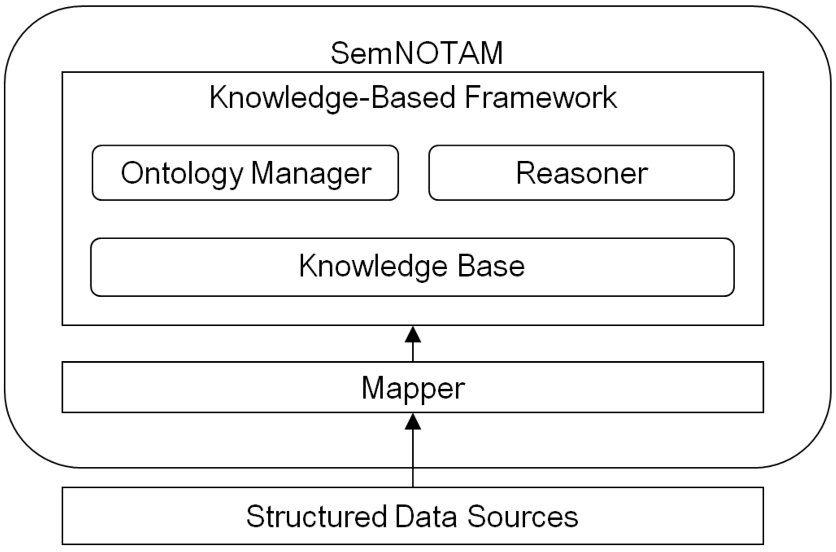

Figure 1.1: SemNOTAM Knowledge-Based Framework (Burgstaller et al., 2015).

The Ontology Manager is used to maintain the SemNOTAM ontology. An ontology is used to define "*explicit and formal specification of a shared conceptualization*" (Gruber, 1995). Furthermore the KBF consist of a Knowledge

Base (KB) and a reasoner. The KB contains the explicit facts and includes deduction rules which are used by the reasoner to derive new facts. These derived facts represent the implicit knowledge since it is not available without applying the deduction rules (Gringinger, Eier, & Merkl, 2011).

In order to access this knowledge the KBF provides a semantic querying interface (Frequentis AG, 2015). This interface allows to use various criteria. These criteria are based on location in space, progress in time, different events, aircraft, or flight types or any combination of them. Furthermore, structured data such as digital NOTAMs can be loaded into the KB. However, all data including queries must be mapped to a corresponding knowledge representation in order to store it in the SemNOTAM ontology of the KB (Burgstaller et al., 2015). Therefore a mapper is needed, as depicted in Figure 1.1. It maps various structured data sources such as the AIXM representation of digital NOTAMs to the corresponding declarative representation. This mapper is the main topic of this thesis.

1.2 Problem Statement

This thesis comprises the implementation of the mapper. As depicted in Figure 1.1 the mapper is a central component of the whole SemNOTAM system since it operates as an interface between the KBF and the structured data sources.

However, the implementation of such a mapper is not a trivial task due to the various data sources and interfaces. During the mapping process no information must be lost and the resulting knowledge-based representation must be syntactically and semantically correct. In addition to the central mapping task, the mapper must also provide preprocessing capabilities in order to enrich the accessed data. Since new data sources can be introduced in the future, the mapper must provide extension capabilities which allow to add new mapping and/or preprocessing tasks without modifying the existing implementation. The SemNOTAM system is designed with a focus on high adaptability and flexibility in order to face changing requirements. This design rational has to be followed in the implementation of the mapper to avoid design breaks.

The mapper transforms the structural data sources to their corresponding knowledge-based representation. The resulting knowledge-based representation is loaded into the KB. The mapper allows to configure in- and outputs, handle exceptions, and to conduct various processing tasks. Moreover it completes missing information and provides extension capabilities supporting future data sources. Without the mapper it would neither be possible to define new conceptualizations, insert new facts nor pose queries against the KB as their declarative representation could not be determined. The absence of the capability to insert new knowledge from structured data sources, such as digital NOTAMs, and to conduct specified queries would make the whole SemNOTAM system inoperative. This clearly shows the contribution of this thesis to the usage and progress of the SemNOTAM KBF.

The implementation of the mapper is based on requirements, which are derived from the SemNOTAM KBF. The requirements will be used to develop an mapping approach and to identify suitable technologies. Finally, the implementation will be evaluated in order to determine the performance.

1.3 Outline

The remaining thesis is organized in the following sections: At the beginning the SemNOTAM architecture and the used data representation is detailed, followed by the requirements analysis and the introduction of a mapping approach fulfilling them. Afterwards the implementation is presented. Finally, an evaluation of the mapper and a conclusion are provided.

Section 2 introduces and explains basic concepts needed for this thesis. Therefore the SemNOTAM project is detailed and its architecture is analysed (Section 2.1). Furthermore, textual NOTAMs is examined (Section 2.2.1). Thereafter the drawbacks of textual NOTAMs are explained and how they can be overcome by digital NOTAMs (Section 2.2.2). Finally, the used representations are introduced. This includes the AIXM representation (Section 2.3) for structured data and OL (Section 2.4) for knowledge.

A requirement analysis is conducted in Section 3. Therefore the task is delineated which covers the examination of the available data sources (Section 3.1). The data source examination comprises NOTAMs (Section 3.1.1), configuration data (Section 3.1.2), and queries (Section 3.1.3). In Section 3.2 the requirements are extracted, grouped, and detailed based on the task description. Besides the requirements, Section 3 describes challenges and problems which can occur during the fulfillment of the requirements (Section 3.3).

In Section 4 the basic mapping concept is introduced (Section 4.1). Moreover, the mapping approach is introduced in order to fulfill all previously defined requirements (Section 4.2). To implement the mapping concept suitable technologies are analyzed, evaluated, and the best performing is selected (Section 4.3). Furthermore, the extension capabilities of the selected technology are delineated in Section 4.4.

Based on the mapping concept and the selected technology the mapper is implemented in Section 5. Therefore, a conceptual design is introduced which supports extension and configuration capabilities (Section 5.1). The mapper is implemented based on the selected technology and the conceptual design (Section 5.2). Moreover, the performance is evaluated in order to show how the mapper performs while increasing the workload (Section 5.2.3).

Finally a conclusion is given which summarizes the thesis and provides an outlook on future work Section 6.

Chapter 2

SemNOTAM

Contents

This section will introduce the basic concepts needed for this thesis. Therefore the SemNOTAM project is detailed and especially the current architecture of the SemNOTAM system is analysed. A delineation of the system is mandatory in order to identify interfaces which can be used by the mapper.

Besides describing SemNOTAM, the textual NOTAMs and their successor, the digital NOTAMs, are examined. Moreover, a theoretical background of the used representations and technologies, which are crucial for the understanding the mapping task, are provided.

2.1 SemNOTAM-Project

As mentioned in Section 1.1, the SemNOTAM research project addresses the challenge of filtering NOTAMs and was launched in 2014 (Schrefl, 2014). It is a cooperative project between industry and university partners. The Austrian high-tech company Frequentis AG[1] is operating in ATM and public safety and transport branches providing and developing information and communication systems (Frequentis AG, 2015). The Institute of Data & Knowledge Engineering of the Department of Business Informatics of the Johannes Kepler University Linz which conducts research in the area of semantic systems, business intelligence, business process modelling and integration, and in web-based system (Gringinger, 2014). Other cooperation partners and subcontractors are EUROCONTROL, AustroControl and the FAA.

2.1.1 Background

The SemNOTAM project relies on previous research done in the EUROCONTROL's Single European Sky ATM Research Program (SESAR) and FAA's Next Generation Air Transportation System (NextGen). As stated in (Gringinger et al., 2011) these two programs aim to modernize and harmonize the ATM systems locally and globally. They are developing new capabilities, procedures and technologies in order transform the ATM systems. Former ground-based systems, which depend on voice communication between controller and the pilot and radar, are transformed to air/ground-integrated aviation systems based on digital data communication and satellite navigation (European Union, 2015). One modernisation is the replacement of textual NOTAMs with digital NOTAMs in the future AIM (Schrefl, 2014).

As stated Section 1.1, flight crews, especially pilots, have to face a flood of information. This can lead to stress, lack of situation awareness, missing prioritization, information overload and misunderstandings (Burgstaller et

[1] `http://www.frequentis.com`

al., 2015). In addition to the increasing number of NOTAMs this problem can be also traced back to the mandatory requirement of achieving one hundred percent recall of relevant NOTAMs (Burgstaller et al., 2015). One hundred percent recall is mandatory because a not retrieved relevant and safety critical NOTAM can lead to unpredictable negative consequences. As stated in Section 1.1, present services provide filtering of NOTAMs but on a very limited scale. Intelligent filtering capabilities are needed to supporting complex queries like filtering for flight plans (Burgstaller et al., 2015).

As shown in (Burgstaller et al., 2015), there are two approaches found in the literature addressing intelligent NOTAM processing. The first approach of (Gringinger, Trausmuth, Balaban, Jahn, & Milchrahm, 2012) is an ontology-based one not supporting business rules. However, without business rules it is not possible to define how, for whom and when information is relevant and how the recipients should be notified (Gringinger et al., 2012). In contrast to the approach of (Gringinger et al., 2012), (Zimmer et al., 2011) introduce a rule-based approach, using business rules which determine when and which information is significant for whom depending on the flight phases. However this approach does not support reasoning over ontologies leading to restricted capabilities since derived knowledge is not available or considered while filtering (Gringinger et al., 2012).

2.1.2 Problem Description

The SemNOTAM project follows a rule- and ontology-based approach introducing a knowledge-based framework that uses and extends semantic technologies enabling intelligent and fine-grained filtering and querying of digital NOTAMs (Burgstaller et al., 2015). As already mentioned in Section 1.1, the project aims to provide a machine-readable and ontology-based representation of digital NOTAMs which enable semantic querying using a wide range of criteria (Frequentis AG, 2015). This provides the airmen intelligent support for the management of their NOTAMs (Frequentis AG, 2015).

Due to the heterogeneous character of digital NOTAMs regarding their structure and semantics, intelligent querying becomes a challenging task. As stated in Section 2.2.1 about NOTAMs, they can be used to describe specific events. Events are defined within event scenarios which describe possible NOTAM conditions while being linked to specific AIXM features (Burgstaller et al., 2015). For each of these events, such as airspace restrictions, rules need to be specified (EUROCONTROL, 2010). Based on the given event scenario and pre-defined business rules, the SemNOTAM system derives which NOTAMs are relevant or not and how important they are (Burgstaller et al., 2015). Furthermore, SemNOTAM supports business terms representing the precise, understandable, and machine-interpretable concepts which are explicitly specified within an ontology. Furthermore, business terms can be defined in a hierarchical manner which allows to derive sub- or super-business terms.

This heterogeneity and the complexity results in various requirements for intelligent filtering concerning geographic querying, prioritizing, grouping, determining changes, customizing and personalization without re-design or redevelopments (Burgstaller et al., 2015). Due to the evolving character of the aeronautical environment solutions based on high-level-programming languages and relational database management systems are disadvantageous because they are characterized by a high adoption effort (Burgstaller et al., 2015). Therefore an adaptable knowledge-based system representing a declarative and machine-interpretable approach is used in SemNOTAM.

2.1.3 Method

As introduced in Section 1.1, the KBF consist of a reasoner and a knowledge base. These two parts are the core of every knowledge-based system (Gringinger et al., 2011). The KB contains the actual knowledge represented through facts, business terms and deduction rules. Facts are instances of business terms which are created during runtime. These base facts embody the explicit knowledge such as NOTAMs or aircraft types. Deduction rules describe how new facts can be derived. These derived facts represent the implicit knowledge since it is not available without applying deduction rules. As depicted in Figure 2.1, the appliance of these rules is done by a reasoner which reveals implicit relationships between facts. The derived facts will again be inserted into the knowledge base and used for further reasoning tasks. The knowledge-based framework of the SemNOTAM system is based on this reasoning process.

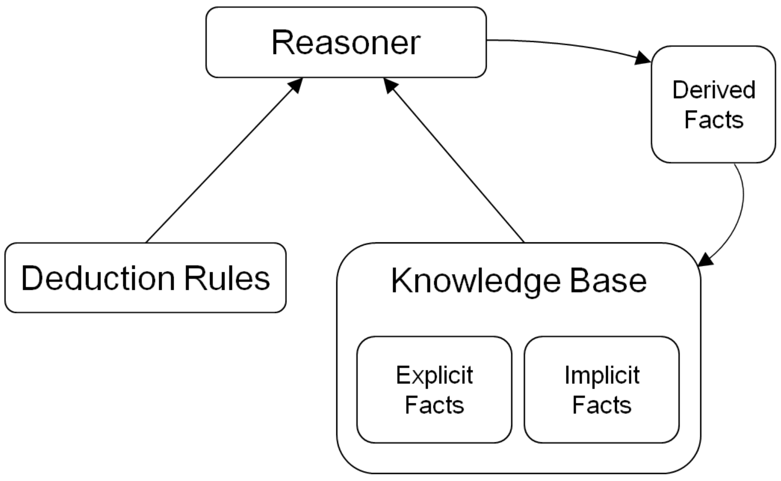

Figure 2.1: Process of reasoning (Gringinger et al., 2011).

As depicted in Figure 2.2 the framework accesses various structured data sources. The AIXM baseline data which is referenced in the digital NOTAMs represents static data which can be accessed remotely or locally (EUROCONTROL & Federal Aviation Administration, 2011b, p. 9). The baseline includes all values of all properties of a given permanently changed feature state (EUROCONTROL & Federal Aviation Administration, 2010, p. 7). Furthermore it accesses configuration data like route segments, aircraft characteristics and the query which includes specific interests (Burgstaller et al., 2015). All this accessed data must be mapped to the corresponding knowledge representation in order to store it in the SemNOTAM ontology of the KB (Burgstaller et al., 2015). As shown in figure 2.2, this AIXM to Ontology Mapper is one central layer within the SemNOTAM system because it provides the interface between the structured data and their semantic representation.

Figure 2.2 shows that the SemNOTAM ontology is located in the KB. It contains the base, derived facts and business terms which are specified by the user or the framework (Burgstaller et al., 2015). Business rules are included within the SemNOTAM Rules in order to define whether a NOTAM is relevant and how important it is. Beside the SemNOTAM ontology and the business rules, the KBF provides a SemNOTAM Interest Specification. This allows the user to specify his/her interest and thus defines which SemNOTAM Rules will be applied. Their temporal, aircraft and spatial interest are provided as simple interests which can be combined to complex interests using the set operators intersection and union.

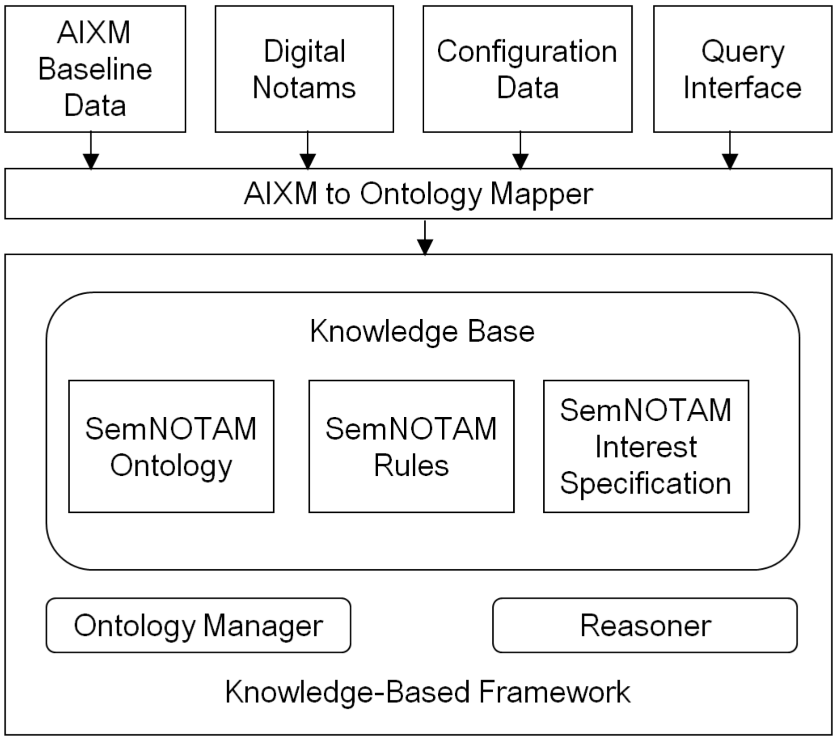

Figure 2.2: SemNOTAM Knowledge-Based Framework (Gringinger et al., 2011).

This KBF introduced in the SemNOTAM system provides high adapting capabilities due to the declarative representation of data and rules (Burgstaller et al., 2015). Furthermore it enables the intelligent and fine-grained filtering based on specific interests which can take individual interests into account. By supporting business terms, the framework allows to group query results according to topics (Burgstaller et al., 2015).

2.1.4 SemNOTAM Architecture

The architecture of the SemNOTAM system is based on the core architecture depicted in Figure 2.2. It supports embedding into environments which can be connected to external systems. The only prerequisites are a interface to a database supporting the system with NOTAMs and a Result Builder combining the outcome with the original NOTAMs (Burgstaller, Szabolcs, Steiner, & Frequentis AG, 2014, p. 6).

Beside the environment, the SemNOTAM architecture can be split into the

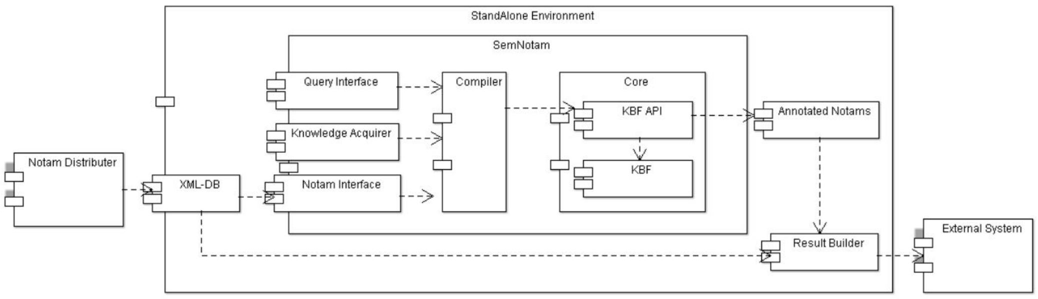

Figure 2.3: SemNOTAM architecture and interfaces (Burgstaller et al., 2014, p. 6).

following two parts, as depicted in figure 2.3 (Burgstaller et al., 2014):

- **SemNOTAM component:** it provides SemNOTAM interfaces to the environment. These interfaces grant all access needed to load or purge data and rules (Knowledge Acquistion), conduct queries (Query Interface), provide configuration data such as segments (Segment Interface) and retrieve results (Result Interface) (Burgstaller et al., 2014, p. 11). All of these interfaces use an AIXM respectively XML representation as input data and pass them over to the mapper.
- **Core:** the core comprises the KBF and the appropriate KBF API (Burgstaller et al., 2014, p. 10). Both parts depend on the ontology-based representation provided in OL. The whole KBF component is written in OL and can be access by the API of the OntoBroker Reasoner of Semafora. Since the SemNOTAM component implements interfaces to load and purge NOTAMs the core provides the KBF API to support this actions (Burgstaller et al., 2014, p. 10). Furthermore, the KBF API grants loading and purging of segments as well as other configuration data such as concepts, aircraft, groups, priority orders, group arrangements and default parameter (Burgstaller et al., 2014, p. 10). However these NOTAM and knowledge acquisition interfaces require an OL representation of the configuration data respectively NOTAMs provided by the mapper. The SemNOTAM interfaces for querying and retrieving the result are also supported by the KBF API. In contrast to the Query Interface which requires again an OL representation as input, the Result Interface outputs a XML respectively AIXM representation (Burgstaller et al., 2014, p. 10).

As shown the SemNOTAM component depends on an AIXM representation whereas the Core expects an OL representation. Both depend on the

AIXM to OL mapper between them because their interfaces access each other. Based on this detailed architecture, mapping requirements can be derived. However, before this will be done the two representations will be examined.

2.2 Notices to Airman

NOTAMs represent unclassified notices or advisories that are distributed by means of telecommunication (Myers, 1978). They primarily address aviation personnel and systems that are essential for flight operations such as controllers and pilots. NOTAMs contain information regarding any hazards of a route at a specific location, essential timely knowledge or any change, establishment, or condition in any aeronautical facility or service. Changes or essential timely knowledge are represented by events. Events which indicate changes in conditions or availability of aeronautical facilities or services can lead to possible hazards. Examples for hazards can be closed runways, military exercises, temporary route changes, dust or volcanic ash contaminations or changes states of runways due to weather conditions.

From the examples it can be seen that there are many NOTAMs which can be divided into several types regarding their addressed audience. A coarse division is provided by the FAA (Federal Aviation Administration, 2010), where international, domestic, civilian and military NOTAMs are distinguished. However, there are important specific types such as SNOWTAMs, BIRDTAMs and ASHTAMs (Icao-Ais-Aimsg, 2011). SNOWTAMs are used as notifications about the removal or existence of unsafe conditions due to ice, snow, water or slush indicating that the movement area is possibly restricted (Icao-Ais-Aimsg, 2011). BIRDTAMs include information about possible bird strike areas and ASHTAMs contain warnings about volcanic activities such as eruptions and ash clouds (Icao-Ais-Aimsg, 2011).

Even thought NOTAMs can address different audiences and cover various events, they are the most basic and important concept within the ATM domain. NOTAMs were already defined in 1978. In the last 50 years NOTAMs developed from simple text messages, supplying pilots and other operational flight personnel with critical safety information, to containers for other not safety related information such as Navigational Aids (NAVAIDs) (EUROCONTROL, 2015b). Since NOTAMs require human interpretation this extended usage has led to an information overload.

Performance statistics of the European Aeronautical Information Services Database (EAD) show an increase of newly added NOTAMs within Europa from 44764 in 2007 to 91578 in 2013 (EUROCONTROL, 2015c). This increase represents a doubling of newly added NOTAMs within only six years. The analysis of the increased NOTAM circulation in the France AIS (International Civil Aviation Organization, 2013), identified several reasons for this development. They show that the number of NOTAMs is not directly correlated to air traffic; but it depends on the number of air navigation facilities and activities. The main causes for the extensive NOTAM usage can be found in the Global Navigation Satellite System (GNSS) navigation, instrument flight procedures, information about runway closures or restrictions, taxiways, air navigation warnings, information about time slots and restrictions regarding aerodromes and their facilities, services, and airspaces (International Civil Aviation Organization, 2013). As already stated in Section 1.1, this flood of NOTAMs is given to flight crews which use them for pre-flight briefings resulting in a Pre-flight Information Bulletin (PIB) of ten to fifty pages for an internal European flight (EUROCONTROL, 2015c). Due to current limited filtering capabilities forty and sometimes up to ninety percent of the provided information is neither important nor relevant for their flight (EUROCONTROL, 2015b). This can lead to flight crews unaware of important and safety critical information since they are not able to detect it within this flood of information.

2.2.1 Textual Notices to Airmen

NOTAMs cover various events and use cases. This comprises especially dynamic data such as temporal knowledge. For that reason a textual representation for non-static information was chosen. As mentioned in Section 1.1, this led to textual NOTAMs which were loosely structured and mainly provided as free text. However this conflicted with automated AIM systems (EUROCONTROL, 2010). AIM systems highly depend on structured and standardised data structures representing accurate, timely, and quality assured aeronautical information (EUROCONTROL, 2010).

Since dynamic information is encoded as free text it becomes the burden of the pilot or controller to recognize which information is relevant and which is obsolete (EUROCONTROL, 2015b). This manual process offers an entry point for errors and requires a post-submission quality control (EUROCONTROL, 2015b). Possible errors can occur due to misinterpretations of NOTAMs based on different or missing understanding of used words (EUROCONTROL, 2015b). Besides that, human interpretation is also required for safety critical events which can be partly encoded as text. To cover such events, a detailed description is required which again leads to complexity which cannot be comprehended easily by non-experts. (EUROCONTROL, 2015b). Although human interpretation is error-prone, it is needed to feed automated systems. The encoded information within textual NOTAMs cannot be extracted otherwise in a satisfying way. This significantly slows down the information flow and, as mentioned before, leads to errors and misunderstandings (EUROCONTROL, 2015b).

The information overload mentioned in Section 2.2.1 is not just a result of the increased number of NOTAMs, it is also a consequence of the geographical and temporal inaccuracies of textual NOTAMs (EUROCONTROL, 2015c). Geographical aeronautical information is expressed by geometrical forms defining an ATM element such as airport surface, airspace, routes, runways, etc. The automatic interpretation of NOTAMs containing geographical data is currently limited to the consideration of the position and the radius of influence (EUROCONTROL, 2015c). These limitations lead to vague geographical definitions where operators tend to overestimate the radius of influence in order to be on the safe side (EUROCONTROL, 2015c). These overestimations negatively affect flight crews as NOTAMs which are completely irrelevant for them will be assigned to their pre-flight briefing. As stated before, the temporal aspect is also being considered in order to find the latest NOTAMs and their states. The interpretation whether a NOTAM is relevant for a given time period, again depends on the human interpretation which can result once more in a high number of irrelevant NOTAMs (EUROCONTROL, 2015c).

2.2.2 Digital Notices to Airmen

To overcome the drawbacks of textual NOTAMs, EUROCONTROL and FAA launched a joint project in 2009 with the aim of developing digital NOTAMs (EUROCONTROL & Federal Aviation Administration, 2011b, p. 2). In contrast to textual NOTAMs, digital NOTAMs are structured data sets which can be read and interpreted by automated systems removing the need of human interpretation. A digital NOTAM is defined as a "*data set made available through digital services containing information concerning the establishment, condition or change in any aeronautical facility, service, procedure or hazard, the timely knowledge of which is essential to systems and automated equipment used by personnel concerned with flight operations*"

(EUROCONTROL, 2010). Although the focus is set on automated systems, humans will still be addressed by NOTAMs and therefore the information can be transformed to a human readable textual and graphical representation (EUROCONTROL & Federal Aviation Administration, 2011b, p. 6). Digital NOTAMs do not only comprise a simple conversion to a more structured format. It is a shift in the paradigm. Temporary or permanent information updates will not be encoded within an own data structure. They are integrated within the information of longer duration utilizing the same data structures and distribution channels (EUROCONTROL, 2015b).

The encoding for digital NOTAMs was developed in cooperation between FAA and EUROCONTROL with the support of the international AIS community and the International Civil Aviation Organization (ICAO) (EUROCONTROL, 2010). It is based on the previously Aeronautical Information Exchange Model (AIXM) version 4.5, which can be used to model ATM elements such as aircraft, runways, etc. (EUROCONTROL & Federal Aviation Administration, 2011b). Prior versions of AIXM were already used by central database of EAD and locally in different States but with limited modeling, temporal and geographical capabilities (EUROCONTROL, 2010). A more detailed description of AIXM will be provided in Section 2.3.

Digital NOTAMs will not replace the textual NOTAMs immediately, for many years they will be issued parallel with classical NOTAMs (EUROCONTROL & Federal Aviation Administration, 2011b, p. 6). The implementation of digital NOTAMs will follow an incremental approach, where the most important and mostly used types of NOTAMs will be supported first (EUROCONTROL & Federal Aviation Administration, 2011b, p. 6).

Besides overcoming the drawbacks of textual NOTAMs, digital NOTAMs will be also used to improve the efficiency and enhance global civil aviation safety (EUROCONTROL, 2015c). The information within digital NOTAMs can be automatically plotted for visual representation. Temporal aspects such as schedules can be computer interpreted, cross references within static data are provided and transformations between different formats and/or textual or graphical output will be supported. Moreover, automatic integration between computer systems will be provided and it will be possible to conduct complex queries in order to select the most recent NOTAMs which fulfill user-specified criteria (EUROCONTROL, 2010).

Digital NOTAMs will allow automatic checks due to their machine-readable structure. This will improve coherence and correctness and enhance the overall data quality (EUROCONTROL, 2010). Since a graphical data representation is also provided, visual checks by human operators will be supported too (EUROCONTROL, 2010). With these checks, missing and wrong data can be easily detected which again contributes to an improved data quality. Besides that, digital NOTAMs will facilitate an accurate situation awareness based on shared and up-to-date data sets (EUROCONTROL, 2010). Situation awareness encompasses the perception of elements within an environment with respect to time and space; the comprehension of their meaning and the projection of their consequences into the near future (Baader et al., 2009). It allows to understand what is happening, how information and actions could impact goals now and in the future in order to avoid faulty decisions. Furthermore digital NOTAMs will be able to trigger automated actions which result out of hazards or other events (EUROCONTROL & Federal Aviation Administration, 2011b, p. 6).

2.3 Aeronautical Information Exchange Model

As mentioned before, in order to realize the concept of digital NOTAMs a comprehensive data model for aeronautic information was needed. Therefore the AIXM version 5.1 was developed by EUROCONTROL and FAA. The first version of the AIXM specification was developed to support the distribution and encoding of aeronautical information which is provided by the AIS (EUROCONTROL, 2015a). This data-centric approach was needed in order to ensure data quality, real-time information, efficiency and cost effectiveness of AIM systems (EUROCONTROL & Federal Aviation Administration, 2006). Besides that, it provides one data source used by different systems enabling interoperability.

The key concepts and the extensions specified in AIXM 5.1 are depicted in Figure 2.4. In contrast to AIXM version 4.5, which focused just on the encoding of static data, AIXM 5.1 supports dynamic data as well (EUROCONTROL, 2006b). Dynamic data encoding is supported by a temporality model which captures different states of features and their properties during their lifetime. Since AIXM 5.1 is built in a modular way, single modules can be simply re-used or expanded by new features, properties, domain values or message types. This allows using all the benefits of the complex standard, while still considering local interests without affecting

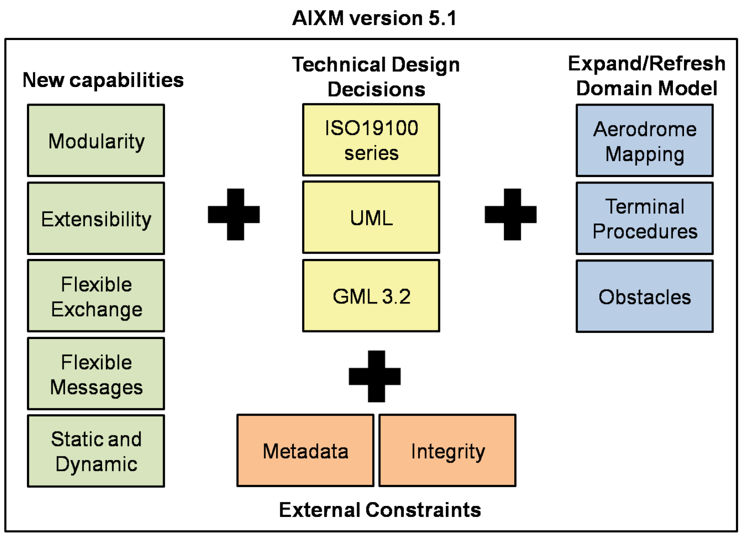

Figure 2.4: Key concepts and extensions of AIXM version 5.1 (EUROCONTROL & Federal Aviation Administration, 2006).

the global interoperability (EUROCONTROL, 2006a). Furthermore, AIXM 5.1 includes International Organization for Standardization (ISO) standards for geospatial information such as ISO19100 providing a framework for developing geography-based domain specific standards and providing metadata about geographic information. Moreover it includes and extends the Geography Markup Language (GML) XML Schema version 3.2 which is an international standard for exchanging geographical features.

AIXM is compliant with the ATM Information Reference Model (AIRM). Besides AIXM, the ATM provides additional data exchange models such as the Flight Information Exchange Model (FIXM)[2] used for sharing information about flights throughout their life-cycle and the Weather Information Exchange Models and Schema (WXXM)[3] for meteorological information (Burgstaller et al., 2015). AIRM satisfies the need of ensuring semantic interoperability within ATM by providing understandable, clearly, structured, harmonized, and uniquely defined ATM business term definitions (EUROCONTROL, 2012, p. 1). It contributes to the System Wide Information Management (SWIM) ensuring that information is not altered or lost while it is used (EUROCONTROL, 2012, p. 1)(Burgstaller et al., 2015).

[2]http://www.fixm.aero/
[3]http://www.wxxm.aero/

2.3.1 eXtensible Markup Language

The data structure behind AIXM is XML which is ideal for data sharing between different systems. XML is used for representing semi-structured information such as documents, configurations or data. It is a simple text-based format used for sharing structured information (W3C, 2015). The main advantages of XML is that it is self-describing, independent of programming languages, data base structure and hardware and it can be easily extended (EUROCONTROL, 2006c, p. 2). XML represents a hierarchical structure including elements with attributes. These XML elements can contain text or other XML elements. Every XML element must be closed by an appropriate XML end element in order to be considered as well-formed. Well-formed XML documents assure that the document is syntactically correct. However, it is also possible to validate an XML document against a given XML Schema. As stated by (W3C, 2015), na XML Schema represents a description of a type of XML document. There are various different languages such as Schematron, Relax-NG, Document Types Definitions, and, the most used, XML Schema Definition (XSD). XML Schemas are used to associate types with values found in the XML documents; furthermore it is possible to define a list of allowed elements and their attributes. Besides that it is possible to constrain where elements and attributes can be used and what content is allowed within those elements. However the most important property is to use the XML Schema as a human-readable as well a machine-processable documentation including formal descriptions of the document.

2.3.2 AIXM Conceptual Model and XML Schema

The AIXM specification is large and complex, covering various aeronautical data of ATM elements, their attributes and their relations. It provides over one hundred different features and data types. However, two main parts can be identified. The logical information model and a data exchange format (EUROCONTROL, 2015a). Both parts can be and are still extended and provide extensive capabilities supporting temporal models and geographical data.

AIXM Conceptual Model: the conceptual model of the aeronautical data was specified using the visual Unified Modelling Language (UML) which allows to describe behaviour, relationships and abstract concepts (EUROCONTROL, 2006a). It is widely used and a de facto modelling standard with an extensive tool support allowing to convert UML diagrams to programming languages, database structures or XML schemas. UML is therefore used to model concepts defined in the Aeronautical Information Conceptual Model (AICM) (EUROCONTROL, 2006c). The AICM represents the logical basis for AIM databases and provides a common conceptual understanding. Comprised concepts amongst other are aerodromes, airspaces, NAVAIDs and fixes, routes, procedures and organizations and services:

- The Airspace Concept represents a region of any three dimensional space in the air with aeronautical significance such as restricted areas or air traffic control sectors (EUROCONTROL, 2006c, p. 17). An airspace can be defined by a single airspace using polygons including the range of the altitude and horizontal borders or combined by other primitive airspaces.
- The area describing the structure of airport and heliports can be specified using the Aerodrome Concept. This covers definitions of runways, taxiways, limitations, obstacles, aircraft parking places and airport time tables.
- In order to define significant points in space which are needed for air traffic control and navigational purposes the NAVAID and Designated Points Concept is used. NAVAIDs are also used to provide landing aids (EUROCONTROL & Federal Aviation Administration, 2006).
- Routes for flights are defined by a series of significant points defining route segments (EUROCONTROL, 2006c, p. 20). Each route segment contains information of the minimum en-route altitude, traffic flow restrictions and operating hours which is all comprised in the Routes Concept.
- Standard arrival routes, departure procedure and instrument approach procedures are defined within the Procedure Concept considering operating hours of the aerodrome and obstacles (EUROCONTROL, 2006c, p. 21). Again significant points are used to define procedure segments.
- Information about aerodromes, procedures, NAVAIDs and airspaces are provided by various services (EUROCONTROL, 2006c, p. 22). In order to describe divisions, units, organizations and their provided services, the Service Concept is used. Furthermore this concept allows to model connections between aeronautical elements which are provided by the services.

Figure 2.5 depicts how these conceptual areas are modelled within UML using features, attributes and associations. They represent important

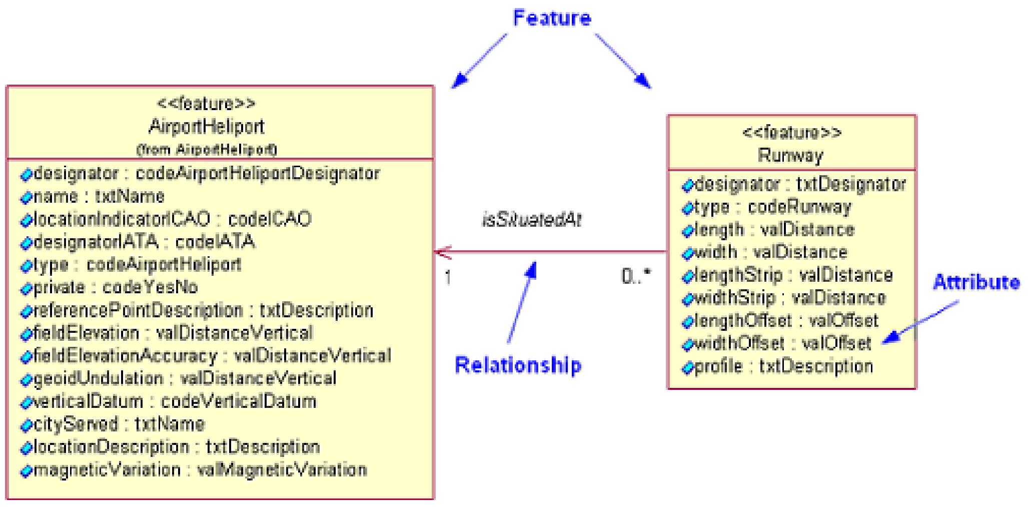
Feature
<<feature>>
AirportHeliport
(from AirportHeliport)
designator : codeAirportHeliportDesignator
name : txtName
locationIndicatorICAO : codeICAO
designatorIATA : codeIATA
type : codeAirportHeliport
private : codeYesNo
referencePointDescription : txtDescription
fieldElevation : valDistanceVertical
fieldElevationAccuracy : valDistanceVertical
geoidUndulation : valDistanceVertical
verticalDatum : codeVerticalDatum
cityServed : txtName
locationDescription : txtDescription
magneticVariation : valMagneticVariation
isSituatedAt
1
0..*
Relationship
<<feature>>
Runway
designator : txtDesignator
type : codeRunway
length : valDistance
width : valDistance
lengthStrip : valDistance
widthStrip : valDistance
lengthOffset : valOffset
widthOffset : valOffset
profile : txtDescription
Attribute

Figure 2.5: UML example of an conceptual area (EUROCONTROL & Federal Aviation Administration, 2006).

aeronautical entities such as runway, aerodromes or routes. Since they are central entities they are modelled as UML classes (EUROCONTROL & Federal Aviation Administration, 2006). Each of them is characterized by various elements such as the boundaries of runways. Multiple features can be associated to each other using relationships indicating for example that a specific runway is situated at an aerodrome (EUROCONTROL & Federal Aviation Administration, 2006). Moreover, AIXM allows to define plausibility checks on the data and business rules.

XML Schema: after defining the concepts within UML, an exchange model to transfer them is needed. Such an exchange model for aeronautical data is provided by the AIXM XML Schema which is defined using XSDs. It is derived from the AIXM Conceptual Model and allows system-to-system exchange of aeronautical information. Due to its size and complexity it is not addressing flight personnel, instead it ensures that automated systems can communicate correctly with each other. As mentioned before, XSDs are used to define attributes, associate them with types and model simple or nested elements. Basically, the AIXM exchange format consists of three main UML files. The first one[4] defining AIXM base feature and object constructs such as abstract features, time slices, objects and property types. The second one[5] defining data types used within the UML model and the third one[6] representing all other features and their properties.

2.3.3 Temporality Model

Since each aeronautical feature instance is affected by time, the AIXM provides a model to consider temporality. Every feature has a lifetime which is determined by its start and end of life (EUROCONTROL & Federal Aviation Administration, 2010, p. 5). Moreover, the features properties and the relationships to other features can change during the lifetime. This can result in properties which are not defined over a time period. A key assumption of the temporality model is that each property and relationship can change permanently or temporally, except the global unique feature identifier. It consists of two main components. Events which represent changes of one or more feature properties and states which represent valid feature properties over a time period (EUROCONTROL & Federal Aviation Administration, 2010, p. 6). Events occur during transitions between states. So called "Time Slices" are used to describe feature properties during states and events.

As depicted in Figure 2.6, a time slice is a container including all time varying properties of the features. Moreover, it defines how long a value is set for which property. Since each feature can contain multiple Time Slices, the sequence number is used as an identifier for them within a feature (EUROCONTROL & Federal Aviation Administration, 2010, p. 11). This sequence number also allows to update or replace already communicated Time Slices. As changes can be conducted temporally or permanently, different types of Time slices are provided. A BASELINE Time Slice describes the state of a feature which results of a permanent change (EUROCONTROL & Federal Aviation Administration, 2010, p. 12). In contrast to that, a TEMPDELTA Time Slice is used to define a temporal feature state. In order to describe the differences of a feature state which results of a permanent change, PERMDELTA Time Slices are

[4]`http://www.aixm.aero/gallery/content/public/schema/5.1/AIXMAbstractGMLObjectTypes.xsd`

[5]`http://www.aixm.aero/gallery/content/public/schema/5.1/AIXMDataTypes.xsd`

[6]`http://www.aixm.aero/gallery/content/public/schema/5.1/AIXMFeatures.xsd`

used. To communicate the current status of a feature, the baseline data and any active temporal data needs to be merged; this is provided by the SNAPSHOT Time Slices.

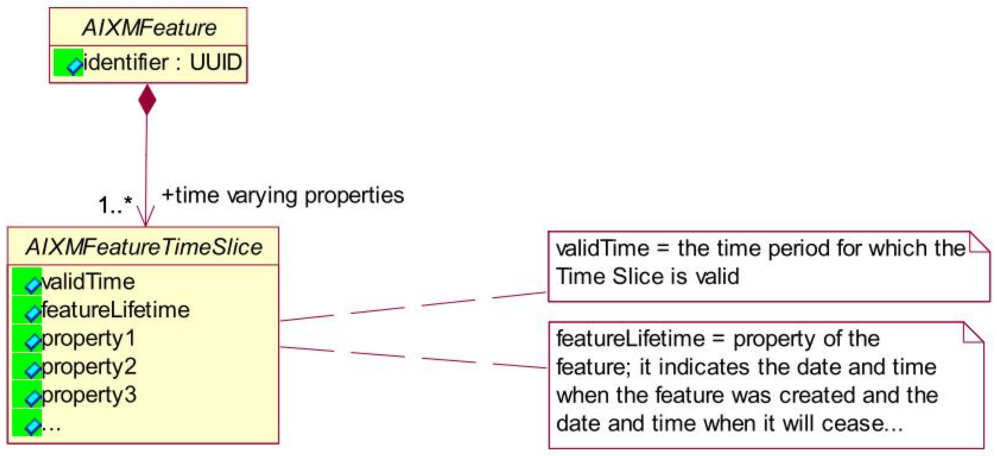
AIXMFeature
identifier : UUID
1..*
+time varying properties
AIXMFeatureTimeSlice
validTime
featureLifetime
property1
property2
property3
...
validTime = the time period for which the Time Slice is valid
featureLifetime = property of the feature; it indicates the date and time when the feature was created and the date and time when it will cease...

Figure 2.6: UML concept of a time slice (EUROCONTROL & Federal Aviation Administration, 2006).

Time Slices describe the features and their properties for a given time period. However there are properties which do not contain constant values during their validity of time (EUROCONTROL & Federal Aviation Administration, 2010, p. 13). Properties can have cyclic varying values with an associated timetable. This timetable describes the times when a given value is used for these properties. To incorporate these timetables, the temporality model provides the concept of schedules. These schedules avoid the generation of a BASELINE or/and TEMPDELTA time slices every time the feature property state is changed.

The dynamic content of NOTAMs can be represented by this temporality concept. Time Slices and schedules consider the fact that the content of NOTAMs can change during their life-time. Based on this, dynamic information can be encoded in a structured and machine readable way which is needed for the SemNOTAM system (Section 2.1).

2.3.4 Geographic Model

The geographical model of AIXM is based on the GML Encoding Standard in order to express geographical features. GML can be used as a modelling language, as well as an interchange format for geographic data (EUROCONTROL, 2006a). Using the GML Schema, it is possible to express geometries such as lines, curves, points, polygons or surfaces.

It is important to notice that the AIXM exchange model is based on a subset of the GML, so called profiles. This leads to AIXM features which are basically GML features and AIXM objects which are GML objects (EUROCONTROL, 2008, p. 12). Furthermore, AIXM follows the GML object-property concept which prohibits that a GML object has an immediate child representing another GML object (EUROCONTROL, 2008, p. 12). Applied to AIXM, this means that an association between two features respectively between a feature and an object must be established over a property of the feature (EUROCONTROL, 2008, p. 12). The implementation of the object-property concept is realized by declaring a type and assigning properties to it. These properties include attributes as well as relationships. The declared type will be then assigned to features (EUROCONTROL, 2008, p. 12).

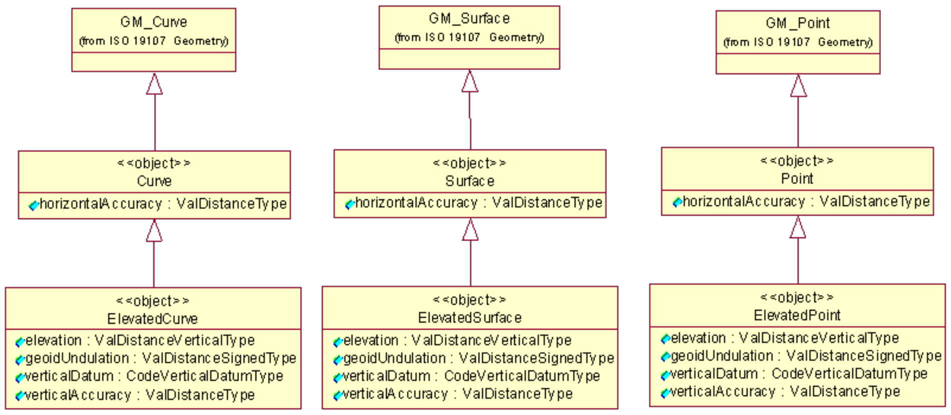
GM_Curve
(from ISO 19107 Geometry)
GM_Surface
(from ISO 19107 Geometry)
GM_Point
(from ISO 19107 Geometry)
<<object>>
Curve
horizontalAccuracy : ValDistanceType
<<object>>
Surface
horizontalAccuracy : ValDistanceType
<<object>>
Point
horizontalAccuracy : ValDistanceType
<<object>>
ElevatedCurve
elevation : ValDistanceVerticalType
geoidUndulation : ValDistanceSignedType
verticalDatum : CodeVerticalDatumType
verticalAccuracy : ValDistanceType
<<object>>
ElevatedSurface
elevation : ValDistanceVerticalType
geoidUndulation : ValDistanceSignedType
verticalDatum : CodeVerticalDatumType
verticalAccuracy : ValDistanceType
<<object>>
ElevatedPoint
elevation : ValDistanceVerticalType
geoidUndulation : ValDistanceSignedType
verticalDatum : CodeVerticalDatumType
verticalAccuracy : ValDistanceType

Figure 2.7: AIXM extensions of the GML (EUROCONTROL & Federal Aviation Administration, 2007).

As depicted on figure 2.7, AIXM uses a 2.5 Dimension (2.5D) representation of the GML geometry model (EUROCONTROL & Federal Aviation Administration, 2007). A 2.5D representation is used in order to encode geometric elements above the earth's surface. Therefore the vertical distance from the Mean Sea Level to the to the highest point on the geometry is added as the elevated property "elevation". Furthermore the height of the geometry is encoded within the vertical extent. All the properties are defined regarding a vertical accuracy. The definition of 2.5D geometries is used in AIXM for ATM elements such as airspaces or routes.

As mentioned in Section 2.1.3, the query interfaces allows to consider criteria such as the location in space. Therefore a geometry is specified as input. The geometry is analysed in order to detect an overlapping with geometries of NOTAMs. When the geometry of a NOTAM intersects the specified geometry, then the NOTAM is relevant. The detection of geographic intersection is done by checking whether the specified geometry is intersected horizontally or vertically by any other NOTAMs geometry. Since geometries can encode polygons, curves or complex surfaces, this is not a trivial task. To ease the computational effort GML allows to model envelopes. These envelopes define the bounding box of a geometry. A bounding box is the smallest horizontal rectangle comprising all the points of a geometry.

Using the capabilities of GML, AIXM is able to cover all relevant geographical elements needed within the ATM. Furthermore it provides more detailed geometries definitions than used in prior versions of AIXM. This improves the filtering capabilities since the intersection works with more accurate data than using only the radius of influence. Thus intersections of irrelevant NOTAMs are avoided.

2.3.5 Feature Identification and Reference

As mentioned in Section 2.3.3, each AIXM feature is identified through the use of the identifier property. This property is the only time-invariant one and therefore it is situated outside the TimeSlice complex object (EUROCONTROL & Federal Aviation Administration, 2011a, p. 5). In order to provide feature identification, the AIXM 5.1 schema relies on Universal Unique Identifiers (UUIDs) as feature identifiers. These identifiers fulfill two essential requirements; they are unique and universal, which means that the same identifier will never be used unintentionally by anyone else and that the same identifier can be used in all systems to identify a given AIXM feature. It is important to notice that an identifier does not identify a feature itself, it identifies the data that someone has about a feature. When two systems use the same identifier for a feature, it indicates that the data is retrieved from one source or that there are processes ensuring consistency between the data of the two systems.

Since AIXM is used to distribute information to various consumers, the possibility to manage linkages between aeronautical features is supported. Therefore the AIXM schema utilizes the XML Linking Language (XLink) schema which is bundled within GML (EUROCONTROL & Federal Aviation Administration, 2011a, p. 10). XLink is a standard for representing a reference between two XML elements, which represent AIXM features. It allows to address individual XML elements which are located within the document or available over external sources.

The capability to uniquely identify an AIXM feature allows referencing it. Supporting references avoids redundancy and improves consistency since data can be maintained at one place and referenced from anywhere.

2.4 ObjectLogic

Besides the AIXM representation for structured data, the SemNOTAM system requires a knowledge-based representation. Therefore the deductive and object oriented database language OL is used. OL combines the expressiveness and declarative semantics of deductive data base languages with the modelling capabilities supported by object orientation. It is a successor of F-Logic which was developed by (Kifer, Lausen, & Wu, 1995) in 1995. F-Logic is an ontology and knowledge representation language used for ontology management, semantic web services, information integration and intelligent agents. Since F-Logic was developed to overcome the relational approach of predicate logic, it already comprises the object-orientated concepts which are used in OL. Since OL is also used to represent knowledge it provides rules in order to derive new knowledge. Furthermore queries for filtering that knowledge base are supported.

OL is based on the Closed World Assumption (CWA). This assumption is crucial for the reasoning task, since a CWA implies that all available knowledge is complete. Everything which is not known is assumed to be false e.g. an aircraft has a pilot named Bob, the reasoner would determine that there is only one pilot for that aircraft. In contrast to that the Open World Assumption (OWA) implies that all available knowledge is true but not complete. This means for the given example that the aircraft has a pilot named Bob but it can have other pilots too, it is just not known. This difference seems slight, but when a query is conducted to filter aircraft with only one pilot, then only the CWA based reasoner will return the aircraft.

The tool support for OL is restricted to OntoStudio and OntoBroker since OL exclusively provided by semafora systems GmbH. The OntoBroker is the reasoner for OLOL and includes the knowledge base and an API (OntoAPI) in order to access it. The OntoStudio provides a graphical interface for the supported operations of the OntoAPI.

The following description provides an insight of the OL concepts and examples explaining them. Since semafora systems GmbH are exclusively developing OL, they are the only source of documentation for it. Therefore the following explanations refer to the OL Tutorial (semafora systems GmbH, 2012) and the OL Reference (semafora systems GmbH, 2013).

Basic Syntax: Factual knowledge is defined in OL by objects, their relationships, and classification. This kind of knowledge is also known as the explicit knowledge. In contrast to that the intensional knowledge represents the implicit knowledge derived by applying rules. Both the intensional and the factual knowledge can be queried. All these entities of an OL program which can be queried must be named.

```
Jet.                    // term as class name
Jet[maxSpeed -> 800]    // term as method name
jet123_3:Jet.           // term as object name

?- eurofighterX[crewMember -> ?Y]. // term as query
    variable ?Y
```

Listing 2.1: Example of terms.

As shown in Listing 2.1, OL provides terms for naming entities. Terms can be used as constants or to name classes, object or methods. Terms used to name classes, objects or methods are called id-terms and start with a letter followed by letters, digits or underscores. Terms are also used for variables, which are only used within rules and queries. They follow the same grammar as id-terms except that they start with a question mark. Moreover terms can represent methods including the name and a list of one or more terms within braces.

Schema Level Statements: OL supports the object-oriented concepts of classes and attributes. Therefore the schema provides a vocabulary allowing to define classes, attributes and applicable methods. Using this vocabulary, OL allows to define complex class hierarchies. Furthermore various methods and class attributes can be defined this way, providing thereby type-safety and constraints regarding the cardinalities.

Subclass-F-atoms allow to define subclass relationships between two classes. The classes are denoted by id-terms. Multiple inheritances of several classes are permitted in OL.

```
Jet.
Jet::Aircraft.
```

Listing 2.2: Example of a subclass-F-atom.

As shown in Listing 2.2, a class "Jet" is defined. The subclass-F-atom is used to define hierarchical structure declaring that a "Jet" is a subclass of an "Aircraft". Subclass relationships are denoted by "::".

Signature-F-atoms are used to define methods of a class and type restrictions for parameters and results. Moreover, each method can be restricted by a given cardinality, which determines how many entries may be provided at least respectively at most. It is important to note that attributes of a class are modelled as methods without any parameter. Therefore methods without any parameter will be named attributes in the following sections.

```
Pilot::Person.
Jet[type {1:1} *=> xsd#string].
Jet[pilot(xsd#date) {1:*} *=> Person].
Jet[pilot(xsd#date) {1:*} *=> Pilot].
```

Listing 2.3: Example of several signature-F-atom.

The Listing 2.3 starts with defining a class "Pilot" which extends "Person" using a subclass-F-atom. The first signature-F-atoms defines for the domain "Jet" an attribute named "type" with an range "xsd#string". Moreover, the cardinalities indicate that there must be exactly one "type" provided. The domain defines which class is affected and the range determines which types respectively classes can be used as values. The following two signature-F-atoms define the method "pilot" which can be used to determine the pilot(s) of a jet. As shown, the method "pilot" accepts a parameter of the type "xsd#date" indicating since when a given pilot is the pilot of the jet. Furthermore, these two signature-f-atoms show also that the concept of method overloading is provided since both share the same method signature. The only difference is determined by the specified range.

Instance Level Statements: Based on a defined schema, instance level statements can be used to create objects. These objects represent instances of the predefined classes. However a valid OL program does not need to have existing schema level statements. It is also valid without them, when the basic syntax is not violated.

F-atoms are also know as isa-F-atoms which are used to instantiate classes. With isa-F-atoms the class membership is denoted by a single colon separating two id-terms, representing the instance (object) and the class. OL allows the multiple instantiation of several classes.

```
eurofighterX:Jet.
eurofighterY:Jet.
```

Listing 2.4: Example of a isa-F-atom.

A simple instantiation of one class "Jet" is depicted in Listing 2.4.

Data-F-atoms are used to apply methods on objects which were instantiated using f-atoms. Data-F-atoms represent the instance of signature-F-atoms and therefore they consist of a host object (domain), a method and a value which denoted by id-terms. The value can be either an object or a literal.

```
eurofigtherX[type -> "Eurofigther Typhoon"].
mary:Person.
bob:Pilot.
eurofigtherX[pilot(2012) -> mary].
eurofigtherX[pilot(2011) -> bob].
```

Listing 2.5: Example including several data-F-atoms.

Listing 2.5 shows how the methods of the previously instantiated object "eurofigtherX" are used. The attribute "type" is set to "Eurofigther Typhoon"; moreover, a person "mary" and a pilot "bob" are instantiated.

It would also be possible to multiple instantiate "bob" using other OL classes. These two individuals are used for the usage of the objects' "pilot" methods. Both methods receive a year as input parameters.

F-molecules allow to collect information about an object, by combining multiple f-atoms statements. Furthermore f-molecules can also be used to combine subclass-F-atoms and signature-F-atoms. The previous OL examples can be rewritten to the following OL program 2.6 based on f-molecules.

```
// Schema Level statements
Pilot::Person.
Jet[type {1:1} *=> xsd#string, pilot(xsd#date) {1:*}
    *=> {Person, Pilot}].

// Instance Level statements
mary:Person.
bob:Pilot.
peter:Person.
anna:Person.

eurofighterX:Jet.
eurofighterY:Jet[type -> "Eurofigther Typhoon",
    pilot(2012) -> mary, pilot(2011) -> bob,
    controller -> {peter, anna}].
```

Listing 2.6: Example of a F-molecule.

F-molecules allow to combine the data-F-atoms and isa-F-atoms which results in a more readable OL program. An object and its corresponding methods and attributes can be defined within only one statement. The Listing 2.6, is enriched by the attribute "controller" which indicates who is the responsible air traffic controller. As stated before, even if that attribute is not defined in the schema, it is still valid. Furthermore it can be seen that multiple values can be assigned to one attribute at once. When more than one values is set for an attribute at once, the values must be enclosed within braces. The usage of the method "pilot" of the "eurofighterY" object cannot be grouped to one statement since each of them have a different parameter. If both methods would receive the same parameter "2012" then a grouping to "*pilot(2012) -> {mary, bob}*" would be possible.

Namespaces: a term can be used multiple times representing different semantics. Besides using a term multiple times, a term can have different meanings (synonyms). Furthermore, the same id-term can be used in multiple knowledge bases which can lead to problems when they get merged. To overcome these issues, OL provides namespaces. Namespaces are used in the context of the Semantic Web to uniquely identify objects and instances. Therefore they can also be used to handle multiple definitions of concepts (classes) in different knowledge bases. Each namespace must represent a valid identifier according to RFC 2396 and must end with either "#", "/", or ":". These last characters mark the separator between the namespace and the local part of an identifier.

```
:- prefix aero = "httm://www.aixm.aero/schema/5.1#".
airlinerX:aero#Aircraft[aero#id -> "Boeing747-123"].
```

Listing 2.7: Example of using a namespace.

As shown in Listing 2.7, it is possible to define prefixes for namespaces. These prefixes act as a place holder for the namespace, avoiding to rewrite the whole namespace identifier before each term. When a namespace respectively a prefix is used to identify a term, then the separator symbol "#" must be placed between the prefix and the term (local name).

Rules: OL rules are used to extend the knowledge base by deriving new information. Whenever a precondition of a rule is satisfied, the conclusion is true. The preconditions represents the rule body and is formed by combining F-molecule by logical conjunction such as OR, NOT and AND. The rule head is also represented by a conjunction of F-molecules and is separated from the rule body by the symbol ":-". Variables are used within rules as place holders. However, variables used in the rule head must also be used in the rule body.

```
?X[coPilot -> ?Y] :- ?X:Aircraft[flightCrew -> ?Y]
    AND ?Y:Pilot AND NOT ?X[captain -> ?Y].
```

Listing 2.8: Simple rule defining the aircrafts' co-pilot.

Listing 2.8 depicts a simple rule. The rule head defines that a object "?X" has the attribute "coPilot" with the value "?Y" when the rule body is satisfied. The rule body comprises the conditions which need to be fulfilled; in this case the variable "?X" must be an aircraft and "?Y" must be a member of the flight crew. Furthermore, "?Y" must be a pilot who is not the captain of the aircraft "?X". Using this rule the implicit knowledge that "a flight crew member who is a pilot but not the captain of the aircraft is assigned as a co-pilot" can be derived.

Queries: To access the explicit and derived knowledge, OL provides queries. Syntactically a query can be considered as a special kind of a rule with an empty head. Queries start with the symbol "?-" followed by conditions which need to be satisfied. These conditions are denoted by f-molecules including unbound variables. The unbound variables are used to determine which objects are queried respectively which object will be in the result set. Queries retrieve results based on derived and explicit available facts.

```
?- eurofighterX[crewMember -> ?Y].
```

Listing 2.9: Example a simple query.

The query depicted in the example 2, retrieves all crew members of the jet "eurofighterX".

Chapter 3

Requirements Analysis

Contents

This section specifies the requirements of the mapping task. Therefore the task description will be detailed and examined. Based on this the requirements will be determined and analysed. Moreover, it provides an insight of possible problems which can occur. The identified requirements will be then used in Section 5 to design an appropriate architecture and to select suitable technologies.

3.1 Task Description

A coarse description of the mapping task is given in Section 1.1 and Section 2.1. It is stated there that the mapping will transform a given structured data set into a knowledge representation. In the subsequent sections this task is substantiated to a mapper between an XML-based and an OL representation. This section will specify this task by providing detailed description of the data sources, their usage and their structure.

The main task of the mapping is the transformation of several structured input sources to an OL representation. These input sources are depicted in figure 3.1 and represent the data structures introduced in the SemNOTAM system (Section 2.1). Each of the depicted data sources will be processed sequentially.

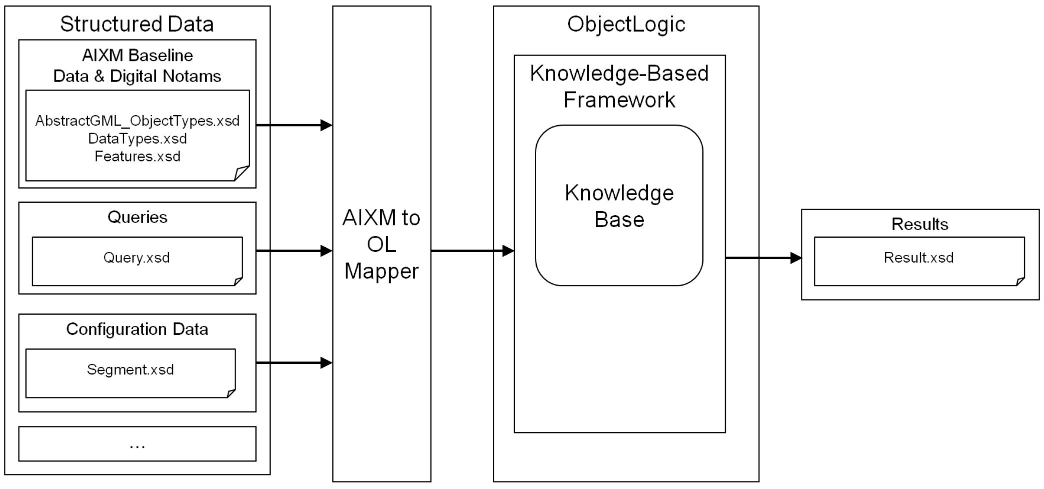

Figure 3.1: Mapping tasks of the AIXM to ObjectLogic Mapper.

Since the XML schemas are still being developed, the available data sources are restricted to NOTAMs, segments and queries. However, other sources such as aircraft data will be supported in the future which can lead to new data sources. Each data source is described through XSD document(s).

The three sources depicted in Figure 3.1 must be handled by the mapper. The AIXM to OL Mapper must accept these three XML documents which will be parsed to corresponding OL files. The resulting segment, query and NOTAM OL files will then be inserted into the knowledge base. This knowledge base is accessible over the OntoAPI of the OntoBroker. The query OL files will be saved and processed by another component. During the mapping process not information must be lost since the data sources, especially NOTAMs, can contain safety critical information. However, information which does not add any knowledge need to be omitted to avoid clutter. That are XML attribute elements which are empty, contain no XML attributes and do not represent an association.

Furthermore, it must possible to configure the mapper. Examples for configuration parameters are the source XML file name, the output OL file name, credentials for accessing the OntoAPI and whether a new ontology will be created or an existing extended. The mapper has to perform preprocessing tasks, such as the calculation of the bounding box of AIXMBasicMessages and their features. This will ease later computational steps on OL. Besides these functions, the time at which the NOTAMs have been processed will be stored within a detection time stamp. This time stamp is needed for so called delta queries which should determine changes in the knowledge base. These changes are relevant because they indicate updates of already mapped data sources.

Since not all sources and therefore possible preprocessing steps are known, the parser must be extensible without changing the existing implementation. Possible preprocessing steps in the future could be the classification of NOTAMs according to their corresponding flight information region, air traffic service (ATS) segments, or airport. Other classifications could include a specific standard instrument departure (SID) or/and standard terminal arrival route (STAR).

The following subsections details the three data sources depicted in Figure 3.1. For each data source the structural representation encoded in XML will be analysed. Therefore it will be examined which information is stored in which XML elements. Moreover, possible content of the XML elements and their corresponding OL representation will be introduced.

3.1.1 NOTAM Data

NOTAMs in AIXM are represented by `AIXMBasicMessages`. The `AIXMBasicMessage` is available as XML data and will be transformed into OL. An `AIXMBasicMessage` contains multiple AIXM features which are included within the `hasMember` association.

```xml
<ns13:AIXMBasicMessage ns5:id="FNS_ID_34828025">
  <ns5:boundedBy xsi:nil="true"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" />
  <ns13:hasMember>
    <ns10:Event ns5:id="Event_1_34828025">
      <ns5:boundedBy xsi:nil="true"
```

```xml
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" />
      <ns10:timeSlice>
        <ns10:EventTimeSlice ns5:id="Event_TS_1_34828025">
          <ns5:validTime>
            <ns5:TimePeriod ns5:id="Event_TS_TP_1_34828025">
              <ns5:beginPosition>2013-02-12T21:51:00.000Z</ns5:
                  beginPosition>
              <ns5:endPosition indeterminatePosition="unknown" />
            </ns5:TimePeriod>
          </ns5:validTime>
          <ns6:interpretation>BASELINE</ns6:interpretation>
          <ns10:scenario>6000</ns10:scenario>
          <ns10:extension>
            <ns11:EventExtension ns5:id="ext_01_34828025">
              <ns11:classification>INTL</ns11:classification>
              <ns11:accountId>KIAD</ns11:accountId>
              <ns11:xoveraccountID>FDC</ns11:xoveraccountID>
              <ns11:xovernotamID>3/8008</ns11:xovernotamID>
              <ns11:airportname>WASHINGTON DULLES INTL</ns11:
                  airportname>
              <ns11:originID>KIAD</ns11:originID>
              <ns11:lastUpdated>2013-09-30T10:50:00.000Z</ns11:
                  lastUpdated>
              <ns11:icaoLocation>KIAD</ns11:icaoLocation>
            </ns11:EventExtension>
          </ns10:extension>
        </ns10:EventTimeSlice>
      </ns10:timeSlice>
    </ns10:Event>
  </ns13:hasMember>
</ns13:AIXMBasicMessage>
```

Listing 3.1: Example of a NOTAMs encoded as an AIXMBasicMessage.

AIXM features can be used to represent events introduced in Section 2.2.1 and Section 2.3.3. As depicted in Listing 3.1, the `AIXMBasicMessage` contains an `Event` as a member. Since the values within a NOTAM can vary, a time slice feature for the event is provided (Section 2.3.3). Furthermore, it is shown that identifiers for the feature elements are provided, whereas the associations are not identified explicitly. The `boundedBy` element is empty since no GML information is provided in Listing 3.1. However, GML information can be encoded within NOTAMs. Moreover, it is shown that the `AIXMBasicMessage` which is used for encoding the NOTAMs, follows the object-property concept of GML (2.3.4). Besides XML elements and their attributes a NOTAM can contain text as content. AIXM allows to formate this text using HyperText Markup Language (HTML) tags.

Using the NOTAM in Listing 3.1 as an input, the AIXM to OL Mapper has to produce the corresponding OL output as depicted in Listing 3.2.

```
FNS_ID_34828025:ns13AIXMBasicMessage[ns5id->"FNS_ID_34828025",
    ns13hasMember->Event_1_34828025].

Event_1_34828025:ns10Event[ns5id->"Event_1_34828025", ns10timeSlice->
    Event_TS_1_34828025, ns5boundedBy -> "xsi:nil"].

Event_TS_1_34828025:ns10EventTimeSlice[ns5#d->"Event_TS_1_34828025",
    ns5validTime->Event_TS_TP_1_34828025, ns6interpretation->"BASELINE
    ", ns10scenario->"6000", ns10extension->ext_01_34828025].

ext_01_34828025:ns11EventExtension[ns5#id->"ext_01_34828025",
    ns11originID->"KIAD", ns11xoveraccountID->"FDC", ns11classification
    ->"INTL", ns11airportname->"WASHINGTON DULLES INTL",
    ns11lastUpdated->"2013-09-30T10:50:00.000Z", ns11icaoLocation->"
    KIAD", ns11xovernotamID->"3/8008", ns11accountId->"KIAD"].

Event_TS_TP_1_34828025:ns5TimePeriod[ns5id->"Event_TS_TP_1_34828025",
    ns5endPosition("indeterminatePosition")->"unknown",
    ns5beginPosition->"2013-02-12T21:51:00.000Z"].
```

Listing 3.2: Example of an AIXMBasicMessage represented in an OL representation.

As shown the `AIXMBasicMessage` and its features are represented by f-molecules. Each of these f-molecules is defined using the given identifier, such as "`FNS_ID_34828025`", as the id-term. The XML attributes of the `AIXMBasicMessage` are mapped to OL attributes. These OL attributes are added to the f-molecule using data-F-atoms. Moreover, the associations to the direct AIXM child features are mapped to OL. These associations are set by adding OL methods which receive the id-term of the AIXM child features as a value.

3.1.2 Configuration Data

As stated in Section 2.1.3, configuration data is used to define route segments and aircraft characteristics. Since currently the segments are used as input sources, the aircraft characteristic will be left aside. However, it must be considered that they will be available in the near future. A route consists of several segments (Section 2.3.2). Each segment is defined by a start and an endpoint (Cordell, 2006). Beside these two points, segments are composed of consecutive connected significant points. These significant points can be used by multiple routes. Therefore it is possible to reference them.

```xml
<Collection xmlns="http://semnotam.frequentis.com/SegmentSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://semnotam.frequentis.com/SegmentSchema ./
      SegmentSchema.xsd"
  xmlns:gml="http://www.opengis.net/gml">
  <hasMembers>
        <Segment>
            <id>1212</id>
            <shape>
                <gml:PolygonPatch>
                    <gml:exterior>
                        <gml:LinearRing>
                            <gml:posList>38.9353015690001 -77.458481289
                                38.93530156900009 -77.45848128899996
                                    38.9358698240001
                                -77.458208563 38.9359786850001
                                    -77.45815443
                            </gml:posList>
                        </gml:LinearRing>
                    </gml:exterior>
                </gml:PolygonPatch>
            </shape>
        </Segment>
        ...
    </hasMembers>
</Collection>
```

Listing 3.3: Example of segments encoded in XML.

As shown in Listing 3.3, segments can be encapsulated in a collection of segments. Each segment is identified by an identifier and contains a GML shape. This GML shape consists of several coordinates representing a geometry. Segments follow object-property concept of GML since they are based on GML shapes. In Listing 3.3, the represented geometry is a linear ring.

3.1.3 Query Data

The parsing of the query will neither lead to a resolution of the query nor to the correct interpretation of it. It will transform the XML document containing the query to its corresponding OL representation. The resolution of the query and the interpretation of it will be done in a separate module.

As stated in Section 2.1 the query will support the combination of various interests. As depicted in Figure 3.2 these interests are used to specify queries regarding aircraft, attributes, periods or areas. It is important to note that the area of interests can use the information provided by segments. Like in the other data source, the object-property model is followed since the query can be used to query GML information.

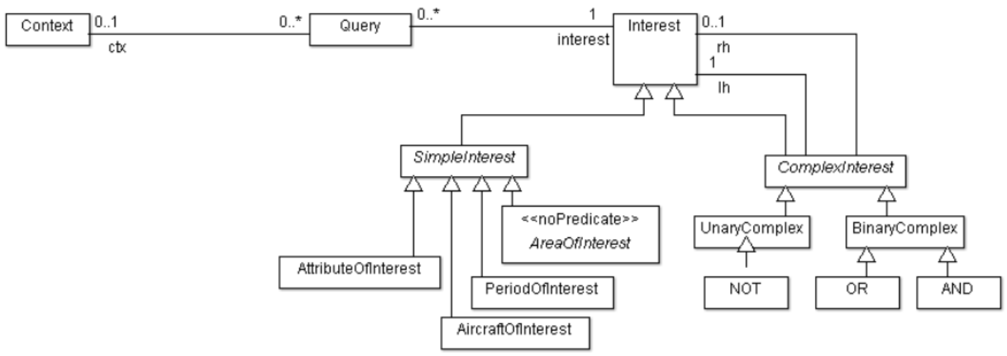

Figure 3.2: Query object structure (Steiner et al., 2014, p. 15).

Due to the high modularity of this structure, each queried aspect is represented by a `SimpleInterest` subclass. These `SimpleInterests` can be combined to `ComplexInterests` using logical connectors such as NOT, OR and AND. AND is used for the intersection of two operands, such as the intersection of two relevant sets. OR is used for the union of two sets and NOT is used as the negation of a referenced set. Each query object contains an identifier which needs to be set by the context.
In the following the different interests and their attributes will be investigated:

- **AircraftOfInterest:** specifies the attributes of the aircraft which can be used to filter NOTAMs which are relevant to the described aircraft.
- **AttributeOfInterest:** refers to AIM elements which are specified in the SemNOTAM system as concepts. Furthermore they can be used to restrict the values of the result. Combinations of concepts and value restrictions are possible.
- **PeriodOfInterest:** defines the time span which will be considered during the filtering. These periods are divided into the two types. The occurrence time which describes the valid time of a NOTAM, and the detection time which describes the time a NOTAM is inserted into the

SemNOTAM system. Therefore every NOTAM has one valid timespan and a detection time stamp which allows querying from any point of view in time.

- **AreaOfInterest:** is used to define restrictions to certain areas, such as Areas, ScopedAreas, GML Shapes, and segments. Area restrictions allow to geographically restrict AIM elements such as retrieving the airport in Linz. ScopeArea restrictions are used to define queries which retrieve NOTAMs of a specific airport while excluding runways. For each exclusion exceptions can be set which allows to exclude all runways except a specific one. GML Shapes and segments can be queried by providing GML data and a buffer. NOTAMs which fit the GML information within the buffer are retrieved.

Besides the querying of current data it is possible to request the changes since the query was last posted to the SemNOTAM system. This is done by using Delta Queries which are a specialization of normal queries. Due to the fact that NOTAMs can be corrected, a correction can lead to NOTAMs which were relevant in the first query but later becoming irrelevant due to the changes.

```xml
<Query xmlns="http://semnotam.frequentis.com/QuerySchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:rs="http
      ://semnotam.frequentis.com/ResultSchema"
  xsi:schemaLocation="http://semnotam.frequentis.com/QuerySchema ./
      QuerySchema3.xsd"
  xmlns:gml="http://www.opengis.net/gml">
  <id>1</id>
  <interest>
    <ScopedArea>
      <scope>
        <Area>
          <concept>
            <AreaClass>Airport</AreaClass>
          </concept>
          <restriction>
            <ValueRestriction>
              <attribute>designator</attribute>
              <operator>=</operator>
              <value>MUC</value>
            </ValueRestriction>
          </restriction>
        </Area>
      </scope>
      <exclude>
        <Exclusion>
          <exclude>
            <AreaClass>Runway</AreaClass>
          </exclude>
          <except>
            <Area>
              <concept>
                <AreaClass>Runway</AreaClass>
              </concept>
              <restriction>
                <ValueRestriction>
                  <attribute>designator</attribute>
                  <operator>=</operator>
                  <value>MUC_RWY10</value>
                </ValueRestriction>
              </restriction>
            </Area>
          </except>
        </Exclusion>
      </exclude>
    </ScopedArea>
  </interest>
</query>
```

Listing 3.4: Example of a simple query encoded in XML.

The query depicted in Listing 3.4, queries all NOTAMs belonging to the airport `Munich`. The airport includes all existing runways. Therefore all runways are excluded from the airport except the runway `RWY10`. It is assumed that the designator attribute is specified at NOTAM level. Moreover, it is assumed that the designator for the airport `Munich` is `MUC` and that the designator for `RWY10` is `MUC_RWY10`.

3.2 Requirements

The task description from Section 3.1 is used in order to determine the requirements. The requirements are determined by identifying and extracting them from the task description. Moreover, they are grouped according their characteristics. This results in a listing containing all requirements which need to be fulfilled by the design respectively the implementation.

1. Input:

   1.1. XML documents have to be processed sequentially.

   1.2. It has to be possible to set the XML-input and OL-output files.
   1.3. The mapper has to allow to define mapping parameters, such as the radius.
   1.4. The credentials for accessing the interface must be configurable.
   1.5. It has to be possible to define whether a new ontology is created or an existing one to be extended.
2. Output:
   2.1. Query specifications have to be mapped to their equivalent OL representation. They will not be enriched or interpreted.
   2.2. Segments have to be mapped to their equivalent OL representation.
   2.3. NOTAMs have to be mapped to their equivalent OL representation.
   2.4. The OL representation has to be saved to a file.
   2.5. An interface for loading OL files into the knowledge base must be provided.
3. Processing:
   3.1. No information must be lost during the mapping task. All features, their attributes, their relationships and the content of the XML elements have to be included in the OL representation. Especially formatted text must be preserved.
   3.2. Local references inside an XML document need to be resolved. This means that instead of the XLink reference statement the identifier of the referred XML elements has to be used. External references do not have to be resolved.
   3.3. The mapper must provide extension capabilities which allow to add new mapping and/or preprocessing tasks without modifying the existing implementation. An extension example is the calculation of bounding boxes (Requirement 4.1).
   3.4. Due to the possibility that the AIXM schema can be extended, the mapper needs consider the different namespaces within the XML documents.

   3.5. A detection time stamp has to be added which represents the time the data source is mapped.

4. Completing missing information:

   4.1. An extension must be provided which completes missing geographical information. If the bounding box of a NOTAM is not available, then it has to be calculated and added to the NOTAM and its features. Bounding boxes for geographic points will be calculated with a given radius.

   4.2. Missing identifier attributes of XML elements, which represent features, have to be added. These identifier must be unique and universal. Unique means that there has to exist a reasonable plausibility that the identifier will never be used unintentionally by anyone else. Universal refers to the requirement, that the same identifier has to be used in all systems to identify a specific AIXM feature.

These requirements will be used in further sections, especially the development of the mapping concept and the selection of the technologies relies on them.

3.3 Challenges and Problems

The requirements describe the main tasks which must be fulfilled by the mapper. However they are not exhaustive since not all aspects were considered. The resulting ambiguities and their emerging challenges and problems will be discussed in this section.

Data identification: The mapping task requires a distinction between AIXM feature and their relationships. The challenge is to define how features respectively relationships are characterized in order to distinguish them. The requirements describe only how they have to be mapped to OL but not how to identify them. An approach must be found which generic enough to be applied on all three data sources. This approach must identify characteristics to distinct between AIXM features and their relationships. It must be ensured that all features and relationships follow the identified characteristics.

Exception handling: Moreover, exceptional XML elements must be determined such as the HTML tags which are used to format text. It must be possible to exclude a set of XML elements which will be ignored or treated as text and therefore not be mapped to an corresponding OL representation. Besides that missing attribute such as the identifier must be handled. It is not exactly specified how the identifier requirements uniqueness and universality can be provided.

OL representation: Since no information has to be lost an approach must be defined which allows to map all information given in the data sources to their corresponding OL representation. The mapping described in the requirements covers only a basic use case. The handling of the attributes of XML elements which represent AIXM relationships is not specified. Moreover, it is not specified how links have to be added to the OL representation. The OL representation of text is limited to simple text without any formatting. However, text can be formatted using HTML tags, tabulators, multiple blanks or line breaks. It is not specified how the formatting has to be mapped in order to preserve the formatting in the OL representation.

Namespaces: the requirements state that the mapper has to consider different namespaces in the data sources especially in AIXM. The handling of namespaces can lead to problems because same prefixes for different namespaces can be used in the data sources. These varying namespace definitions does not conflict within the XML document, but they can lead to conflicts when they are inserted and merged into the KB of the SemNOTAM system.

Preprocessing: Even the defined preprocessing task is challenging since GML elements require a proper handling. The interpretation of coordinates depends on the Coordinate Reference System (CRS). CRSs are required due to the distortions which occur during the plotting of the earths' surface onto a map. The CRS is encoded in the attribute "srsName" which needs to be resolved to find the corresponding CRS. Beside that, calculations based on the geometries must be performed, in order to calculate the bounding box. These calculation again depend on the specified CRS and are not trivial since the earths surface cannot be interpreted as an flat surface. Instead it is represented through an ellipsoid which requires complex trigonometric functions to determine new spatial points. Moreover, it must be possible to transfer coordinates between different CRSs since not all CRS allow these kind of calculations.

Flexibility: Besides the mapping of the data sources to a corresponding OL representation, modification and configuration possibilities must be provided. These modifications are not know currently and therefore the mapper has to be implemented in a way to support them in the future. Therefore a concept must be found to extend the mapper at different points without modifying existing implementations.

Chapter 4

Mapping Approach and Realization Options

Contents

In this section the basic mapping concept and the mapping approach are introduced to fulfill the requirements of Section 3.2. Moreover, the handling of geographic data is analyzed. Based on the mapping concept a suitable technology is chosen. This comprises the analysis and evaluation of the technologies which can be used for the mapping task. Furthermore, the extension capabilities of the selected technology are described in Section 4.4. The mapping concept is shown by using AIXM NOTAMs as it can be applied analogously to all data input sources since they follow the object-property model. Consequently XML attribute elements and AIXM elements which represent OL attributes are used analogously here. This applies also to XML object elements and AIXM features which represent OL objects.

4.1 Basic Mapping Concept

As stated in Section 2.3.2 AIXM follows the object-property model. This model prohibits that an AIXM feature has an immediate child representing another AIXM feature or object. Therefore every relationship between these features must be established over a property/attribute of the feature.

```
<ns13:AIXMBasicMessage ns5:id="FNS_ID_34828025">
<!-- AIXM Feature -->
  <ns5:boundedBy xsi:nil="true" xmlns:xsi="http://www.w3.org/2001/
      XMLSchema-instance" />
  <!-- AIXM Attribute -->
  <ns13:hasMember>
  <!-- AIXM Attribute -->
    <ns10:Event ns5:id="Event_1_34828025">
    <!-- AIXM Feature -->
      ...
      <ns10:timeSlice>
      <!-- AIXM Attribute -->
      ...
              <ns5:beginPosition>2013-02-12T21:51:00.000Z</ns5:
                  beginPosition>
              <!-- AIXM Attribute -->
              <ns5:endPosition indeterminatePosition="unknown" />
              <!-- AIXM Attribute -->
      ...
      </ns10:timeSlice>
    </ns10:Event>
  </ns13:hasMember>
</ns13:AIXMBasicMessage>
```

Listing 4.1: Applying the object-property model to an AIXMBasicMessage.

Listing 4.1 shows an excerpt of a NOTAM which is enriched by comments. These comments indicate which XML element represents an AIXM feature and which an AIXM attribute. The decision whether an XML element represents an AIXM feature or not, is determined by the object-property model. AIXM attributes can represent relationships which refer to other AIXM features or contain text. As shown on line five of Listing 4.1, the XML element `<ns13:hasMember>` represents an AIXM attribute which contains other AIXM features. Whereas the line thirteen represents an AIXM attribute which contains text content only.

AIXM feature and attribute identification: Based on the object-property model two approaches can be followed to identify whether an XML element represents an AIXM feature or an AIXM attribute.

Hierarchical approach: One option for identification is to consider the hierarchy of the XML document. Due to the object-property model AIXM features can only occur every second hierarchy level in the XML document since an AIXM attribute has to be located between them. This applies for the AIXM attributes too, since no AIXM attribute can have other AIXM attributes as direct children. There is always an AIXM feature between two AIXM attributes. This limits the occurrence of AIXM attributes to every second hierarchy level in the XML document. This approach allows to easily identify the type of an XML element due to the alternating occurrence of AIXM features and AIXM attributes. The only prerequisite is that it must be ensured that the first XML element represent an AIXM feature.

Naming approach: Another approach for the determination is to distinguish between UML classes representing AIXM features and UML associations representing AIXM attributes. UML classes are labeled using a name starting with an uppercase letter. Whereas UML associations as well as UML class attributes are labeled using names with a lowercase starting letter. This approach works fine as long as all UML classes and associations follow this naming convention. However there are exceptions such as the usage of acronyms. An example is the association "airpointReferencePoint" which associates GML information to an AIXM airport feature. Instead of using the original name, the XSD schema specifies the association using the acronym "ARP" for the AIXM attribute. Since this acronym starts with an upper case letter, it would be detected as an AIXM feature and not as an AIXM attribute. Therefore this approach requires additional exception handling in order to incorporate such naming exceptions.

Since the hierarchical approach works without an exception handling, it is used to map the AIXM document to the OL representation. The hierarchical approach requires no accessing and checking of the XML element name instead the hierarchy can be determined by a counter which indicates the depth. Assuming that the first XML element on level one represents an AIXM feature, the alternating occurrence of AIXM features and attributes can be determined by simply using a modulo-2-operation. The operation returns a remainder when the XML element on the current level represents an AIXM feature otherwise an AIXM attribute. This approach can be applied to the query and configuration data sources as well, since both follow the object-property model.

After the determination whether an XML element represents an AIXM feature or an AIXM attribute is performed, the XML elements need to be mapped to the corresponding OL representation. AIXM features are mapped to OL objects. OL objects are created using instance level statements to instantiate OL classes. These OL classes can be obtained by mapping the source XSD documents to schema level statements. However, this mapping of the XSD document is not necessary since valid OL programs do not need to have existing schema level statements (Section 2.4). The schema level statements are not needed because the reasoner validates the OL program syntactically, but not the schema/instance correspondence. Therefore, instead of mapping the XSD and the XML document, only the XML document is mapped to the corresponding OL representation. Therefore only the mapping of AIXM features to OL objects and AIXM attributes to OL attributes needs to be defined.

OL Objects: As shown in Section 3.1, the OL objects are defined using f-molecules. Each OL object is identified by an id-term which is represented by the `id` XML attribute of an XML feature element. However, XML feature elements can have multiple XML attributes besides the `id` attribute. These XML attributes are included within the f-molecule by using data-F-atoms.

```
<ns13:AIXMBasicMessage ns5:id="FNS_ID_34828025">
<!-- AIXM Feature -->
  <ns13:hasMember>
  <!-- AIXM Attribute -->
    <ns10:Event ns5:id="Event_1_34828025">
    <!-- AIXM Feature -->
```

Listing 4.2: AIXMBasicMessage cutout.

Listing 4.2 shows an excerpt of an `AIXMBasicMessage` where the first XML feature element is represented by the `AIXMBasicMessage` XML element. The `AIXMBasicMessage` XML element does not represent the id-term of the resulting OL object, it represents the corresponding OL class. Furthermore the XML element includes an id-attribute. As mentioned before, the id-term is determined by the `id` attribute. Based on this information the OL object with the id-term `FNS_ID_34828025` instantiating the `AIXMBasicMessage` class can be created as depicted in Listing 4.3.

```
FNS_ID_34828025:ns13AIXMBasicMessage[ns5id -> "FNS_ID_34828025"].
```

Listing 4.3: Example of creating an OL object with a f-molecule.

Besides using the `id` attribute as the id-term it is added as a data-F-atom within the f-molecule as well. Following these steps leads to an OL object which is named, instantiated and its XML element attributes are assigned to it as OL attributes. However, this does not capture the association to other XML feature elements. To capture these associations, the XML elements in the next hierarchy level must be comprised.

OL Attributes: the second XML element in Listing 4.2, represents an AIXM attribute respectively association. This AIXM attribute is mapped to an OL attribute. The `<hasMember>` XML element does not contain any information except that it determines that all child XML feature elements within this XML element are associated OL objects of the upper OL object `FNS_ID_34828025`. Since these child XML feature elements are again OL objects, new f-molecules for each of them are created.

```
FNS_ID_34828025:ns13AIXMBasicMessage[ns5id -> "FNS_ID_34828025",
    ns13hasMember->Event_1_34828025].
```

Listing 4.4: Adding OL attributes with their corresponding OL attribute value to the upper OL object.

> The id-term of the child OL object is used as the value which is added as the OL attribute value to the f-molecule of the parent OL object `FNS_ID_34828025`. As depicted in Listing 4.4, the OL attribute `ns13hasMember` with the OL attribute value `Event_1_34828025` is added to the f-molecule of the upper OL object `FNS_ID_34828025`. After the addition of the OL attributes and associations, the first f-molecule for OL object `FNS_ID_34828025` is finished.

The next step is to map the XML feature element `Event_1_34828025` with all its XML elements and associations to the corresponding OL representation. Therefore these steps is recursively repeated until all XML elements are mapped to their corresponding OL representation.

4.2 Mapping Approach

In this section the basic mapping concept is extended by considering exceptions which can occur and by considering the processing tasks which have to be performed while mapping the data sources. The origins of them can be found in the properties of the XML or OL structure, restrictions, and requirements which need to be fulfilled. The exceptions must be handled during the mapping process which is why they are analyzed. Moreover, the processing tasks are examined and detailed.

4.2.1 Namespaces

As stated in Section 2.4, OL statements can be qualified by namespaces in order to identify OL classes, objects or attributes. Namespaces are separated from the local name by the "#"-sign as depicted in Listing 2.7. The semantics of namespace-qualified OL classes, objects or attributes is always a pair of strings, representing an Uniform Resource Identifier (URI) (`httm://www.aixm.aero/schema/5.1#`) or a prefix (`aero`) and a local name (`Aircraft`). These namespaces must be available in the OL representation because otherwise the OL statements would not be valid since the used prefixes in the f-molecules could not be resolved by the reasoner. Therefore each new namespace and the corresponding prefix are collected during the mapping of the XML documents which fulfills Requirement 3.4.

```
<ns13:AIXMBasicMessage ns5:id="FNS_ID_34828025" xmlns:ns1="http://www.
    opengis.net/ows/1.1" xmlns:ns2="http://www.aixm.aero/schema/5.1/
    message">
```

Listing 4.5: NOTAM containing a namespace declaration as an attribute.

Inside an XML document, a namespace is defined by using the XML attribute **`xmlns`** as depicted in Listing 4.5. This XML attribute can be used in all elements of the XML document when a new namespace needs to be introduced. However, the usage of the **`xmlns`** attribute without a prefix only defines the default namespace. The default namespace is valid for the current XML element and all child elements. If more than one namespace is used in an XML document, then the **`xmlns`** attribute must be extended by a colon followed by a prefix. This allows to define and use multiple namespaces inside one XML document. Therefore while mapping the XML documents it needs to be checked whether a namespace is defined using the **`xmlns`** attribute. The new namespace is not added as an OL attribute to the f-molecule, instead the value is added as a namespace statement.

```
:- prefix ns1 = "http://www.opengis.net/ows/1.1".
:- prefix ns2 = "http://www.aixm.aero/schema/5.1/message".
FNS_ID_34828025:ns13#AIXMBasicMessage[ns5#id->"FNS_ID_34828025", ns13#
    hasMember->Event_1_34828025].
```

Listing 4.6: Example of a NOTAM including mapped text.

Listing 4.6 depicts the resulting OL statements after mapping the XML element of Listing 4.5. The XML element contains two **`xmlns`** attributes which are mapped to the declaration of two namespace statements consisting of a prefix referring to an URI and an f-molecule which contains the OL object without the **`xmlns`** attributes. Moreover, the prefixes are separated from the OL class and attribute names by using the "#"-sign which conforms to the OL syntax. This is done because the given OL sample representation in Section 3.1.1 merged names and prefixes to one string. The merging would lead to valid OL statements but the namespaces would not be used since the OL processor could not differ a prefix from the OL terms such as `ns13AIXMBasicMessage`.

4.2.2 Text

Text in NOTAMs can occur over multiple lines and at different positions. Furthermore it can contain multiple blanks which are used to format the text output. However f-molecules can only contain text without line breaks. Therefore the line breaks "\n" and "\r\n" are replaced by their HTML equivalent `<br>`. The HTML tag `<br>` is used because it leads to a text represented in a single line. Furthermore the separator information of a new line is not lost. Besides that multiple occurrences of blanks are replaced by one blank because this kind of formatting is irrelevant in one line. It only stretches the OL statement and thereby decreases the readability. In the OL statements the value of an OL attribute can be defined using a literal. This literal is wrapped within double quotation marks. It is possible that the NOTAM text already contains quotation marks around terms and therefore these are handled by escaping them in the OL attribute value using a backslash.

```
<ns10:NOTAM ns5:id="NOTAM_1_34828025">
  ...
  <ns10:text>
    QXXXX WASHINGTON DULLES INTERNATIONAL AIRPORT AIRSPACE ADS-B
    SERVICES TISB AND FISB AVBL FEBRUARY 29, 2012. REPORTS OF TIS-B AND
         FIS-B MALFUNCTIONS SHOULD BE REPORTED BY
        RADIO OR
        TELEPHONE
    TO THE NEAREST "FSS" FACILITY.
  </ns10:text>
  ...
</ns10:NOTAM>
```

Listing 4.7: Example of a NOTAM text content.

Listing 4.7 depicts an excerpt of a NOTAM representing the AIXM feature `<ns10:NOTAM>` and the AIXM attribute `<ns10:text>`. The AIXM attribute `<ns10:text>` contains the text which includes quotation marks and is formatted using blanks.

```
NOTAM_1_34828025:ns10#NOTAM[ns5#id -> "NOTAM_1_34828025", ns10#text ->
    "QXXXX WASHINGTON DULLES INTERNATIONAL AIRPORT AIRSPACE ADS-B <br>
    SERVICES TISB AND FISB AVBL FEBRUARY 29, 2012. REPORTS OF TIS-B AND
     FIS-B MALFUNCTIONS SHOULD BE REPORTED BY <br> RADIO OR <br>
    TELEPHONE <br> TO THE NEAREST \"FSS\" FACILITY."].
```

Listing 4.8: Example of a NOTAM including mapped text.

Listing 4.8 depicts the resulting f-molecule based on the NOTAM excerpt of Listing 4.7. The text is parsed by replacing the line breaks with `<br>`, multiple blanks by one blank, and escaping quotation marks with backslashes.

```
<ns10:NOTAMTranslation ns5:id="NT02_34828025">
  <ns10:type>OTHER:ICAO</ns10:type>
    <ns10:formattedText>
      <html:div xmlns:html="http://www.w3.org/1999/xhtml"
      xmlns="http://www.aixm.aero/schema/5.1/message"
      xmlns:aixm="http://www.aixm.aero/schema/5.1" >
      <![CDATA[<P align=left>04/103 NOTAMR<BR><B>Q) </B>ZNY/QMXLC/IV/NBO
          /A/000/999/4038N07346W005<BR><B>A) </B>KJFK<BR><B>B) </B
          >1404070918<BR><B>C) </B>1505252359EST<BR><B>E) </B>TWY D BTN
          TWY C AND HANGAR 7 RAMP CLSD<BR></P>]]></html:div>
      </html:div>
    </ns10:formattedText>
    ...
</ns10:NOTAMTranslation>
```

Listing 4.9: Example of a NOTAM including formatted text.

Listing 4.9 shows that besides using blanks and line breaks, AIXM 5.1 provides the possibility to format text using HTML elements. This formatted text is used inside the AIXM attribute `<formattedText>`. The text is mapped as the OL attribute value of the OL attribute `formattedText` including the HTML tags. Preserving these HTML tags in the OL attribute value preserves the formatting. Furthermore the namespaces inside the HTML tags are kept and processed as described before. This approach avoids loss of information since the formatting is preserved (Requirement 3.1)

4.2.3 Attributes of AIXM Attributes

AIXM attributes can contain content or they can be used to encode associations between AIXM features. Beside that there can be AIXM attributes which contain additional information represented by various XML attributes. Since only the AIXM attributes are added as OL attributes to OL objects the attributes of the AIXM attribute would be lost. Due to the fact that no information may be lost in the mapping process, a solution needs to be found (Requirement 3.1).

```
<ns5:TimePeriod ns5:id="Event_TS_TP_1_34828025">
  <ns5:beginPosition>2013-02-12T21:51:00.000Z</ns5:beginPosition>
  <ns5:endPosition indeterminatePosition="unknown" xsi:nil="true" xmlns
      :xsi="http://www.w3.org/2001/XMLSchema-instance"/>
</ns5:TimePeriod>
```

Listing 4.10: AIXM attribute with multiple attributes.

Listing 4.10 depicts the two AIXM attributes `<ns5:beginPosition>` and `<ns5:endPosition>`. Both represent OL attributes but the second one contains additional XML attributes. A possible solution to map them to the corresponding OL representation is to introduce a new OL object for AIXM attributes. AIXM attributes are added as f-molecules including the XML attributes as OL attributes. The upper OL object stores a reference to the AIXM attribute by adding an OL attribute referring to it. In order to refer to the attribute OL object an identifier is needed. The generation of OL objects without an identifier is described in detail in Section 4.2.5.

```
Event_TS_TP_1_34828025:ns5#TimePeriod[ns#beginPosition -> "2013-02-12
    T21:51:00.000Z", ns5#endPosition ->
    uuid_b9024736_3a25_4c92_9b2f_bfc0e29ae31b].

uuid_b9024736_3a25_4c92_9b2f_bfc0e29ae31b:ns5#endPosition[
    indeterminatePosition -> "unknown", xsi#nil -> "true"].
```

Listing 4.11: Handling multiple XML attributes of AIXM attributes using OL attribute objects.

As shown in Listing 4.11, the f-molecule representing an OL object is created for the AIXM attribute `<ns5:endPosition>` including its XML attributes as OL attributes. The AIXM attribute `<ns5:beginPosition>` is mapped as already described in Section 4.1.

Another approach is to treat the AIXM attributes, which contain multiple XML attributes, as normal AIXM attributes. In order to prevent information loss by not mapping the XML attributes of the AIXM attribute, they are included as OL attributes with a parameter in the parent OL object. The parameter is named after the XML attribute name in the AIXM attribute. The XML attribute value is assigned to the OL attribute value.

```
Event_TS_TP_1_34828025:ns5#TimePeriod[ns#beginPosition -> "2013-02-12
    T21:51:00.000Z", ns5#endPosition(indeterminatePosition) -> "unknown
    ", ns5#endPosition(xsi#nil) -> "true"].
```

Listing 4.12: Handling multiple AIXM attribute attributes with OL attributes including parameters.

Listing 4.12 depicts the mapping of XML attributes of an AIXM attributes by using OL attributes. As stated in Section 2.4 these OL attributes are actually methods which can receive a parameter. This parameter is the name of the XML attribute and the value is the one of the XML element attributes. It is important to note that XML attributes of AIXM features are still added as OL attributes without parameters.

The second approach is chosen for the implementation since the first approach has drawbacks. The first approach provides a way to avoid loss of information, but it introduces overhead and conflicts with the basic idea where each AIXM attribute is added as an OL attribute. As depicted in Listing 4.11, an overhead is introduced by the newly generated f-molecule for the AIXM attribute. A reference to this f-molecule needs to be added in the upper OL object and therefore an identifier is created. Furthermore the readability of the generated OL program is negatively affected due to newly added lines containing f-molecules. In addition to that future work based on the OL statements will have to interpret OL objects differently since they can represent "real" OL objects or OL objects representing attributes. Moreover, a new OL attribute must be defined to store text content since the name of AIXM attribute is already used as the OL class term.

4.2.4 Processing Timestamp

As defined in Requirement 3.5 the data sources need to be enriched by additional information. This additional information includes a detection time stamp which records the time when the NOTAM has been mapped. The detection time stamp is used to find newly added or updated NOTAMs. To establish this, a new XML attribute `detectionTimestamp` is added to the AIXM feature before it is mapped. During the processing this new XML attribute of the AIXM feature is mapped and added as an OL attribute to the OL object as depicted in Listing 4.13.

```
FNS_ID_3381098:ns13#AIXMBasicMessage[ns5#id->"FNS_ID_3381098", ns5#
    boundedBy(xsi#nil)->"true", ns13#hasMember->{Airport_1_3381098,
    Event_1_3381098}, detectionTimestamp->"2015-07-02T10:44:34.483Z"].
```

Listing 4.13: F-molecule of a NOTAM including a detection timestamp.

However it is more efficient to directly add the time stamp as an OL attribute value to the f-molecule of the NOTAM instead of manipulating the XML document and mapping it. The detection time stamp is added in the following format "yyyy-MM-dd'T'HH:mm:ss.SSS'Z'" which corresponds to the date format of AIXM in order to ease later computational steps.

4.2.5 XML Elements with missing Identifiers

XML elements which are detected as OL objects need to have an identifier. As described in Section 4.1 this identifier is used as the id-term of the resulting f-molecule. XML elements which represent AIXM features usually contain an identifier. However there are cases where the identifier is not available (Requirement 4.2). For example when preprocessing steps add new XML elements or create new OL objects they need to set an id-term. Therefore an identifier needs to be generated. As stated in Requirement 4.2 the identifier must be unique and universal.

As mentioned in Section 2.3.5 AIXM relies on UUID as feature identification. Therefore an UUID generator is used to create missing identifiers. The RFC 4122 defines the structure of an UUID (Leach et al., 2005). The intention of UUID is to provide uniqueness across space and time. It is 128 bits long and requires no central registration process. An automated generation on demand can be done based on the fact that no centralized authority is required to administer them. UUIDs are represented in their canonical form by a sequence of thirty-two hexadecimal digits. As depicted in Listing 4.14 these digits are grouped into a sequence of eight, four, four, four and twelve digits.

```
f81d4fae-7dec-11d0-a765-00a0c91e6bf6
```

Listing 4.14: Example of an UUID.

The formal definition is based on (Crocker & Overell, 2005) and follows the Augmented Backus-Naur Form (ABNF).

```
UUID =  time-low "-" time-mid "-" time-high-and-version "-"
        clock-seq-and-reserved clock-seq-low "-" node

time-low = 4hexOctet
time-mid = 2hexOctet
time-high-and-version = 2hexOctet
clock-seq-and-reserved = hexOctet
clock-seq-low = hexOctet
node = 6hexOctet
hexOctet = hexDigit hexDigit
hexDigit =
"0" / "1" / "2" / "3" / "4" / "5" / "6" / "7" / "8" / "9" /
"a" / "b" / "c" / "d" / "e" / "f" /
"A" / "B" / "C" / "D" / "E" / "F"
```

Listing 4.15: Formal UUID string representation according to (Leach et al., 2005).

There are five different implementations of UUID generators, where each of them implements a different algorithm.

Version 1: As stated before the UUID consists of 128 bits resulting in sixteen octets (Leach et al., 2005). Version 1 represents the basic implementation and is based on the Media-Access-Control (MAC) address and a time stamp. A MAC address is used to identify a network device and represents the uniqueness across space. Version 1 uses a sixty-bit time stamp, which represents the Coordinated Universal Time (UTC) counting the 100-nanosecond intervals since 00:00:00.00, 15 October 1582. The time stamp is used to represent the uniqueness across time. As depicted in Listing 4.15 it is split into octets where the first four octets represent the low field of the time stamp (`time-low`). The following two octets represent the middle field of the time stamp (`time-mid`) and the last two octets represent the high field of the time stamp multiplexed with the version number (`time-hi-and-version`). The version number determines the implementation variant of the UUID. It is located in the most significant 4 bits of the time stamp (bits 4 to 7 of the `time_hi_and_version` field). The next two octets are based on the clock sequences which are used to avoid duplicates that could occur when the clock is set backwards in time. A random number is used to initialize the clock sequence in order to minimize the correlation across systems. The clock sequence is split into two octets. The first octet includes the high field of the clock sequence multiplexed with the variant (`clock_seq_and_reserved`). The second octet includes the low field of the clock sequence (`clock_seq_low`). The last six octets are used to store the host address respectively the MAC address of the network card. Systems without a MAC address will use a randomly generated value.

The chance of a collision in generated UUID in Verison 1 does not actually exist, due to the fact that it represents a single point in space (computed using the MAC) and time (the number of intervals). Even if two UUIDs are generated in the same interval of nanoseconds the time stamp will be incremented by one. This prevents collisions in a reliable way. However, it was shown that collisions can occur. One use case which causes collisions is when the MAC is spoofed and thereby possibly not unique any more. Another way to cause collisions is if UUIDs are generated on a device without a MAC address due to a missing network card. Another drawback, besides the chance of collisions, of this version is that it reveals the time when it is generated and the identity of the computer. Therefore other versions were developed.

Version 2: The Version 2 is also known as the Distributed Computing Environment (DCE) Security version (Leach et al., 2005). It is similar to Version 1 except that the first four octets of the time stamp are replaced by the Portable Operating System Interface (POSIX) User Identifier (UID) of the user (The Open Group, 1997). Furthermore the last octet of the clock sequence is replaced by a POSIX UID domain identifier. Due to this reduced clock size it is going to produce collisions sooner than Version 1.

Version 3 and 5: Both versions are based on generating UUIDs from names using hash functions (Leach et al., 2005). These names should be unique such as textual names or namespaces. Therefore Uniform Resource Locators (URLs) or domain names are mostly used. The generation algorithm has to generate always the same UUID based on a given name in a namespace. Furthermore two UUIDs generated from two different names in the same or different namespace should be different. Version 3 uses the Message Digest 5 (MD5) Algorithm defined in RFC 1321 (Rivest, 1992). The MD5 algorithm generates a 128 bit hash output based on the given input which is truncated to the 122 bits available in the UUID since six bits are reserved for fixed values. The use of MD5 in one system will probably lead to no collision but on a global scale it can lead to collision as shown in

(Selinger, 2006) and (Stevens, 2006). (Stevens, 2007) has shown, that collisions can occur after 2^{33} invocations. The structure of Version 3 differs because for the construction of the time stamp the MD5 hash of the given name is used instead of the interval of the nanoseconds. Also the clock sequence and the node are generated from the given name. Version 5 is similar to Version 3 except that for the generation the algorithm Secure Hash Algorithm (SHA-1) is used (U.S. Department of Commerce & National Institute of Standards and Technology, 2012). The SHA-1 provides a 160 bit hash which is also truncated to 122 bit. As shown by (Rijmen & Oswald, 2005) a collision by obtaining SHA-1 hashes from small messages can occur after 2^{63} invocations. Generic algorithms can obtain a collision after 2^{80} invocations on average.

Version 4: UUID Version 4 relies on a random number (Leach et al., 2005). The first six bits are reserved, where the first four of them represent the version number. The remaining 122 bits are set using a random number. That means that the sixty-bit value of the time stamp, the fourteen-bit value of the clock and the forty eight-bit value of the node are randomly generated.

As shown in (Suzuki, Tonien, Kurosawa, & Toyota, 2006), the birthday paradox can be used to calculate the chance of a collision of two Version 4 UUIDs. The birthday paradox shows that in contrast to the human intuition a low number of randomly assigned people is needed to find a pair with the same birthday date. This paradox can also be applied to the probability of collisions within a randomly generated number based on 122 bits. The chance is approximated using the formula $p(n) \approx 1 - e^{-\frac{n^2}{2x}}$ where the number of already generated UUIDs relates to 2^n and x is the number of bits occupied by the random number. The Table 4.1 shows the resulting probabilities which result by applying the formula on the UUID generation.

After creating 1,12 quadrillion (2^{50}) UUIDs the probability of causing a collision while generating the next one is 1,1921 * 10^{-7}.

Based on these insights of the different versions and their uniqueness a decision can be made. Version 1 can provide reliable UUIDs regarding uniqueness if the issues regarding the MAC spoofing and machines with no MAC address can be handled. However as long as this cannot be ensured, Version 1 and Version 2 cannot be used. Version 3 and Version 5 are both based on hash functions which use names and namespaces in order to generate a hash. As mentioned before, collisions can occur after 2^{33}

Table 4.1: Approximated probability of a collision after 2^n UUID calculations

Calculates UUIDs (n)	**Collision Probability**
$2^{40} = 1099511627776$	$1{,}1369 * 10^{-13}$
$2^{45} = 35184372088832$	$1{,}1641 * 10^{-10}$
$2^{50} = 1125899906842624$	$1{,}1921 * 10^{-7}$
$2^{55} = 36028797018963968$	10^{-4}
$2^{60} = 1152921504606846976$	0,1175
$2^{65} = 36893488147419103232$	0,9999

and 2^{63} invocations. However, both are based on names and namespaces which again need to be distinguishable in order to create unique hashes. Therefore Version 4, which relies completely on random numbers, is preferred.

As shown in Listing 4.16 the AIXM UUID identifier consists of a prefix `uuid` which is append to the value of the `id` attribute. The Listing 4.16 shows that the prefix `uuid` is set at the beginning of the identifiers' value.

```xml
<aixm:SpecialNavigationStation gml:id="uuid.b9024736-3a25-4c92-9b2f-
    bfc0e29ae31b">
  <gml:identifier codeSpace="urn:uuid:">
    b9024736-3a25-4c92-9b2f-bfc0e29ae31b
  </gml:identifier>
  <aixm:featureMetadata>
<gmd:MD_Metadata>
```

Listing 4.16: UUID of an AIXM feature.

This format is kept in order to be consistent with other elements. In OL id-terms have to start with a character to be valid. Appending the `uuid` prefix prevents id-terms containing UUIDs to be invalid since UUIDs can be generated starting with a number. Besides that, the hyphens in the UUID need to be replaced with underscores. In OL the hyphens are reserved since they are a part of the assignment "->" of OL attribute values to OL attributes. Moreover, the "." character is reserved in OL to declare the end of a statement. Therefore it is also replaced by an underscore as shown in Listing 4.17.

```
uuid.b9024736-3a25-4c92-9b2f-bfc0e29ae31b // invalid OL id-term
uuid_b9024736_3a25_4c92_9b2f_bfc0e29ae31b // valid OL id-term
```

Listing 4.17: Invalid and valid OL id-terms.

The Listing 4.18 depicts an excerpt of an AIXM feature which is recognized as an OL object element without an existing identifier.

```
<aixm:SpecialNavigationStation>
  <aixm:featureMetadata>
    <gmd:MD_Metadata>
```

Listing 4.18: AIXM feature without an identifier.

Therefore an identifier is generated using a UUID Version 4 generator. This UUID is manipulated by replacing all hyphens with underscores and adding the `uuid_` prefix in front of the UUID value. After this, the identifier is added to the AIXM feature as an XML attribute. This XML attribute is then mapped according to Section 4.2.3.

```
uuid_b9024736_3a25_4c92_9b2f_bfc0e29ae31b:aixm#SpecialNavigationStation
    [id->" uuid_b9024736_3a25_4c92_9b2f_bfc0e29ae31b", aixm#
    featureMetadata -> uuid_7ee612af_15a5_4468_807d_3c8ae33649c3].
uuid_7ee612af_15a5_4468_807d_3c8ae33649c3:gmd#MD_Metadata[id->"
    uuid_7ee612af_15a5_4468_807d_3c8ae33649c3"].
```

Listing 4.19: F-molecule including newly generated identifier for the AIXM features.

Listing 4.19 shows the resulting f-molecule including the newly generated identifiers. For both AIXM features `SpecialNavigationStation` and for the sub AIXM feature `MD_Metadata` an identifier is generated.

4.2.6 Link Resolution

As stated in Requirement 3.2 local references inside an XML document must be resolved. Therefore the identifier of the referred XML element has to be used instead of the XLink reference statement. Section 2.3.5 describes the referencing capabilities of AIXM. It is stated that AIXM provides internal and external references by using XLink. Internal references refer to existing XML elements inside the XML document.

As shown in Listing 4.20 line twelve, the XML element with the identifier `Event_1_3246418` is being referred. Therefore the locator attribute `href` is used which allows an XLink application to find a remote or locale resource. However, the locator attribute `xlink:href` can only identify a resource which represents an XML document. To refer to a specific XML element the XML Pointer Language (xPointer) is used. An XPointer allows to link to a specific fragment of an XML document by using XML Path Language (xPath) expressions. Therefore the "#"-sign, which indicates that the local XML document represents the resource, followed by the target identifier is used.

```
 1 <ns13:AIXMBasicMessage ns5:id="FNS_ID_32464181">
 2   ...
 3   <ns13:hasMember>
 4     <ns6:AirportHeliport ns5:id="Airport_1_3246418">
 5       <ns5:boundedBy xmlns:xsi="http://www.w3.org/2001/XMLSchema-
           instance" xsi:nil="true"/>
 6       <ns6:timeSlice>
 7         <ns6:AirportHeliportTimeSlice ns5:id="Airport_TS_1_3246418">
 8         ...
 9           <ns6:extension>
10             <ns10:AirportHeliportExtension ns5:id="AHE_EVENT_1_3246418">
11               <!-- Reference to other AIXM XML feature -->
12               <ns10:theEvent xmlns:ns2="http://www.w3.org/1999/xlink"
                   ns2:href="#Event_1_3246418"/>
13             </ns10:AirportHeliportExtension>
14           </ns6:extension>
15         </ns6:AirportHeliportTimeSlice>
16       </ns6:timeSlice>
17     </ns6:AirportHeliport>
18   </ns13:hasMember>
19   <ns13:hasMember>
20     <ns10:Event ns5:id="Event_1_3246418" xmlns:ns10="http://www.aixm.
         aero/schema/5.1/event">
21       ...
22     </ns10:Event>
23   </ns13:hasMember>
24 </ns13:AIXMBasicMessage>
```

Listing 4.20: Local reference within an XML document and the XLink target.

The reference in Listing 4.20 is mapped according to Section 4.2.3 since it is detected as an XML attribute of an XML element representing an OL attribute. The mapping output of this is depicted in Listing 4.21.

```
AHE_EVENT_1_3246418:ns10#AirportHeliportExtension[ns5#id="
    AHE_EVENT_1_3246418", ns10#theEvent(ns2#href)->"#Event_1_34058787
    "].
```

Listing 4.21: Mapping of references as XML attributes of AIXM attributes.

However, the reference value is not resolved since it is still treated as a literal instead as an id-term. The mapping described in Section 4.2.3 needs to be adapted for links. Instead of referring to the OL object identifier literal, the OL attribute refers directly to the OL object. In order to distinguish local from external references the parameter `ns2#href` respectively `xlink#href` is replaced by `ref` (Listing 4.22). This makes it easier to detect local references in the OL representation.

```
AHE_EVENT_1_3246418:ns10#AirportHeliportExtension[ns5#id="
    AHE_EVENT_1_3246418", ns10#theEvent(ref)->Event_1_34058787].
```

Listing 4.22: Mapping of resolved references.

Beside local references, AIXM supports concrete external references where information about referred AIXM features is available via web services. Again XLink in combination with XPointer is used to refer to these resources. Instead of providing just the identifier of the referred XML element, an URL is additionally used. The URL will be resolved to an XML document which contains the XML element which is referred using a "#" and the identifier (Listing 4.23).

```
<aixm:clientAirspace ns5:id="CA_1_3246418" xlink:href="http://aim.faa.
    gov/services/AirspaceService#uuid.a82b3fc9-4aa4-4e67-8def-
    aaea1ac595j"/>

<aixm:clientAirspace ns5:id="CA_1_3246418"
xlink:href="http://aim.faa.gov/services/AirspaceService?get=a82b3fc9-4
    aa4-4e67-8defaaea1ac595j#xmlns(ns1=http://www.opengis.net/gml/3.2)
    xmlns(ns2=http://www.aixm.aero/schema/5.1)xpointer(//ns2:Airspace[
    ns1:identifier='a82b3fc9-4aa4-4e67-8def-aaea1ac595j'])"/>
```

Listing 4.23: Simple and complex remote XLink statement with an XPointer reference.

Both references in Listing 4.23 refer to the same XML element. The first reference uses a simple XLink syntax whereas the second one defines the reference in a more detailed way. However since both are external references they are not resolved during the mapping. Both are mapped according to Section 4.2.3 since they are detected as XML attributes of an XML element representing an OL attribute.

4.2.7 Bounding Boxes

Requirement 4.1 defines that the missing bounding box for NOTAMs has to be calculated and added to the NOTAM and its features. A bounding box represents the minimum rectangle which is oriented to the x- and y-axes in order to enclose a geographic feature of a geographic dataset (Caldwell, 2005). It is also known as Minimum Bounding Rectangle (MBR) or envelope. As depicted in Figure 4.1, the bounding box is specified by two geographic coordinates which define the minimum x- and y-value and the maximum x- and y-value. The coordinates are determined by two longitudes and two latitudes representing the western-most, eastern-most, northern-most and southern most limits.

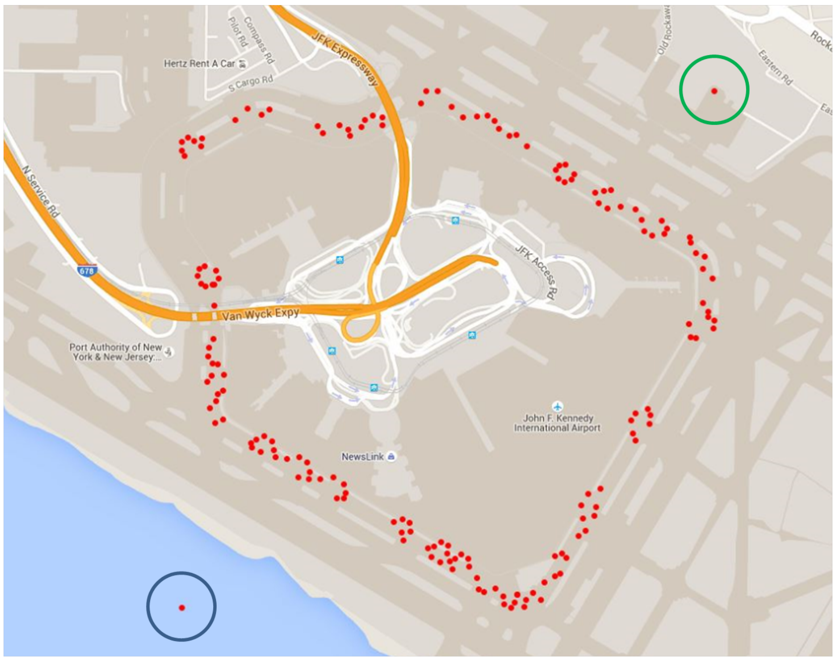

Figure 4.1: Geographic coordinates of the bounding box of the JFK Airport.

There exist several other bounding container shapes beside bounding boxes such as bounding parallelograms, circles, balls or ellipses (Caldwell, 2005). The bounding box is the simplest bounding container and requires the least computational effort which is why it is mostly used. Due to the low computational effort it is used in many geometry applications for collision avoidance, ray tracing or hidden object detection. Moreover it is used for spatial index schemes which use them to subdivide space. In the SemNOTAM system it is used to detect overlaps in order to find relevant NOTAMs for a given GML shape.

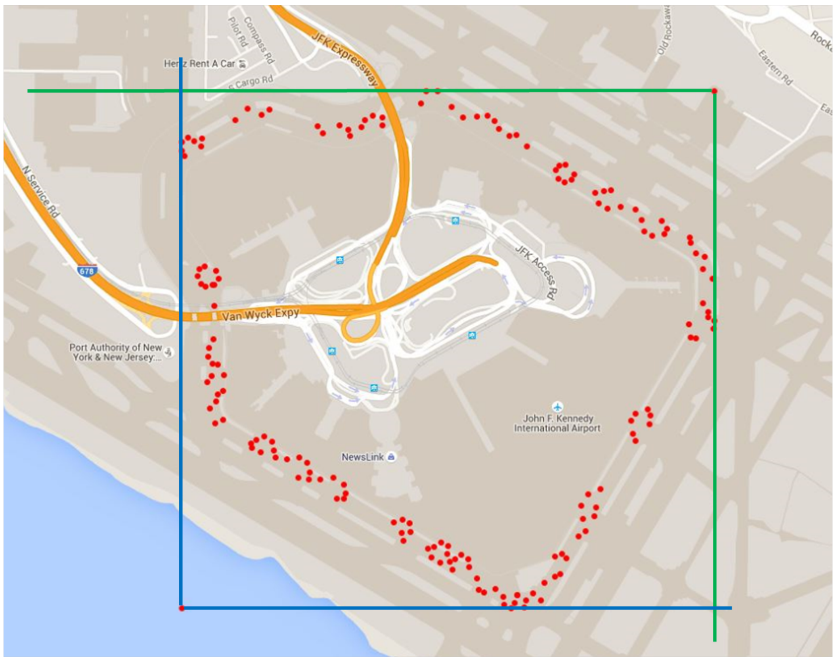

Figure 4.2: Visualization of a GML envelope.

In GML the bounding box is encoded as the `Envelope` XML element (Cox, Daisey, Lake, Portele, & Whiteside, 2005, p. 59). As depicted in Figure 4.2 an envelope determines the area "*by using a pair of positions defining opposite corners in arbitrary dimensions*". The north-east coordinate determines the `upperCorner` and the south-west coordinate determines the `lowerCorner` of the GML envelope (Listing 4.24). Moreover, the XML element is associated to an XML element which represents an AIXM feature by the AIXM attribute `<gml:boundedBy>`. As stated in Requirement 4.1 the bounding box respectively the GML envelope has to be calculated for the NOTAM as well as the AIXM features within it. This has to be done when no bounding box exists but a geographic data is available.

```xml
<gml:boundedBy>
  <gml:Envelope srsName="urn:ogc:def:crs:EPSG::4326">
    <gml:lowerCorner>-32.0886111111111 -47.0</gml:lowerCorner>
    <gml:upperCorner>57.690815969999996 52.4283333333333</gml:
        upperCorner>
  </gml:Envelope>
</gml:boundedBy>
```

Listing 4.24: Example of an XML GML envelope.

Since bounding boxes are used to enclose geographical datasets, they have to capture two use cases. The first one is a dataset containing multiple geographic points which can represent various GML shapes. The second use case is a dataset with only one geographic point in it. The bounding box depicted in Figure 4.2 represents a dataset containing multiple geographic points representing a so called linear ring. It depicts the path of a runway of the John F. Kennedy International Airport. In order to calculate the bounding box the western-most, eastern-most, northern-most and southern-most points of the dataset need to be determined. Therefore the coordinates of the points need to be compared. The northern-most point is characterized by the maximum latitude value whereas the southern-most point is determined by the minimum latitude. Similar to this the western-most point is determined by the point with the minimal longitude value whereas the eastern-most is determined by the maximal longitude value of a point.

However, the bounding box calculation for one geographic point in a dataset is more complex. To calculate a bounding box for a single point, additional information such as the radius of the bounding box is needed. As depicted in Figure 4.3 the radius is used to determine the distance from the given GML point to the points which are needed to calculated the GML envelope. The center point is the given GML point for an AIXM feature such as the airport reference point of an airport. It represents the center of the circle which is used to determine the western-most, eastern-most, northern-most and southern-most points. These points can be determined by using the Harversine Formula which assumes that the earths' surface can be represented by a sphere (United States Census Bureau, 1997). Specifying the azimuth (angle) and distance (radius) from a given point, a new one can be calculated. Another algorithm for calculating points is represented in (Vincenty, 1975). In contrast ot the Harversine Formula this algorithm assumes that the earths' surface is an ellipsoid. Since the earths shape is more similar to an ellipsoid than to a sphere the algorithm of Vincenty (1975) leads to more accurate results.

After the points are calculated they can be used to determine the

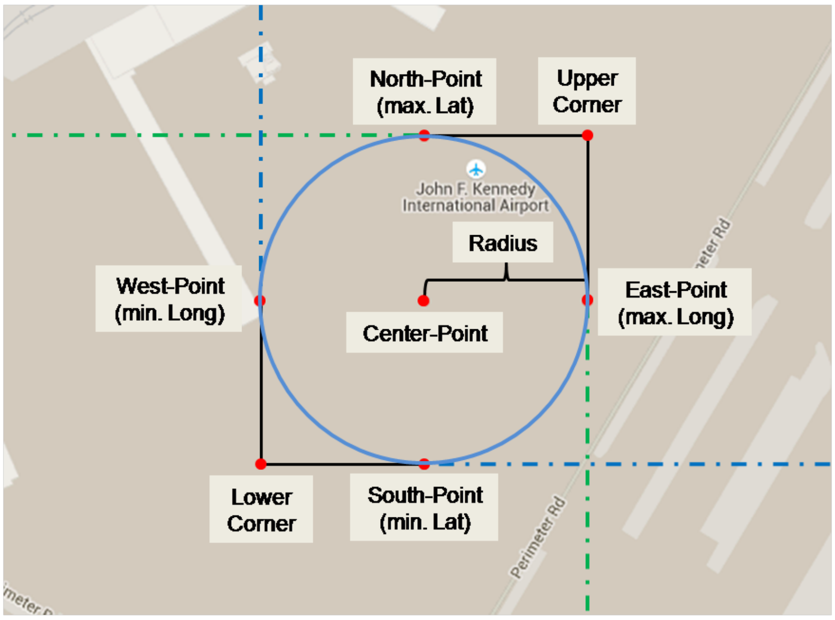

Figure 4.3: Bounding box calculation for one geographic point.

upper and lower corner of the bounding box as depicted in Figure 4.3. The latitude of the north-point and the longitude of the east-point determine the upper corner. The latitude of the south-point and the longitude of the west-point determine the lower corner. The calculated corners are then added in the XML document as a GML envelope as depicted in Listing 4.24. Since this is a preprocessing step it is added to the XML document instead of adding it directly to the mapped f-molecule.

As mentioned before, a NOTAM can contain multiple features which can contain GML points and therefore may require a bounding box calculation. Beside this, each NOTAM has its own bounding box. This bounding box of the NOTAM is the outer one which encloses all GML points included in the features' bounding boxes. For the calculation of the NOTAMs' bounding box all GML points of all features must be considered. However it is easier to access the already calculated bounding boxes and use the corner points for the calculation of the outer bounding box. It is not necessary to access all GML points again after the bounding boxes for the NOTAMs' features were calculated.

As stated in Section 3.3 the interpretation of GML information especially the coordinates depends on the given CRS. CRSs are required due to the distortions which occur during the plotting of the earths surface onto a 2D map. CRSs define how the earths surface can be mapped to latitude and longitude coordinates and how this mapping can be projected on a map (Frazier, 2013). As mentioned before, the earths' shape can be modeled using a sphere or an ellipsoid. An ellipsoid shape is preferred since the earth is slightly distorted at the equator. Datums are defined to anchor the abstract coordinates to the earth. They specify an origin point of the coordinate axes and their direction resulting in a 3D ellipse or sphere with latitude and longitude coordinates. Multiple projections such as a cone, plane, or cylindrical one can be used to represent the 3D globe to a 2D map representation. Figure 4.4 depicts a cylindrical projection of the earths surface. As shown the projection of the surfaces causes distortions. Especially the area at the top and the bottom of the projection is distorted. Angles and distances are preserved. Due to the occurring distortions other projections are used depending on which distortion should be minimized. As shown in Listing 4.24 the CRS is encoded in the attribute `srsName`. This `srsName` needs to be resolved to find the corresponding CRS. Due to the different projections the interpretations of the given coordinates depends on the CRS.

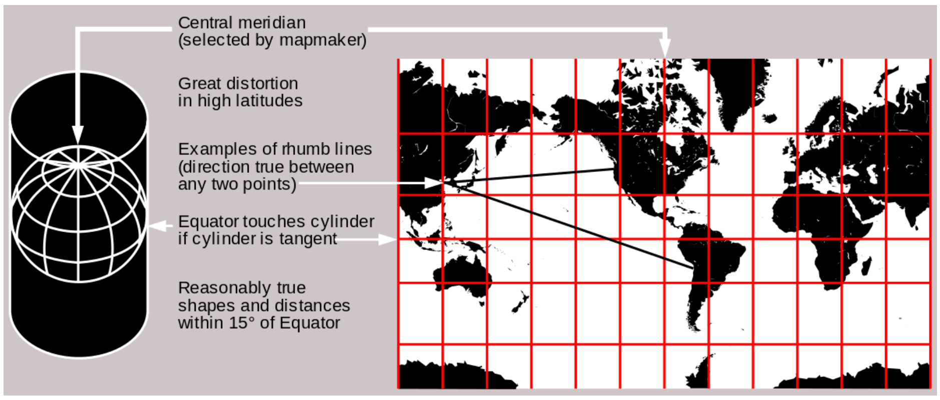

Figure 4.4: Cylindrical projection of the earths surface[1].

Since NOTAMs can include GML datasets of worldwide locations it is possible that different CRSs are used to encode the data. Therefore the datasets need to be transformed to a common CRS in oder to align them with each other. This transformation ensures that the units are interpreted the same way.

4.3 Mapping Technologies

The mapping of the structured data sources to the corresponding OL representation is the central task of the mapper. It is needed to fulfill the Requirement 2.1, Requirement 2.2, and Requirement 2.3. Therefore a suitable technology must be found. This section provides an overview of existing transformation technologies for mapping XML documents. Each technology is analyzed and a prototype of the mapper is implemented with it. The prototype includes all basic steps excluding the preprocessing step such as the GML handling. However, the preprocessing steps have to be considered while selecting the technology. Finally the implementations are compared and the best performing technology is chosen.

As mentioned before, the technologies represent transformation technologies for mapping XML documents. Compiling technologies are not evaluated since the mapping task differs from a compiling task. Compiling covers the transformation of a source code written in programming language into another programming or computer language (Aho & Ullman, 1972, p. 59 ff.). It is mostly used to transform source code from a high-level programming language (source language) to a lower level programming language (target language). The source code written in the source language represents a string of characters which is converted by a compiler to object code. The compilation task is split into two phases. The analysis phase is used to analyze the lexical structure in order to retrieve lexical units called tokens. Tokens can represent keywords, strings, operators or numbers. Moreover, the syntax analysis checks whether the source code corresponds to the predefined syntax respectively grammar of the source programming language. The semantic analysis checks whether the source code follows predefined rules. Examples for such rules can be that a variable must be declared before it is used or that only a value of the corresponding type must assigned to it. The result of the first phase is a decorated syntax tree. After the analysis phase is finished, the second phase starts. This generation phase uses the decorated syntax tree and creates programming code of the target language which is optimized according to data-flow and dependence analyses.

[1] `https://commons.wikimedia.org/wiki/File:Usgs_map_miller_cylindrical.PNG`

In contrast to that, the mapping task does not transform a source programming language into a target programming language. It transforms XML, which is a markup language for documents, into the knowledge-representation language OL. There is no programming source code which will mapped to a target programming language. However, the basic idea of compiling can be applied. In the analysis phase the bit sequence representing the XML document must be converted to a character sequence which can be used for lexical analysis (Lam, Ding, & Liu, 2008). The result of this are tokens such as start/end XML elements, attributes names and values, and text. The syntax and the semantic of the data sources are predefined by the XML syntax and their corresponding XSD documents. As mentioned in Section 2.3.1 the XML syntax determines the well-formedness and the XSD document determines the validity. The result of the first phase is a model representing a tree or events depending on the used XML technology. The model is used in the second phase in order to map it to OL statements representing OL objects and attributes. Since f-molecules are used instead of single F-atoms and data-F-atoms an optimization is performed. Besides that the source representation is enriched by additional information or handling missing information.

In order to map XML documents to OL statements the technologies must provide the possibility to access the XML document structure and define transformation rules. The accessing of the XML document corresponds to the analysis phase which is supported by various XML parsers. Numerous processing technologies which support the parsing of XML documents can be found. XSLT, XQuery, and directly using XML parsers such as SAX, StAX, and DOM are the most common ones. The following subsections will analyze each of them.

4.3.1 XSLT

XSLT is a part of the Extensible Styling Language (XSL) which was developed to perform complex styling operations on XML documents (Lam et al., 2008). In contrast to Cascading Style Sheets (CSS), which are used to format HTML, the XSL styling sheets use an XML notation in order to format XML documents. Every XSLT stylesheet is a valid XML document since it follows the XML notation. Beside the formating capabilities of XSL, XSLT allows to transform XML documents by applying XSLT stylesheets to other XML documents, HTML pages, plain text, or other representations

(W3C, 2007). XPath is included as a subset of XSLT and is primarily used to identify subsets of the XML document tree. Beside that it can be used to perform calculations. XPath also provides various functions which enrich the restricted capabilities of XSLT.

The data model which is used in XSLT is the XQuery and XPath data model (XDM) (IBM, 2010). XDM represents the content of an XML document as a tree structure. This tree structure is described using a sequence of items. An item is either an atomic value or a node. Atomic values are represented by the built-in atomic data types defined in the XML Schema such as strings, dates or integers. Nodes are characterized by properties which indicate its name, attributes, parent, children or other information. There are seven different kinds of nodes which are supported by XDM. Each kind indicates whether the node is a document, attribute, element, text node, processing instruction, or namespace. Moreover the XDM stores the node identity and the hierarchy of the document and therefore each node.

XSLT is based on pattern-matches and templates. The pattern-matches are determined by XPath expressions selecting specific elements of the XDM. A template contains rules which are triggered when a pattern-match is detected. The template defines the transformation and the output. Furthermore advanced processing steps can be executed. XSLT allows to define and read variables, however they cannot be altered after they are bound (De Schrijver, De Neve, Van Deursen, De Cock, & de Walle, 2006). Besides that it allows to define control flows providing choices or loops. These control flow elements again use XPath expressions. Moreover, templates can be reused and therefore used to modularize the XSLT stylesheet.

Figure 4.5 depicts the XSLT transformation process. One or more XML documents and one or more XSLT stylesheets can be used as input files. The XSLT processor starts with reading and preparing the XSLT stylesheets. After that the processor builds a source tree based on the given XML documents. The built source tree is processed by finding the best-matching template for a given XML node. When the template is identified it is instantiated and recursively applied to the XML node in the source tree. The result is either the creation of a node in the result tree or the processing of other nodes in the source tree. Finally, after the XSLT processor finished the processing of the source tree and the applying of XSLT templates the resulting result tree is output.

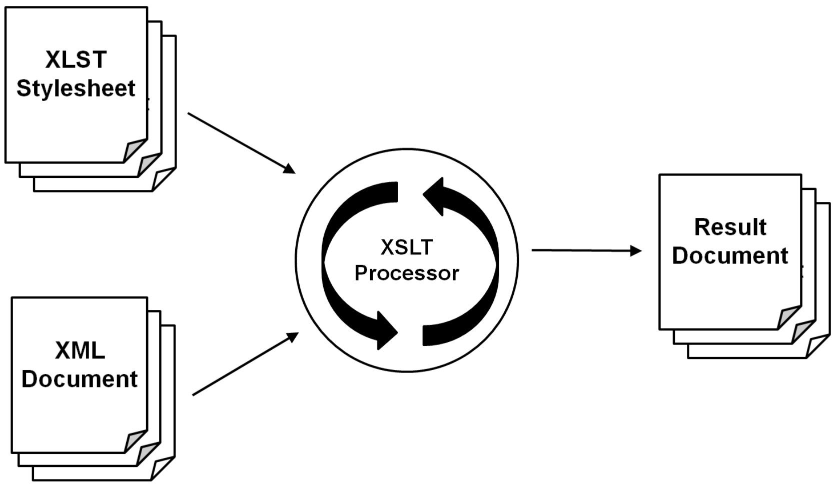

Figure 4.5: XSLT transformation process (W3C, 2007).

XSLT Prototype Implementation: XSLT allows to implement a prototype which covers main features of the mapping concept as depicted in Listing B.1. However there is a huge drawback of this solution. The recursive approach cannot be applied easily since the XSLT processor directly creates an output. The recursive approach would result in nested f-molecules. However, the recursion is needed to iterate through the XML tree representing the source data. The problem is that XML elements exist which have no identifier. For these XML elements an identifier according to Section 2.4 needs to be generated. As stated in Section 4.1 this identifier is used in the OL f-molecule of an XML feature element as an OL attribute value in order to store the associations to other OL objects. However this identifier needs to be passed recursively to sub XML elements which use this identifier as the id-term for the mapped OL f-molecule.

This issue can be overcome by changing the recursion to an end recursive approach. However, this cannot be easily established in XSLT stylesheets since variables cannot be reassigned to new values and no additional data model can be defined within the scope of XSLT. Moreover it is not trivial to return the output as the return value of a template without directly outputting it. Another solution would be to preprocess the whole XML input document and add missing identifiers before it is transformed with XSLT. However, other preprocessing steps such as the GML handling must be conducted before in order to prevent XML elements without identifiers.

This shows that the implementation is possible but it would result in complex and hardly maintainable XSLT stylesheets. In contrast to the XQuery and DOM approach the XSLT implementation would not allow to easily extend the existing implementation which contradicts the Requirement 3.3.

4.3.2 XQuery and BaseX

XQuery is a functional programming and query language used for querying and transforming data (W3C, 2014). The accessed data is usually represented as XML documents, but other data formats such as plain text, relational database, or JavaScript Object Notation (JSON) can be accessed using extensions. Similar to XSLT, XPath is a subset of XQuery. Both XSLT and XQuery are developed by the World Wide Web Consortium (W3C) which is also responsible for XPath. Moreover, XSLT and XQuery share the same data model XDM which is described in Section 4.3.1. XQuery allows to select documents and elements regarding their content, attributes, or structure. Moreover it allows to merge sequences of elements to new elements.

Since XQuery was initially developed to provide a query language for XML databases it is inspired by Structured Query Language (SQL) which is used to query relational databases (W3C, 2014). XQuery provides FLWOR expressions which are similar to the `SELECT` statements of SQL. FLWOR expressions allow to bind variables, iterate through sequences, and to define pattern for the result. A FLWOR expressions consists of the five clauses FOR, LET, WHERE, ORDER BY, RETURN. However a valid FLWOR expression does not need to contain all of them. The FOR clause is used to iterate through sequences. Moreover it can be used to join them in order to create a Cartesian product. A FOR clause is always returning a sequence using the RETURN clause. The ORDER BY clause allows to sort the items within the sequence in an ascending or descending manner. In order to filter the returned items of the sequence it is possible to define WHERE clauses which check whether a logical expression is fulfilled by the item or not. Beside FLWOR expressions, XQuery provides the possibility to union, intersect, or except sequences. Aggregate and arithmetic functions are also supported as well as control flow clauses. It is important to note that FLWOR expression can be nested within each other. Moreover variables can be bound to the returned sequence. In order to modularize the queries it is possible to define own or use existing functions.

```
(: query all notams encoded as AIXMBasicMessages :)
let $notams := (
  for $element in doc($db)//* where local-name($element) = "
      AIXMBasicMessage" order by xs:date($element/@detectionTimestamp)
      descending
    return $element
)
```

Listing 4.25: Example of a FLWOR expression.

Listing 4.25 depicts a FLWOR expression which returns a sequence which contains XML nodes with the local name `AIXMBasicMessage`. Moreover the sequence is ordered by the attribute value `detectionTimestamp` in a descending manner and bound to the variable `$notams`.

XQuery can be used to directly access single XML documents. However, usually XQuery is used to query collections of XML documents. As depicted in Figure 4.6 an XML database is used to store XML document collections. Besides the storing of multiple XML documents an XML database allows to use index techniques which increase the performance. The indexed collections can then be accessed by an XQuery engine which resolves the query formulated using FLWOR expressions. The retrieved sequence is returned as the query result.

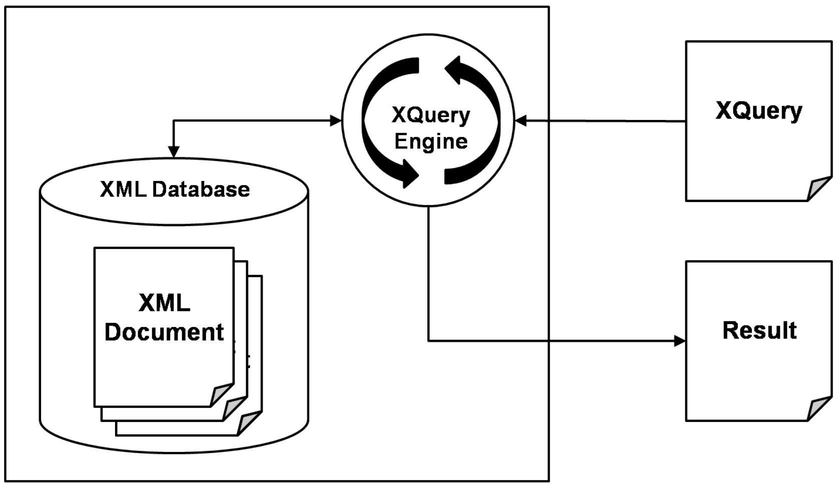

Figure 4.6: XQuery querying and transformation process (W3C, 2014).

Popular XML databases amongst others are eXistDB[2], BaseX[3], and MarkLogic[4]. For the prototype implementation the BaseX XML database is selected due to fact that it provides an XPath and XQuery 3.1 processor and therefore supports the latest version of XQuery. Furthermore BaseX provides an easy to use and interactive Graphical User Interface (GUI) frontend (BaseX, 2015). Structural indexes such as name, path, or resource indexes can be created. Moreover, value indexes are also supported which allow to index attributes, text and full-text.

XQuery Prototype Implementation: The XQuery prototype implementation is depicted in Listing B.2. Since XQuery and XSLT use the same data model both face the same problems which occurs using recursion. The outcome of the XQuery results in nested OL f-molecules. However, XQuery allows to implement an alternative solution using an end recursive approach. The reason for this can be found in the FLWOR expressions which can easily be used to retrieve sequences of items without outputting them directly. This fact is crucial since it allows to return sequences to outer FLWOR expressions. Especially when missing identifiers are generated they can be used as a value of the association while still being able to return them to outer FLWOR expression. The outer-most FLWOR expression returns all the sequences which are then used to start the end recursion.

The basic idea of the XQuery prototype implementation is to create a sequence which contains the mapped f-molecules represented as elements and text. Later it is mapped to a textual representation of the f-molecules. The sequence is returned by the end recursion which returns all f-molecule sequences of AIXM features. The first step is to retrieve all NOTAM elements stored in the BaseX database (Listing B.2). The implementation retrieves only the NOTAM data sources since the other two data sources can be mapped the in the same way. These NOTAM elements represent the first level of the hierarchy. They are parsed at first by calling the parse function using an empty identifier and an element as input parameter. An empty identifier indicates that no identifier was generated and therefore the identifier of the XML id attribute should be used. Instead of directly mapping the element to a textual representation a sequence containing the XML element `<fmol>` is created. This sequence contains the id-term, the OL class, the attributes of the element, and the children of the element. These

[2] `http://exist-db.org/exist/apps/homepage/index.html`
[3] `http://basex.org/`
[4] `http://marklogic.com`

child elements represent AIXM attributes which either include text content or represent an associations to other AIXM features. The child elements are looped and for each of them the attribute name, its attributes, the content, and, most important, the associated AIXM features are stored in XML elements. The associated AIXM features are stored in the `<toParse>` element because they represent the children of the AIXM attribute. Since the identifier of them is used later as the value of an OL attribute it is determined and added to the sequence as the element `<subobjid>`. Besides that the associated AIXM feature is stored in the element `<subobjname>`. After the sequence containing the f-molecule element is build it is returned. This sequence contains all F-atoms and data-F-atoms of the f-molecule encoded as elements. However up to now no recursion was started. The recursion starts when the f-molecules of the associated AIXM features are parsed. All these AIXM features are stored in the `<toParse>` element of the already parsed NOTAMs. Therefore all elements in the `<toParse>` element are iterated and for each of them the parse function is called. The value of the `<subobjid>` is used as the id parameter and the `<subobjname>` element is used as the element parameter. Again a sequence containing the `<fmol>` element is build as described above but this time the given parameter is used as the id-term. After the sequence is build the recursion is started by calling the parse function for all elements which are included in the `<toParse>` element of the newly parsed `<fmol>` element. When the end recursion is finished all sequences are returned. Since the element `<subobjname>` is not needed anymore it is removed from the sequences. Afterwards the strings within the sequences are joined and the outcome is combined and returned.

4.3.3 SAX and StAX Parser

SAX and StAX parser are used to efficiently parse huge XML documents. They do not create a tree structure like in XDM used by XQuery and XSLT. Instead of building a whole XML document tree, events are created and fired while performing the lexical analysis (Lam et al., 2008) (Megginson, 2001). Which means that SAX and StAX do not conduct the whole analysis phase, both stop after lexical analysis. As mentioned before the result of the lexical analysis are tokens such as start/end XML elements, attribute names and values, and text. Whenever a token is recognized by the SAX or StAX parser an event is fired in order to notify the accessing application. This allows to access partial data before the parsing of the whole XML document is complete. After the event is fired, the token and the event are destroyed which prevents a growing memory usage while parsing the entire

XML document. SAX and StAX parsers allow a memory efficient usage in contrast to XSLT and XQuery where the whole XML document tree is stored within the memory.

As depicted in Figure 4.7 SAX implements the so called push model (Lam et al., 2008). Whenever an event is fired, due to a detection of a token, the corresponding callback function is invoked. To invoke the corresponding callback function the application must provide an event handler which is registered to the parser. The callback functions determine how the event will be handled. They need to be defined in the accessing application. The SAX parser loops through the entire XML document and continuously checks which token is produced from the lexical analysis in order to fire the corresponding event and invoke thereby the corresponding callback function.

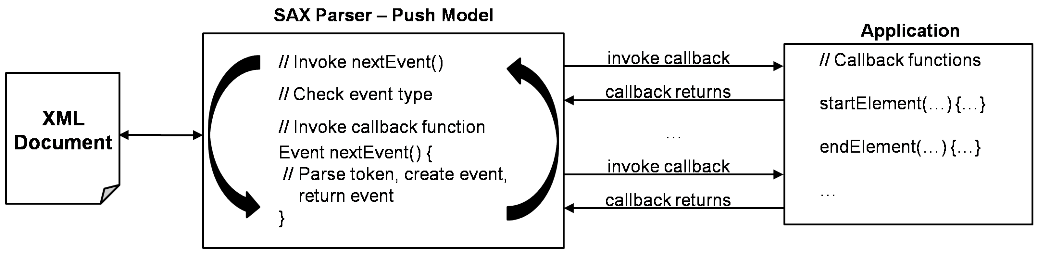

Figure 4.7: SAX push parsing approach (Lam et al., 2008).

In contrast to the push model of the SAX parser, the StAX parser is based on the so called pull model (Figure 4.8). Instead of continuously firing events and invoking the corresponding callback functions, the StAX parser waits until the application invokes the lexical analysis of the next token. The token is then parsed and the corresponding event is fired. This allows the application to skip uninterested events, in contrast to the SAX parser where all events must be handled.

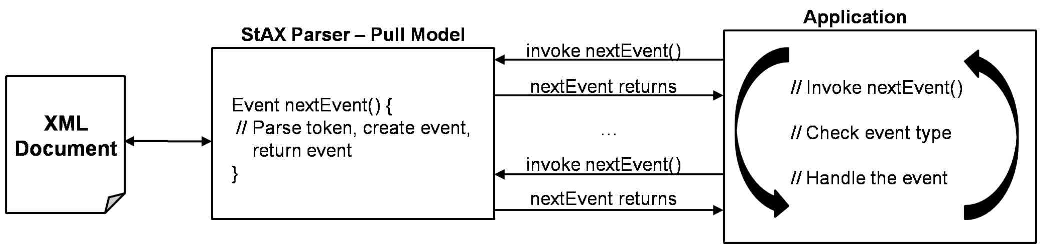

Figure 4.8: StAX pull parsing approach (Lam et al., 2008).

As indicated in Figure 4.7 and Figure 4.8 the transformation of the detected tokens must be handled by the application. Since both only provide access to the parsed tokens, no transformation, query or selection possibilities are given. The drawback of SAX and StAX is that XPath is not supported since no XML tree is available.

SAX and StAX Prototype Implementation: Due to the drawback that the SAX as well as the StAX parser do not provide the capability of directly selecting specific XML elements a prototype is not implemented. In order to apply the concepts introduced in Section 4.1 and Section 4.2 the capability to select specific elements is crucial. Especially the link resolution depends on this feature. Even if the link resolution can be implemented with a workaround the GML handling requires the selection capabilities. Without it the GML elements of the AIXM features cannot be selected which are needed in order to conduct the bounding box calculation described in Section 4.2.7.

4.3.4 DOM Parser

"*The DOM is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents*" (W3C, 2009). It is an API for HTML and XML documents which specifies the logical structure of documents as a collection of objects. Moreover, it describes how they can be accessed and manipulated. DOM is an object model specifying interfaces; it is not a set of data structures. Relationships such as the child or parent relationship are defined by logical relationships which are defined by the API. DOM can be used as a basis for implementing XML applications which process an XML document as if it were a tree.

Parsers implementing DOM can be used to represent and interact with objects in XML, HTML, or XHTML documents. In contrast to SAX and StAX parsers a DOM parser conducts the whole analysis phase returning an XML tree structure. This tree structure can be accessed using XPath expressions. Therefore the XML DOM parser provides methods to traverse the tree structure, access it, and insert new, or delete existing nodes within it. Moreover, the hierarchical structure of the XML document is preserved. Similar to XDM, DOM stores the whole tree structure in the memory which is why it is not as well performing as SAX or StAX parsers.

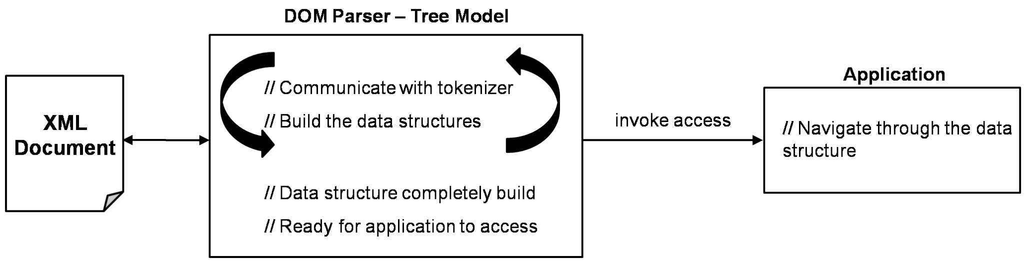
DOM Parser – Tree Model
// Communicate with tokenizer
// Build the data structures
// Data structure completely build
// Ready for application to access
XML Document
invoke access
Application
// Navigate through the data structure

Figure 4.9: DOM parsing approach (Lam et al., 2008).

Figure 4.9 depicts the DOM parsing process. The DOM parser accesses the XML document and loops through it. While looping through it the lexical analysis detects tokens which are added to the construction of the tree. After the XML document is completely iterated and the data structure is finished, it is returned to the application. The application can then access the DOM tree model and navigate through it using XPath expressions. However, the DOM parser only provides access to the XML document without providing transformation or query capabilities.

DOM Prototype Implementation: Similar to SAX and StAX the application needs to implement the mapping task since no transformation or query capabilities are provided. An object-orientated programming language is chosen for the application because a DOM XML parser represents the XML document as an object collection structured as a tree. Besides the fact that no transformation or query capabilities are provided, no additional data structures are provided by the DOM parser. Therefore the application needs to build an appropriate data structure where the parsed elements and their content can be stored. However, this allows to fully adapt the implementation according to the requirements.

The prototype implementation is depicted in Listing B.3. However, Listing B.3 is not providing the whole source code, it shows the core of the parser. As mentioned before the application needs to create a data model to store the mapped data. The data structure consists of an `OLProgram` class which contains multiple `OLNamespaces` and `OLFMolecules`. The `OLFMolecules` store its namespace, the OLClass and the OLAttributes which are represented by OLObjects. An OLObject is an abstract super class of OLLiteral and OLLink. This data structure is filled during the mapping task. The prototype starts with building an XML DOM tree based on a given input XML document. The root element is selected and the mapping is started by calling the `parseNode()`-method assigning the document, the root element, the hierarchy level zero and the name of the root element as the parent name to it. In the method the level is updated by checking if the parent has changed. When the parent has changed the level is increased and the parent is updated to the new one. After the level is updated it is checked whether the element represents an XML node since only XML nodes can be either an AIXM feature or an AIXM attribute. Due to the fact that AIXM features and AIXM attributes are alternating the element type is determined using a modulo-2-operation.

When an AIXM feature is encountered a new f-molecule is instantiated, initialized and assigned to the `OLProgram`. For each of the child elements of the AIXM feature the `parseNode()`-method is called starting the recursion. It stops when no child elements exists. When the modulo-2-operation encounters an AIXM attribute it is added to the parent f-molecule as an OL attribute. The value of this OL attribute is either text or representing an association to another AIXM feature. If it is an association the identifier of the associated AIXM feature is determined and added as the OL attribute value of the parent f-molecule. Moreover the level is updated and the `parseNode()`-method is invoked using the AIXM feature as a new input parameter.

The recursion stops after there are no more AIXM attributes of AIXM features or if there are no more child AIXM features of AIXM attributes. The filled data structure contains an OL program which includes all f-molecules and namespaces. These f-molecules and namespaces are iterated and output according to their corresponding OL syntax.

4.3.5 Evaluation and Selection

This sections evaluates the implemented prototypes in oder to select the best performing technology for the concrete implementation of the mapping task. The evaluation of the prototypes is conducted by measuring the time which is required to map a predefined number of NOTAMs to the corresponding OL representation. This will show how the prototypes perform while increasing the workload. The measurements are carried out on a computer with an fifth generation Intel(R) Core(TM) i7-5500U processor and 8GB Single Channel DDR3L 1600MHz RAM where one GB is directly assigned to the prototype application.

Therefore samples must be obtained in order to measure the time which is needed by the prototypes to map the given XML data sources. Six samples are created where each of them contains an logarithm increased number of NOTAMs. The logarithmic to the base ten is used which leads to samples containing one, ten, one hundred, 1000, and 10000 NOTAMs. For each of them the mapping time is measured in milliseconds.

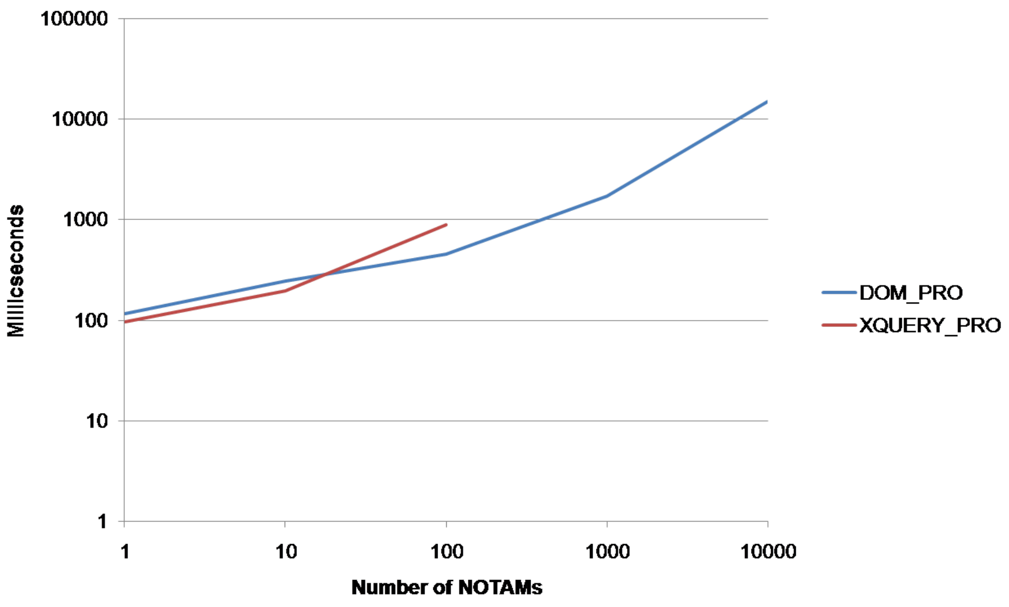

Figure 4.10: Evaluation of the XQuery and DOM prototype implementation.

Figure 4.10 depicts the line diagram which represents the measurements. The axes show the time in milliseconds and the number of NOTAMs. The blue line represents the measurements obtained by the DOM XML prototype implementation whereas the red line represents the measurements of the XQuery prototype implementation. Both prototypes map the samples of one and ten NOTAMs in nearly the same time. However it is shown, that the XQuery prototype performs slightly better. In contrast to that the DOM XML prototype outperforms the XQuery prototype while mapping one hundred NOTAMs. Moreover, it is shown that the red line stops after one hundred NOTAMs. The reason for that is that the query result size exceeds the limits of the database when 1000 NOTAMs are mapped. As shown the DOM XML prototype implementation performs well even with an increased workload. It supports the mapping of XML sample documents containing 1000 or 10000 NOTAMs in acceptable time.

Based on this performance evaluation the DOM XML prototype will be used for the implementation. Besides the better performance, the prototype allows to fully adapt the mapping process since no predefined transformation or query capabilities are used. Moreover, the created data structure can be reused and modified. It is important to note that the DOM XML prototype does not require an additional XML database, it accesses the XML documents directly.

4.4 DOM Extension Capabilities

In this section the extension capabilities of the chosen technology are delineated and the Requirement 3.3 which defines the extension capabilities is detailed. Since the DOM technology is selected to parse the XML input sources the basic parsing steps are described. Moreover, hooks are introduced which allow the extensions of the parsing steps.

As described in Section 3.2 the main task of the mapping is the transformation of NOTAMs, queries, and segments to their corresponding OL representation. Besides these three XML data sources other sources will be supported in the future since the XML schemas are still being developed. The introduction of new XML data sources can require to add new mapping and/or preprocessing tasks to the mapper. Moreover, new exceptions and restrictions can be introduced with them. As stated in Requirement 3.3 the mapper must provide the possibility to add new mapping and/or preprocessing tasks without modifying the existing implementation. To fulfill this the mapper needs to be designed to support future preprocessing and processing steps.

As long as the introduced XML data sources follow the object-property model the basic mapping concept of Section 4.1 can be applied without modifying the mapper. This is possible since the mapping, based on the object-property model, is not limited to data sources representing NOTAMs, queries, and segments. Each of them will be mapped to their corresponding OL representation. This allows the mapper to be used with more than the three named XML sources. However, processing tasks exceeding the basic mapping concept requires modifications. Therefore it is necessary that future modifications can extend the existing implementation without adapting it. To establish this the behavior of the DOM XML mapper must be modifiable.

Since the DOM parser does not provide any transformation or query capabilities traversing and mapping of the DOM tree must be implemented using a programming language (Section 4.3.4). Therefore the object-orientated programming language Java is chosen because it is already used in the implementation of the DOM parser prototype. The basic idea is to provide methods, so called hooks, which are located at different processing steps in the DOM parser. Hooks allow to modify the mapping task by simply extending their functionality. As depicted in Listing 4.26, several hooks are needed which handle XML root elements, XML exceptional elements, XML object elements, XML attribute elements, associated XML object elements, and the XML content.

```
// startMapping(...)
Level 0: handle root element; parse root element

// parseNode(...)
Level 1: handle exceptional elements if needed
Level 1: check element type
IF: OL object; handle object element; parse child elements
ELSE: OL attribute; handle attribute element; iterate through child
    elements
Level 2: handle object elements; parse child OL objects
Level 2: handle text elements
```

Listing 4.26: Basic steps of the parser.

The mapping starts by invoking the `startMapping()`-method. Level zero represents the root level and thereby covers the handling of the root element. When the root element is handled the parsing of it starts by invoking the `parseNode()`-method. By invoking this method the first parsing level is entered. In this first level exceptional elements can be handled. After exceptional elements are handled the differentiation between AIXM features which represent OL object and AIXM attributes which represent OL attributes is conducted in the first level. An element can either represent an OL object or an OL attribute. When an AIXM feature is identified the handling of it is invoked and child elements are parsed by invoking the `parseNode()`-method. When an AIXM attribute is detected the attribute element is handled and its child elements are iterated. By iterating over the child elements the second level is entered. Child elements representing AIXM features are handled by the object element handling of the second level and text elements are handled by the text handling.

Chapter 5

The Mapper

Contents

This section will introduce the conceptual design and the implementation of the AIXM to OL mapper using DOM. The conceptual design as well as the implementation are based on the previously developed mapping approach and the selected transformation technology (Section 4). Besides the extension capabilities the conceptual design covers also configuration capabilities, the interfaces, and the data model. The implementation realizes the mapping concept and the conceptual design. Finally, the implementation is evaluated in order to determine the performance.

5.1 Conceptual Design

The conceptual design covers the fulfillment of requirements which are beyond the functional requirements fulfilled by the mapping concepts introduced in Section 4.1 and Section 4.2. Besides the requirements the data model is detailed. Especially the Requirements 1.2, 1.3, 1.4, and 1.3 which define the configuration capabilities, Requirement 2.5 which describes the interfaces, and Requirement 3.3 which specifies the extension capabilities are comprised by the conceptual design.

5.1.1 Data Model

The data model which is implemented in the DOM XML prototype will be reused since it covers all OL statements which are needed. It is depicted in Figure 5.1 and consists of the six main classes `OLProgram`, `OLNamespace`, `OLFMolecule`, `OLObject`, `OLLink`, and `OLLiteral`. The detailed diagram is depicted in Appendix A.1.

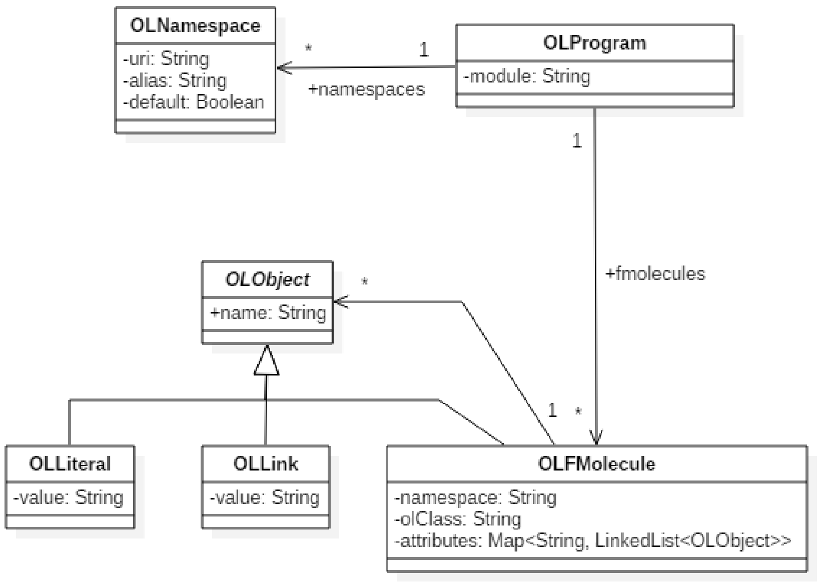

Figure 5.1: UML class model representing OL statements.

The `OLProgram` class is the main class of the data model since it contains all namespaces and all f-molecules. The `OLNamespaces` class contains data about an OL namespace. This covers the namespace URI, the prefix (alias), and whether it is the default namespace or not. The `OLFMolecules` contains data about the namespace, the OL class, and the OL attributes. The OL attributes are represented as a map where the key is represented by the OL attribute name and the value by a list of `OLObjects`. The abstract `OLObject` class contains a name and is extended by the classes `OLLiteral`, `OLLink` and `OLFMolecule`. The classes `OLLiteral` and `OLLink` are used to distinguish whether an OL attribute contains an literal or refers to an OL object.

5.1.2 Configuration Capabilities and Interfaces

As described in the Requirements 1.2, 1.3, 1.4, and 1.3 it must be possible to configure the mapper. Configuration capabilities allow to modify the implementation without the need of adapting the code. Therefore class attributes are used which can be accessed and set.

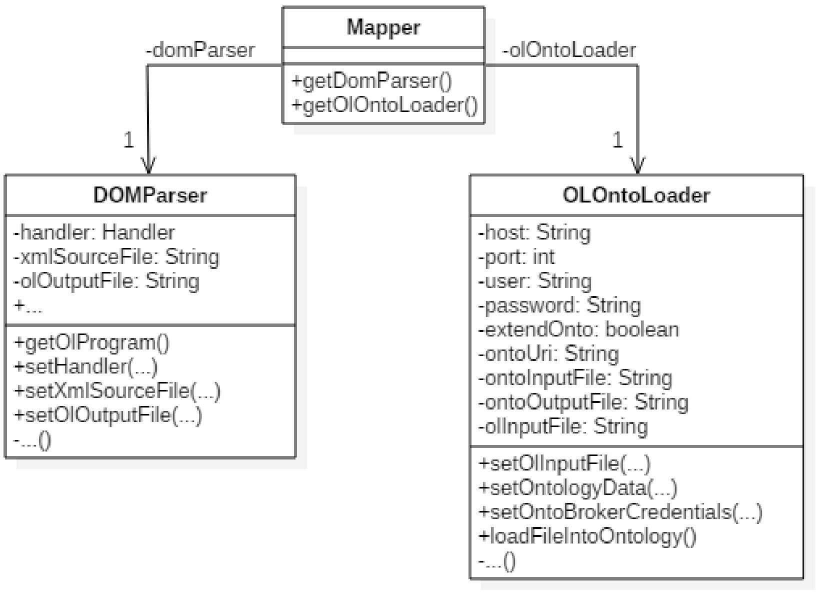

Figure 5.2: UML class model depicting the configuration capabilities.

Figure 5.2 depicts the structure of the mapper. The detailed diagram is depicted in Appendix A.2. The mapper consists of the two main classes `DOMParser` and `OLOntoLoader`. Both can be accessed by methods defined in the `Mapper` class. The `DOMParser` allows to configure the input data source and the OL output file. Moreover, the handler can be set which allows to apply the prior introduced decorated handler. The interface to the ontology API is provided by the `OLOntoLoader` class which allows to configure the OL input file, data about the ontology and the credentials to access the ontology API. The data about the ontology comprises an boolean value which indicates whether an ontology is extended or created, an ontology URI, the input file of an existing ontology, and the output file of the saved ontology. The input file is only needed when an existing ontology which is stored in a file needs to be accessed. The output file is always needed since it determines the location where the ontology is stored after the OL statements are inserted. The credentials comprise the host, port, user, and password which are required to access to ontology API. Moreover the `OLOntoLoader` class provides a method to start the loading process.

5.1.3 Extension Capabilities

To fulfill the Requirement 3.3 the conceptual design needs to comprise the DOM extension capabilities introduced in Section 4.4. The extension capabilities allow future modifications which extend the existing implementation without adapting it. Therefore design patterns which support the adaption of implementation behavior are analyzed.

As introduced in Section 4.4 the basic idea is to provide methods, so called hooks, which are located at different processing steps in the DOM parser. Hooks allow to modify the mapping task by simply extending their functionality. To provide extension capabilities it must be possible to extend these handling methods. A simple approach is to create subclasses which implement the specialized handling of the mapping process. In this case the super class would implement the handling methods representing the default handling of the data sources. A subclass can extend this super class and add new functionality to it such as the handling of GML data. This approach works fine as long as only the implementation of one subclass is used without combining them. However when it is needed to use several subclass implementations a new subclass implementation comprising all the functionality of the needed subclasses needs to be implemented. Another way is to create a new subclass which extends the subclasses and reuses their functionality. This can lead to an class explosion especially when it is needed to combine the functionality of three or more subclasses. Moreover, this approach requires multiple inheritance since the subclass of the subclasses extends several super classes. Besides these issues subclassing only allows to define the behavior in a static manner since it must be define before the compile time.

Instead of defining the behavior at compile time it is better to provide a design which supports to defining it at runtime. The Decorator pattern can be used to establish this. It provides a way to modify the behavior of individual objects (Cooper, 1998, p. 103f.). Therefore it can be used to change the behavior of individual objects from the same class without affecting the other ones' behavior. Moreover, the Decorator pattern allows to stack multiple decorators which allows to merge behavior without creating new subclasses or merging existing implementations in a new class. This is achieved by wrapping the original class by the decorator class.

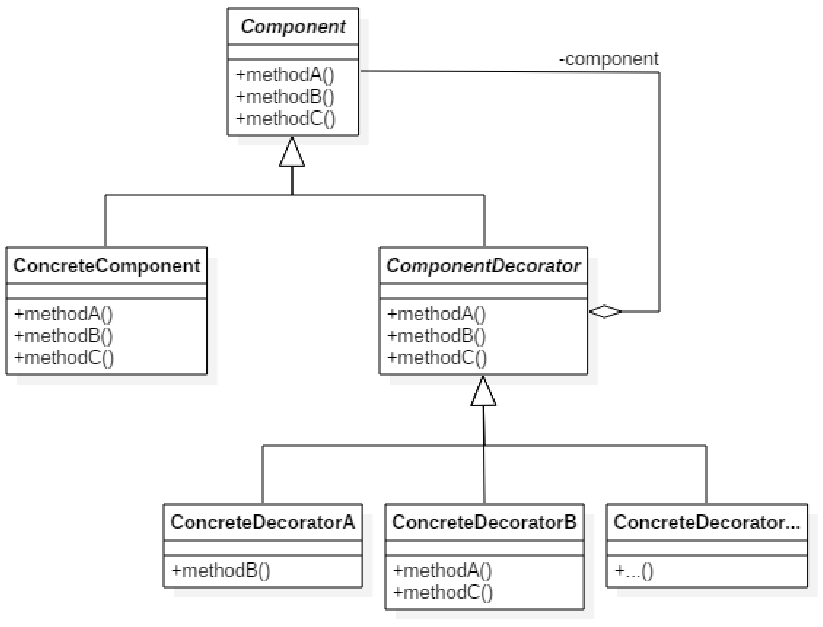

Figure 5.3: Concept of the Decorator Pattern (Cooper, 1998, p. 103f.).

As depicted in Figure 5.3 the original abstract class `Component` is extended by the two classes `ConcreteComponent` and `ComponentDecorator`. The `ConcreteComponent` implements all three methods of the `Component` class. The abstract `ComponentDecorator` contains a `Component` class attribute `component` which is initialized in the constructor. This class attribute is used to delegate all components methods. Therefore the methods of the `Component` class are invoked in the methods of the `ComponentDecorator` class. The `ConcreteDecoratorA` extends the `ComponentDecorator` and implements the behavior of the method which needs to be modified. The modified `methodA()` first calls `methodA()` of the super class and then continues its implementation. This allow to firstly invoke the behavior of a concrete implementation before the modifications are invoked. Besides the

`ConcreteDecoratorA` there can be several other decorator classes extending the behavior of the `Component`. All of them can be used to wrap the `ConcreteComponent` class and therefore add new behavior to it.

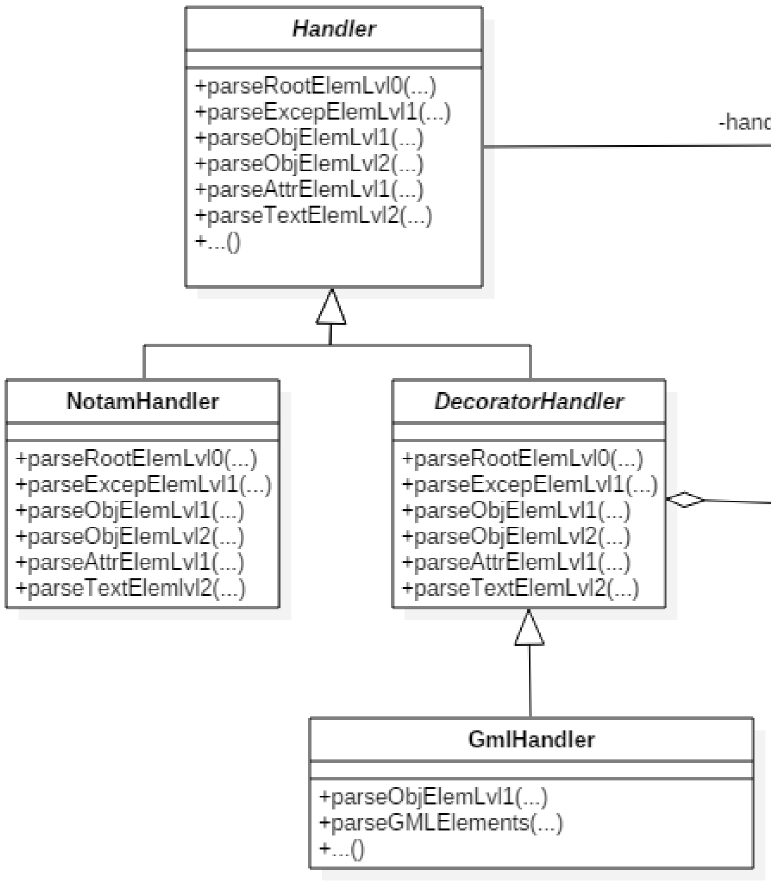

Figure 5.4: Decorator pattern applied to the mapper implementation design.

Figure 5.4 depicts the design of the `Handler` which is used by the DOM parser according to Listing 4.26. The detailed diagram is depicted in Appendix A.3. The abstract `Handler` class includes the methods which act as hooks. Moreover, it includes methods in order to ease the handling of XML elements within a DOM tree. The `NotamHandler` class extends the `Handler` class and implements the mapping concept defined in Section 4.1 and Section 4.2 except the GML handling of Section 4.2.7. The GML handling is added using a decorator `GmlHandling` which extends the `DecoratorHandler`. The `DecoratorHandler` includes a reference to the `Handler` class and delegates the method invocations to it. The `GmlHandling` class overrides the `parseObjElemLvl1()`-method since the handling should be conducted once for each NOTAM. A new concrete decorator needs to be added as a subclass of the `DecoratorHandler` if new preprocessing capabilities or data sources are introduced in the future.

5.2 Implementation

The implementation is based on the object-orientated programming language Java. It was chosen since it is already used in other implementations of the SemNOTAM system. In order to ease the understanding of the implementation detailed class diagrams are attached in the appendix in Chapter A.

As mentioned before the `DOMParser` class contains a handler which can be set using a decorated handler object. This handler provides the interface to the hooks. The parsing of the XML document is done by the Java API for XML (JAXP) DOM parser following the basic parsing steps defined in Listing 4.26. The parsing is comprised in the `DOMParser` class where the XML input data source is accessed by creating a DOM tree in the `startMapping()`-method. After the DOM tree is created the root XML element is accessed and handled as defined in the method `parseRootElemLvl0()` of the set handler. Despite its name the `parseRootElemLvl0()`-method does not parse the root element. It determines the `id` of queries and segments since it is available in an XML element instead of an XML attribute. This method provides the first hook since the parsing follows the basic parsing steps introduced in Listing 4.26. After the handling of the root element it is parsed by invoking the `parseNode()`-method with it as a parameter.

In the `parseNode()`-method exceptional elements are handled by the `parseExcepElemLvl1()`-method of the handler. After exceptional elements are handled the differentiation between AIXM features and AIXM attributes is conducted in the first level. This differentiation is conducted according to the hierarchical approach described in Section 4.1. If an AIXM feature is detected it is mapped to a f-molecule and added to the OL program as defined in Section 4.1. This mapping is again conducted by invoking the `parseObjElemLvl1()`-method of the defined handler. If an identifier is not available it is generated according to Section 4.2.5. The child elements of the AIXM feature, which represent AIXM attributes according to the object-property model, are parsed by invoking the `parseNode()`-method with them as a parameter.

If an AIXM attribute is detected it is added as an OL attribute to the f-molecule of the upper AIXM feature by invoking the handler method `parseAttrElemLvl1()`. Thereafter the content of the AIXM attribute is checked. It contains either text which is mapped according to the handler method `parseTextElemLvl2()` or one or more child AIXM features. These child AIXM features are parsed by invoking the `parseObjElemLvl2()`-method of the handler. Moreover, their child AIXM attributes are parsed by invoking the `parseNode()`-method with them as a parameter.

The recursion stops after there are no more AIXM attributes of AIXM features or if there are no more child AIXM features of AIXM attributes. The filled data structure contains an OL program which includes all f-molecules and namespaces. These f-molecules and namespaces are iterated and output according to their corresponding OL syntax.

This OL output file is used as the input file of the `OLOntoLoader` class. The `OLOntoLoader` class provides an interface to the OntoBroker accessing the OntoAPI. The OL statements are extracted from the file and added either to a new ontology or to an existing one.

5.2.1 NOTAM Handler

The `NotamHandler` implements the default handling of the data sources. Despite its name it also comprises the mapping of segment and query data sources since the mapping concept can be applied on them as well. The `NotamHandler` implements the hooks covering the functionality described in Section 4.2 except the calculation of the bounding boxes. The `parseRootElemLvl0()`-method implementation covers the determination of the identifier of queries and segments. Moreover it implements the mapping of the AIXM features and attributes. The `parseObjElemLvl1()`-method implementation adds the `detectionTimestamp` as defined in Section 4.2.4 to the data sources. The `parseObjElemLvl2()`-method adds the identifier of the AIXM feature as the value of the OL attribute to the upper f-molecule. The AIXM attributes are mapped by the implementation of the `parseAttrElemLvl1()`-method by adding them to the upper f-molecule as OL attributes. The `parseAttrElemLvl1()`-method also implements the handling of Section 4.2.6 and Section 4.2.1 while adding the OL attributes to the f-molecule. The `NotamHandler` also implements the `parseTextElemLvl2()`-method where the text content of OL attributes is added and formatted according to Section 4.2.2. Thus the NotamHandler is covering all tasks of the mapping concept except the handling of GML.

5.2.2 Example of an Extension - GML Handler

The implementation of the `GmlHandler` is based on the GeoTools[1] framework. It is available as an open source Java library providing tools for processing geospatial data. Therefore it is used to fulfill the preprocessing task described in Section 4.2.7. The GeoTools library allows to correctly interpret GML points according to their CRS. Moreover it provides the possibility to retrieve the GML envelope respectively the bounding box of GML shapes which consist of several GML points. However, it does not provide the possibility to retrieve the GML envelope for single points. To calculate the bounding box for a single point the radius of the bounding box is needed. As described in Section 4.2.7, the radius is used to determine the distance from the given GML point to the points which are needed to calculate the GML envelope. The GeoTools library provides a `GeodeticCalculator` to preform calculation based on GML data. It implements an improved version of the algorithm developed by Vincenty (1975) in order to calculate new points. Based on a center point, a given angle, and the distance the southern-, eastern-, northern-, and western-most points are calculated. Besides that the `GeodeticCalculator` supports the transformation of GML points based on different CRS.

The `GmlHandler` modifies only the `parseObjElemLvl1()`-method since the bounding box has to be calculated once for NOTAMs and their features. Therefore it is checked whether a bounding box calculation is required and if there are existing GML points. If a calculation is required and GML points are available the GML envelope is created and added to the DOM tree for each AIXM feature. The GML envelope of the NOTAM is added to the DOM tree as the outer-most bounding box.

5.2.3 Evaluation

In this section the final implementation is evaluated in oder to determine its performance. This evaluation follows the same approach and is conducted on the same computer as already used in Section 4.3.5. The time

[1]`http://www.geotools.org/`

in milliseconds which is required by the mapper to map a number of NOTAMs is measured. The obtained samples containing one, ten, one hundred, 1000, and 10000 NOTAMs are reused. Different configurations of the implementations are evaluated since the decorator pattern allows to add behavior such as the calculation of bounding boxes. Therefore the performance of the implementation using the `NotamHandler` (NH) is measured. In order to measure the overhead which is introduced by decorating the `NotamHandler` a new `TestHandler` is introduced. This `TestHandler` extends the `DecoratedHandler` class (TH+NH) and implements all provided methods. The implementation executes a simple addition of two numbers. The last measurements covers the performance of the `NotamHandler` which enriched by the `GmlHandler` (GH+NH).

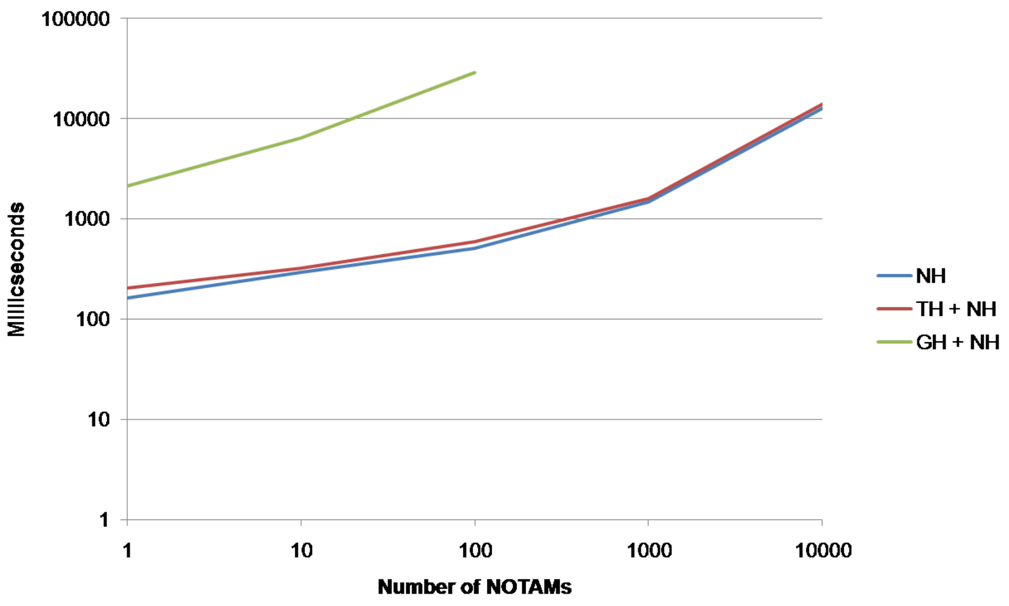

Figure 5.5: Evaluation of the implementation using different handler.

Figure 5.5 depicts the line diagram which represents the measurements. The blue line represents the measurements obtained by the `NotamHandler`, the red line represents the measurements obtained by the `NotamHandler` which is decorated by the `TestHandler`, and the green line represents the measurements obtained by the `NotamHandler` which is decorated by the `GmlHandler`. As shown the time required to conduct the mapping task of the `NotamHandler` is constantly increasing till the mapping of 1000 NOTAMs which is followed by a steeper increase. The decoration of the `NotamHandler` by the `TestHandler` is slightly decreasing the performance.

The overhead is recognizable but not remarkable. In contrast to that the decoration of the `NotamHandler` with the `GmlHandler` remarkably decreases the performance. As shown the mapping time for one NOTAM increased by a factor of ten from about 200 milliseconds to over 2000 milliseconds. Moreover, it is shown that the green line stops after one hundred NOTAMs. The reason for that is that the memory required to fulfill the mapping task exceeds the assigned space. The maximum number of NOTAMs which can be parsed by this configuration is 175.

As depicted in Listing 5.1 this bottleneck can be traced back to the parsing of GML points provided by the GeoTools library (Gundel, 2012) (Deoliveira, 2012). Moreover, it was tested to parse always the same GML point in order to avoid additional processing steps but this does not solved the issue. Besides that the `geoToolsParser` was newly initialized each time to force a cleaning of the heap space but this does not solved the issue either.

```
Exception in thread "main" java.lang.OutOfMemoryError: GC overhead limit exceeded
...
at com.sun.org.apache.xerces.internal.impl.XMLDocumentFragmentScannerImpl.scanDocument(Unknown Source)
at com.sun.org.apache.xerces.internal.parsers.XML11Configuration.parse(Unknown Source)
at com.sun.org.apache.xerces.internal.parsers.XMLParser.parse(Unknown Source)
at com.sun.org.apache.xerces.internal.parsers.AbstractSAXParser.parse(Unknown Source)
at com.sun.org.apache.xerces.internal.jaxp.SAXParserImpl$JAXPSAXParser.parse(Unknown Source)
at com.sun.org.apache.xerces.internal.jaxp.SAXParserImpl.parse(Unknown Source)
at org.geotools.xml.Parser.parse(Parser.java:240)
```

Listing 5.1: Excerpt of the parsing exception.

Chapter 6

Conclusion

This thesis covers the implementation of an extensible mapper which transforms AIXM and other structured data sources to OL. The mapper is a central component of the SemNOTAM system. The SemNOTAM system is developed to provide intelligent and fine-grained filtering capabilities of ATM related information. Besides other systems it supports the filtering of NOTAMs which contain safety critical, temporal, spatial information. Therefore the mapper accesses available structured XML input sources. These XML sources contain data about AIXM NOTAMs, queries and segments. All of them will be mapped to their corresponding OL representation following the object-property model. This allows the mapper to be used with more than the three named XML sources. The OL representation is then inserted into the SemNOTAM system.

Besides giving a theoretical background this thesis analyses and extracts the requirements for the mapper based on the given task description. Therefore the mapping in- and outputs, exceptions, and constraints on the processing task are analyzed. The resulting requirements are used to develop a mapping approach which details how the XML data sources are mapped to their corresponding OL representation. The mapping approach follows the object-property model and is able to execute the mapping under the given processing constraints as well as to handle exceptions. Especially the challenging handling of geographical data is covered in this mapping concept which defines how bounding boxes are calculated.

Moreover this thesis includes the evaluation and selection of suitable technologies for the implementation. Therefore the XML transformation technologies XSLT, SAX, StAX and DOM are analyzed. This covers an introduction to the technology and the implementation of a basic mapping prototype with it. The mapping prototypes are evaluated which shows that the DOM outperforms the other tested technologies.

A conceptual design providing configuration and extension capabilities is introduced in this thesis. The design is based on the previously developed mapping concept and the selected technology. Moreover, this thesis includes the implementation of the mapper which realizes the mapping concept and the conceptual design. Finally the mapper is evaluated using different configurations in order to determine its performance.

Concluding, future work can be conducted to optimize the performance of the mapper. Therefore preprocessing steps, especially the handling of geographical data, can be extracted from the mapper and executed separately. The bottleneck which occurs while parsing geographical data can be investigated. Moreover, the loading of the OL files into the ontology can be changed to omit the OL files and directly insert the OL statements stored in the data model. This can avoid unnecessary creating, writing, and reading file operations. At last future work can cover the export of NOTAMs by mapping the OL representation back to XML.

Bibliography

Aho, A. V., & Ullman, J. D. (1972). *The Theory of Parsing, Translation, and Compiling.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc.

Baader, F., Bauer, A., Baumgartner, P., Cregan, A., Gabaldon, A., Ji, K., ... Schwitter, R. (2009). A Novel Architecture for Situation Awareness Systems. In *Automated reasoning with analytic tableaux and related methods* (pp. 77–92). Springer Berlin Heidelberg.

BaseX. (2015). *BaseX Documentation.* Retrieved 2015-07-07, from `http://docs.basex.org/wiki/Main_Page`

Burgstaller, F., Steiner, D., Schrefl, M., Gringinger, E., Wilson, S., & van der Stricht, S. (2015). AIRM-based, fine-grained semantic filtering of notices to airmen. In *Integrated communication, navigation, and surveillance conference (icns), 2015* (pp. 1–13).

Burgstaller, F., Szabolcs, N., Steiner, D., & Frequentis AG. (2014). *SemNOTAM - Interfaces.*

Caldwell, D. R. (2005). *Unlocking the Mysteries of the Bounding Box.* Retrieved 2015-07-01, from `http://www.stonybrook.edu/libmap/coordinates/seriesa/no2/a2.htm`

Cooper, J. W. (1998). The design patterns Java companion. , 1–216. Retrieved 2015-06-22, from `https://drive.google.com/file/d/0B_6WIwPo1qCtSi1WdmZMYWNKcHc/view`

Cordell, B. (2006). *EnRoute Routes.* Washington DC. Retrieved 2015-06-06, from `http://www.aixm.aero/gallery/content/public/design_review/07c_aicm_routes.pdf`

Cox, S., Daisey, P., Lake, R., Portele, C., & Whiteside, A. (2005). *OGC 02-023r4 OpenGIS® Geography Markup Language (GML) Encoding Specification.*

Crocker, D., & Overell, P. (2005). *Augmented BNF for syntax specifications: ABNF.* Retrieved 2015-06-26, from `https://tools.ietf.org/pdf/rfc5234`

De Schrijver, D., De Neve, W., Van Deursen, D., De Cock, J., & de Walle, R. (2006, August). On an Evaluation of Transformation Languages in a Fully XML-Driven Framework for Video Content Adaptation. In *Innovative computing, information and control, 2006. icicic '06. first international conference on* (Vol. 3, pp. 213–216). doi: 10.1109/ICICIC.2006.487

Deoliveira, J. (2012). *Parsing a large GML2 document.* Retrieved 2015-07-14, from `http://sourceforge.net/p/geotools/mailman/message/30132309/`

EUROCONTROL. (2006a). *Aeronautical Informaiton Exchange Model - Standards.* Retrieved 2015-06-10, from `http://www.aixm.aero/public/standard_page/concepts_standards.html`

EUROCONTROL. (2006b). *Aeronautical Information Exchange Moder - Design Concepts.* Retrieved 2015-06-10, from `http://www.aixm.aero/public/standard_page/concepts_design.html`

EUROCONTROL. (2006c). *AIS to AIM AIS Data and Web Services Worldwide implementation of AIS interoperability AICM and AIXM.* Retrieved 2015-06-29, from `http://www.aixm.aero/gallery/content/public/design_review_2/C-AICMandAIXMIntroduction.pdf`

EUROCONTROL. (2008). *UML to XML Schema Mapping.* Retrieved 2015-06-06, from `http://www.aixm.aero/gallery/content/public/final_release_5_0/design_docs/AIXMUMLtoAIXMXSDMapping1.0.pdf`

EUROCONTROL. (2010). *Digital Notam - The Digital Age.* Retrieved 2015-06-07, from `https://www.eurocontrol.int/sites/default/files/publication/files/20100601-digitalnotam-brochure.pdf`

EUROCONTROL. (2012). *ATM Information Reference Model (AIRM).* Retrieved 2015-06-05, from `https://www.eurocontrol.int/sites/default/files/publication/files/2013-swim-airm-factsheet.pdf`

EUROCONTROL. (2014). *EUROCONTROL Seven-Year Forecast February 2014 - 7-year IFR Flight Movements and Service Units Forecast : 2014-2020.* Retrieved 2015-06-07, from `https://www.eurocontrol.int/sites/default/files/content/documents/official-documents/forecasts/seven-year-flights-service-units-forecast-2014-2020-feb2014.pdf`

EUROCONTROL. (2015a). *Aeronautical Information Exchange Model (Phase 3 P-09).* Retrieved 2015-06-10, from `https://www.eurocontrol.int/services/aeronautical-information-exchange-model-phase-3-p-09`

EUROCONTROL. (2015b). *Digital NOTAM (Phase 3 P-21).* Retrieved 2015-06-05, from `https://www.eurocontrol.int/articles/digital-notam-phase-3-p-21`

EUROCONTROL. (2015c). *European AIS Database - Performance Statistics Year over Year Evolution.* Retrieved 2015-06-05, from `https://www.ead.eurocontrol.int/eadcms/eadsite/operations/performance/past.html`

EUROCONTROL. (2015d). *What is air traffic management?* Retrieved 2015-06-13, from `https://www.eurocontrol.int/articles/what-air-traffic-management`

EUROCONTROL, & Federal Aviation Administration. (2006). *Aeronautical Information Exchange Model.* Retrieved 2015-06-10, from `http://www.aixm.aero`

EUROCONTROL, & Federal Aviation Administration. (2007). *AIXM is GML.* Retrieved 2015-06-10, from `http://www.aixm.aero/gallery/content/public/2007_10_us_class/AIXM0710Class1H5AIXM5AIXMisGML_final.pdf`

EUROCONTROL, & Federal Aviation Administration. (2010). *AIXM 5 Temporality Model.* Retrieved 2015-06-12, from `http://www.aixm.aero/gallery/content/public/release_candidate_3/AIXMTemporality05.pdf`

EUROCONTROL, & Federal Aviation Administration. (2011a). *AIXM 5 - Feature Identification and Reference.* Retrieved 2015-06-13, from `http://www.aixm.aero/gallery/content/public/AIXM51/AIXM_Feature_Identification_and_Reference-1.0.pdf`

EUROCONTROL, & Federal Aviation Administration. (2011b). *AIXM Digital NOTAM Event Specification.* Retrieved 2015-06-16, from `https://www.eurocontrol.int/sites/default/files/content/documents/information-management/20101010-digitalnotam-event-specification-increment1.pdf`

European Union. (2015). *State of Harmonisation Document* (Tech. Rep.). Retrieved from `http://www.sesarju.eu/sites/default/files/documents/reports/State-of-Harmonisation.pdf?issuusl=ignore`

Federal Aviation Administration. (2010). *FAI FSS - NOTAM Overview.* Retrieved 2015-06-07, from `http://www.faa.gov/about/office_org/headquarters_offices/ato/service_units/systemops/fs/alaskan/alaska/fai/notam/ntm_overview/`

Frazier, M. (2013). *Overview of Coordinate Reference Systems (CRS) in R.* Retrieved 2015-07-01, from `https://www.nceas.ucsb.edu/~frazier/RSpatialGuides/OverviewCoordinateReferenceSystems.pdf`

Frequentis AG. (2015). *semNotam.* Retrieved 2015-06-04, from `http://www.frequentis.com/en/at/group/frequentis-core`

-competences/technology-leadership/semnotam/#!

Geospatial Intelligence TWG. (2006). *AIXM v4.5.* Retrieved 2015-06-20, from http://www.gwg.nga.mil/documents/asfe/AIXM_v4.5.pdf

Gringinger, E. (2014). *Delivering Digital Services.* Retrieved 2015-06-10, from http://www.aixm.aero/gallery/content/public/2014_08_ATIEC/Day_2/08-semNOTAMS-EduardGringinger.pptx.

Gringinger, E., Eier, D., & Merkl, D. (2011). NextGen and SESAR moving towards ontology-based software development. In *Integrated communications, navigation and surveilance conference (icns), 2011* (pp. H3–1–H3–10).

Gringinger, E., Trausmuth, G., Balaban, A., Jahn, J., & Milchrahm, H. (2012). Experience report on successful demonstration of SWIM by three industry partners. In *Integrated communications, navigation and surveillance conference (icns), 2012* (pp. G6–1 – G6–8).

Gruber, T. R. (1995). Toward principles for the design of ontologies used for knowledge sharing? *International journal of human-computer studies*, *43*(5), 907–928.

Gundel, A. (2012). *geoserver 2.2 running out of memory.* Retrieved 2015-07-14, from http://osgeo-org.1560.x6.nabble.com/geoserver-2-2-running-out-of-memory-td5004112.html

IBM. (2010). *XQuery and XPath data model.* Retrieved 2015-07-06, from http://www-01.ibm.com/support/knowledgecenter/SSEPGG_9.7.0/com.ibm.db2.luw.xml.doc/doc/xqrdatam.html

Icao-Ais-Aimsg. (2011). Aeronautical Information Services-Aeronautical Information Management Study Group (AIS-AIMSG) Fourth meeting - Study note 2.

International Civil Aviation Organization. (2013). *NOTAM PROFLERATION ANALYSIS* (Tech. Rep. No. January 2012). Retrieved from http://www.icao.int/airnavigation/IMP/MeetingDocs/IMP-1/WP/IMP-1WP-05IATANotampaper-v2.pdf

Kifer, M., Lausen, G., & Wu, J. (1995). Logical Foundations of Object-oriented and Frame-based Languages. *J. ACM*, *42*(4), 741–843.

Lam, T., Ding, J. J., & Liu, J.-C. (2008, September). XML Document Parsing: Operational and Performance Characteristics. *Computer*, *41*(9), 30–37.

Leach, P., Microsoft, Mealling, M., Networks, L. R., Salz, R., & Inc., D. T. (2005). *RFC4122 - A Universally Unique IDentifier (UUID) URN Namespace Status.* Network Wokring Group.

Megginson, D. (2001). Sax 2.0: The simple api for xml. *SAX project.* Retrieved 2015-07-01, from http://www.saxproject.org/

Myers, P. W. (1978). Department of the Air Force. *Military medicine*, *143*(9), 613–618.

Rijmen, V., & Oswald, E. (2005). Update on SHA-1. In *Topics in cryptology–ct-rsa 2005* (pp. 58–71). Springer.

Rivest, R. (1992). The MD5 message-digest algorithm. , 1–21. Retrieved from https://www.ietf.org/rfc/rfc1321.txt

Schrefl, M. (2014). *Forschungsprojektdetails.* Retrieved 2015-06-04, from http://fodok.uni-linz.ac.at/fodok/forschungsprojekt.xsql?FP_ID=3342

Selinger, P. (2006). *Collisions in the MD5 cryptographic hash function.* Retrieved 2015-05-24, from http://www.mscs.dal.ca/~selinger/md5collision/

semafora systems GmbH. (2012). *ObjectLogic Tutorial.* Retrieved 2015-06-28, from http://www.semafora-systems.com/fileadmin/user_upload/Publications_EN/ObjectLogic_Tutorial.pdf

semafora systems GmbH. (2013). *ObjectLogic Reference Guide.* Retrieved 2015-06-27, from http://help.semafora-systems.com/index.html

Steiner, D., Burgstaller, F., & Schrefl, M. (2014). *SemNOTAM Project Documentation - Generic Concepts, Knowledge Structure and Queries.*

Stevens, M. (2006). Fast Collision Attack on MD5. *IACR Cryptology ePrint Archive*, *2006*, 104. Retrieved from https://marc-stevens.nl/research/papers/eprint-2006-104-S.pdf

Stevens, M. (2007). *On collisions for MD5.* Retrieved 2015-06-29, from http://www.win.tue.nl/hashclash/OnCollisionsforMD5-M.M.J.Stevens.pdf

Suzuki, K., Tonien, D., Kurosawa, K., & Toyota, K. (2006). Birthday paradox for multi-collisions. In *Information security and cryptology–icisc 2006* (pp. 29–40). Springer.

The Open Group. (1997). *Universal Unique Identifier.* Retrieved 2015-06-26, from http://pubs.opengroup.org/onlinepubs/9629399/apdxa.htm

United States Census Bureau. (1997). *GIS FAQ Q5.1: Great circle distance between 2 points.* Retrieved 2015-07-03, from http://www.movable-type.co.uk/scripts/gis-faq-5.1.html

U.S. Department of Commerce, & National Institute of Standards and Technology. (2012). *180-4-Federal Information Processing Standards Publication-Secure Hash Standard (SHS)-National Institute of Standards and Technology Gaithersburg.* MD. Retrieved from http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf

Vincenty, T. (1975). Direct and inverse solutions of geodesics on the ellipsoid with application of nested equations. *Survey review*, *23*(176), 88–93.

W3C. (2007). *XSL Transformations (XSLT) Version 2.0.* Retrieved 2015-

07-06, from http://www.w3.org/TR/xslt20/

W3C. (2009). *Document Object Model (DOM).* Retrieved 2015-07-08, from http://www.w3.org/DOM/

W3C. (2014). *XQuery 3.0: An XML Query Language.* Retrieved 2015-07-06, from http://www.w3.org/TR/xquery-30/

W3C. (2015). *Extensible Markup Language (XML).* Retrieved 2015-06-10, from http://www.w3.org/XML/http://www.w3.org/standards/xml/

Zimmer, N., Schiefele, J., Bayram, K., Hankers, T., Frank, S., & Feuerle, T. (2011). Rule-Based NOTAM & Weather Notification. In *Navigation and surveillance conference (icns)* (pp. 01–1 – 01–9).

List of Figures

List of Tables

List of Listings

Acronyms

2.5D 2.5 Dimension. 25

ABNF Augmented Backus-Naur Form. 58

AICM Aeronautical Information Conceptual Model. 21

AIM Aeronautical Information Management. 2, 8, 15, 18, 21, 40, 41

AIRM ATM Information Reference Model. 19

AIS Aeronautical Information Services. 2, 15, 17, 18

AIXM Aeronautical Information Exchange Model. 2, 4, 5, 7, 9, 11, 13, 14, 17–20, 22–27, 35–38, 43–45, 47–51, 54–58, 62–67, 69, 78, 80, 82, 86, 87, 93–95, 98, 106, 109, 110

API Application Programming Interface. III, 13, 80, 81, 89, 90, 93, 94, 112, 113

ATM Air Traffic Management. III, 1, 2, 8, 14, 17, 19, 26, 98, 111

CRS Coordinate Reference System. 45, 70, 71, 95

DCE Distributed Computing Environment. 60

DOM Document Object Model. I, III, IV, 47, 73, 75, 80–85, 87, 88, 90, 92, 93, 95, 99, 106, 110, O

EAD European Aeronautical Information Services Database. 15, 17

EUROCONTROL European Organisation for the Safety of Air Navigation. 1, 2, 8, 16–18

FAA Federal Aviation Administration. 2, 8, 14, 16–18

FIXM Flight Information Exchange Model. 19

FNS-NDS Federal NOTAM Service and NOTAM Distribution Service. 3

GML Geography Markup Language. 19, 24–27, 37, 39–41, 45, 50, 67–71, 75, 80, 87, 90, 92, 95, 106, 110

GNSS Global Navigation Satellite System. 15

GUI Graphical User Interface. 77

HTML HyperText Markup Language. 37, 44, 45, 54, 55, 73, 80, 81

ICAO International Civil Aviation Organization. 17

ISO International Organization for Standardization. 19

JAXP Java API for XML. 93

KB Knowledge Base. 3, 4, 10, 11, 45

KBF Knowledge-Based Framework. 3–5, 10–13

MAC Media-Access-Control. 59–61

MD5 Message Digest 5. 60

NAVAID Navigational Aid. 14, 21

NextGen Next Generation Air Transportation System. 8

NOTAM Notices to Airmen. 2–5, 8–18, 24, 26, 34–38, 40–44, 48, 49, 53–55, 57, 66, 67, 69, 71, 78, 83–85, 87, 93–99, 109

OL ObjectLogic. III, IV, 5, 13, 14, 27–33, 35, 36, 38, 39, 43–45, 48, 50–58, 62, 65, 66, 71, 72, 75, 77, 78, 82–90, 93, 94, 98, 99, 107, 109, 110, B

PIB Pre-flight Information Bulletin. 15

POSIX Portable Operating System Interface. 60

SAX Simple API for XML. III, 73, 79–81, 99, 106

SemNOTAM Semantic NOTAM. II–IV, 3–5, 7–13, 27, 35, 40, 41, 45, 67, 93, 98, 106

SESAR Single European Sky ATM Research Program. 8

SHA-1 Secure Hash Algorithm. 60

SQL Structured Query Language. 75, 76

StAX Streaming API for XML. III, 73, 79–81, 99, 106

SWIM System Wide Information Management. 19

UID User Identifier. 60

UML Unified Modelling Language. 20, 22, 23, 49, 88, 89, 107

URI Uniform Resource Identifier. 52, 53, 88, 90

URL Uniform Resource Locator. 60, 66

UTC Coordinated Universal Time. 59

UUID Universal Unique Identifier. 26, 58–63, 108–110

W3C World Wide Web Consortium. 75

WXXM Weather Information Exchange Models and Schema. 19

XDM XQuery and XPath data model. 73, 75, 79, 81

XLink XML Linking Language. 27, 43, 64–66, 110

XML Extensible Markup Language. III, IV, 7, 13, 19–22, 27, 34–38, 43–45, 48–53, 55–58, 63–69, 71–85, 88, 92, 93, 98, 99, 109, 110, 112, 113, O

xPath XML Path Language. 64, 73–75, 77, 80, 81

xPointer XML Pointer Language. 64, 66, 110

XQuery XML Query Language. L, 47, 73, 75–79, 83, 84, 106, 110

XSD XML Schema Definition. 20, 22, 35, 50, 72

XSLT Extensible Stylesheet Language Transformation. III, 73–75, 77, 79, 99, 106, 110, H

Appendix A

Class Diagrams

The UML diagrams presented in this chapter are not complete. They should provide an overview and support the understanding of the implementation.

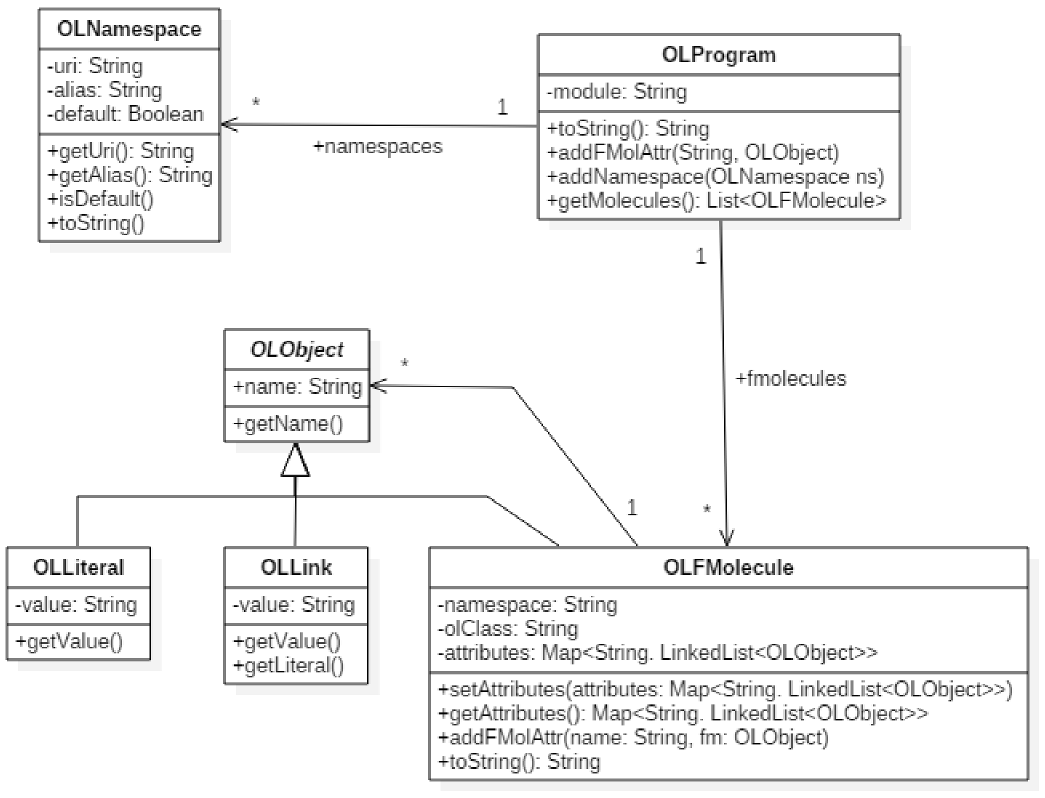

Figure A.1: The UML class diagram of the OL statements.

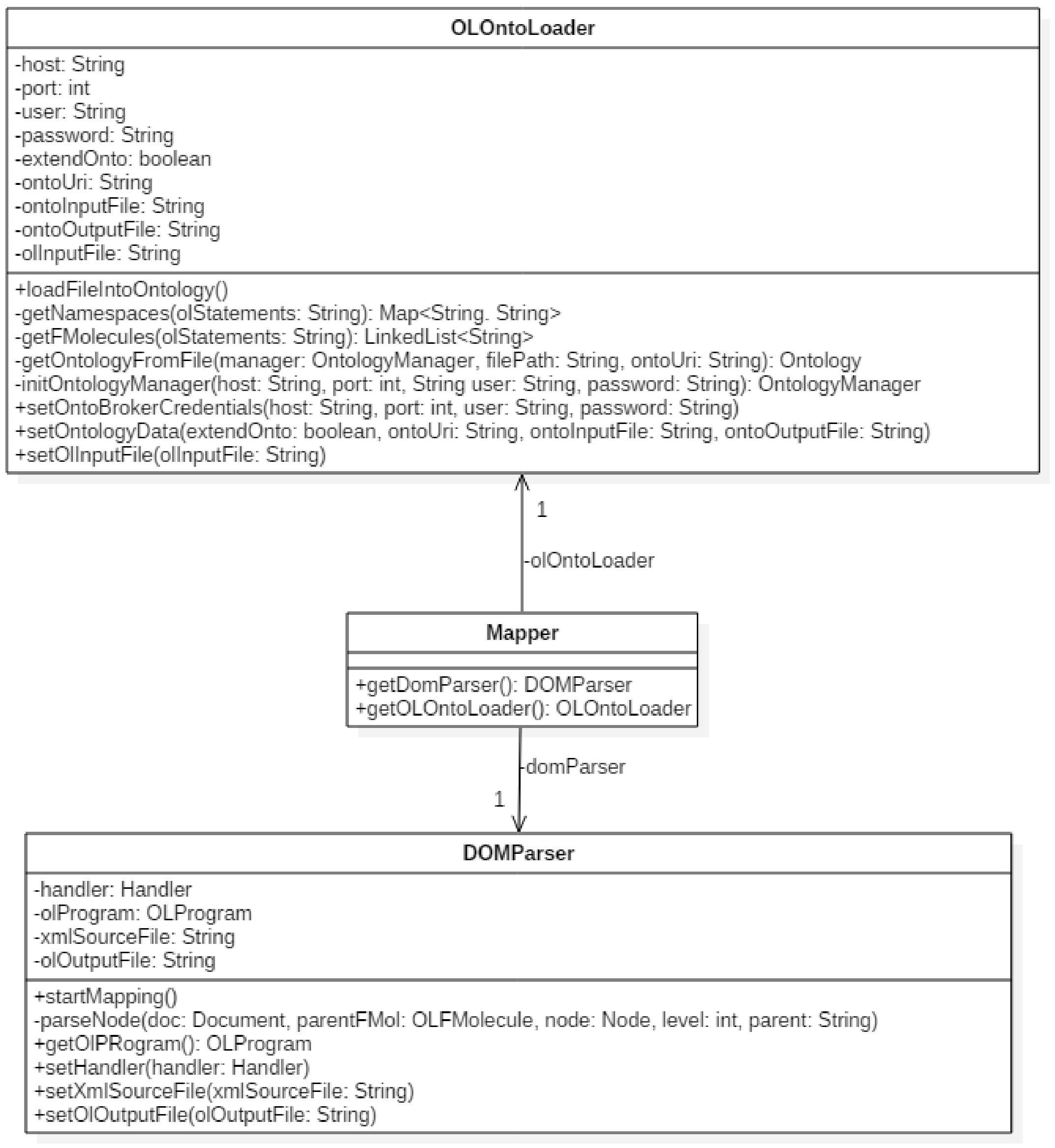
OLOntoLoader
-host: String
-port: int
-user: String
-password: String
-extendOnto: boolean
-ontoUri: String
-ontoInputFile: String
-ontoOutputFile: String
-olInputFile: String
+loadFileIntoOntology()
-getNamespaces(olStatements: String): Map<String. String>
-getFMolecules(olStatements: String): LinkedList<String>
-getOntologyFromFile(manager: OntologyManager, filePath: String, ontoUri: String): Ontology
-initOntologyManager(host: String, port: int, String user: String, password: String): OntologyManager
+setOntoBrokerCredentials(host: String, port: int, user: String, password: String)
+setOntologyData(extendOnto: boolean, ontoUri: String, ontoInputFile: String, ontoOutputFile: String)
+setOlInputFile(olInputFile: String)
1
-olOntoLoader
Mapper
+getDomParser(): DOMParser
+getOLOntoLoader(): OLOntoLoader
-domParser
1
DOMParser
-handler: Handler
-olProgram: OLProgram
-xmlSourceFile: String
-olOutputFile: String
+startMapping()
-parseNode(doc: Document, parentFMol: OLFMolecule, node: Node, level: int, parent: String)
+getOlPRogram(): OLProgram
+setHandler(handler: Handler)
+setXmlSourceFile(xmlSourceFile: String)
+setOlOutputFile(olOutputFile: String)

Figure A.2: The UML class diagram of the interface and its concrete implementation.

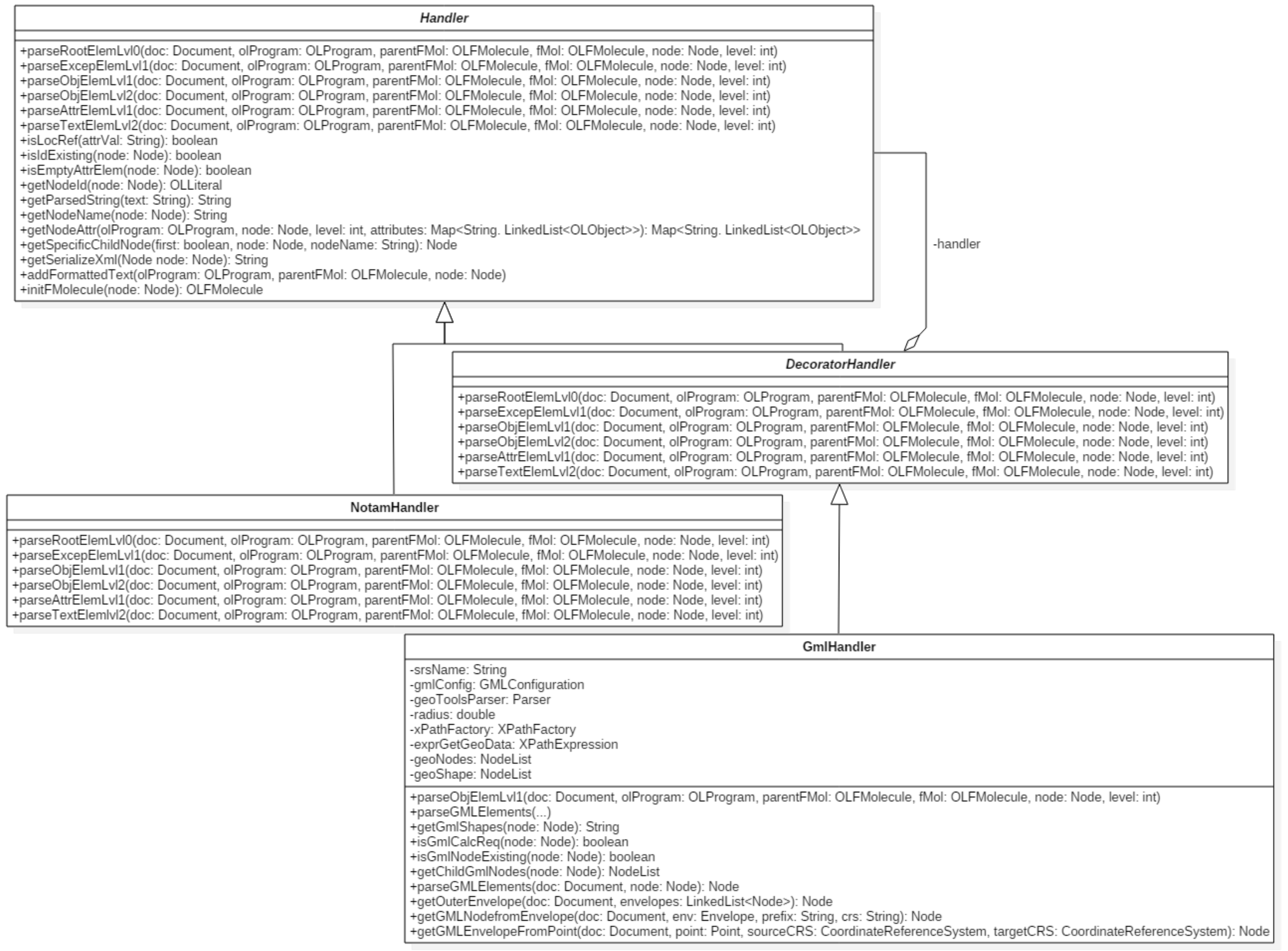
Handler
+parseRootElemLvl0(doc: Document, olProgram: OLProgram, parentFMol: OLFMolecule, fMol: OLFMolecule, node: Node, level: int)
+parseExcepElemLvl1(doc: Document, olProgram: OLProgram, parentFMol: OLFMolecule, fMol: OLFMolecule, node: Node, level: int)
+parseObjElemLvl1(doc: Document, olProgram: OLProgram, parentFMol: OLFMolecule, fMol: OLFMolecule, node: Node, level: int)
+parseObjElemLvl2(doc: Document, olProgram: OLProgram, parentFMol: OLFMolecule, fMol: OLFMolecule, node: Node, level: int)
+parseAttrElemLvl1(doc: Document, olProgram: OLProgram, parentFMol: OLFMolecule, fMol: OLFMolecule, node: Node, level: int)
+parseTextElemLvl2(doc: Document, olProgram: OLProgram, parentFMol: OLFMolecule, fMol: OLFMolecule, node: Node, level: int)
+isLocRef(attrVal: String): boolean
+isIdExisting(node: Node): boolean
+isEmptyAttrElem(node: Node): boolean
+getNodeId(node: Node): OLLiteral
+getParsedString(text: String): String
+getNodeName(node: Node): String
+getNodeAttr(olProgram: OLProgram, node: Node, level: int, attributes: Map<String. LinkedList<OLObject>>): Map<String. LinkedList<OLObject>>
+getSpecificChildNode(first: boolean, node: Node, nodeName: String): Node
+getSerializeXml(Node node: Node): String
+addFormattedText(olProgram: OLProgram, parentFMol: OLFMolecule, node: Node)
+initFMolecule(node: Node): OLFMolecule
-handler
DecoratorHandler
+parseRootElemLvl0(doc: Document, olProgram: OLProgram, parentFMol: OLFMolecule, fMol: OLFMolecule, node: Node, level: int)
+parseExcepElemLvl1(doc: Document, olProgram: OLProgram, parentFMol: OLFMolecule, fMol: OLFMolecule, node: Node, level: int)
+parseObjElemLvl1(doc: Document, olProgram: OLProgram, parentFMol: OLFMolecule, fMol: OLFMolecule, node: Node, level: int)
+parseObjElemLvl2(doc: Document, olProgram: OLProgram, parentFMol: OLFMolecule, fMol: OLFMolecule, node: Node, level: int)
+parseAttrElemLvl1(doc: Document, olProgram: OLProgram, parentFMol: OLFMolecule, fMol: OLFMolecule, node: Node, level: int)
+parseTextElemLvl2(doc: Document, olProgram: OLProgram, parentFMol: OLFMolecule, fMol: OLFMolecule, node: Node, level: int)
NotamHandler
+parseRootElemLvl0(doc: Document, olProgram: OLProgram, parentFMol: OLFMolecule, fMol: OLFMolecule, node: Node, level: int)
+parseExcepElemLvl1(doc: Document, olProgram: OLProgram, parentFMol: OLFMolecule, fMol: OLFMolecule, node: Node, level: int)
+parseObjElemLvl1(doc: Document, olProgram: OLProgram, parentFMol: OLFMolecule, fMol: OLFMolecule, node: Node, level: int)
+parseObjElemLvl2(doc: Document, olProgram: OLProgram, parentFMol: OLFMolecule, fMol: OLFMolecule, node: Node, level: int)
+parseAttrElemLvl1(doc: Document, olProgram: OLProgram, parentFMol: OLFMolecule, fMol: OLFMolecule, node: Node, level: int)
+parseTextElemlvl2(doc: Document, olProgram: OLProgram, parentFMol: OLFMolecule, fMol: OLFMolecule, node: Node, level: int)
GmlHandler
-srsName: String
-gmlConfig: GMLConfiguration
-geoToolsParser: Parser
-radius: double
-xPathFactory: XPathFactory
-exprGetGeoData: XPathExpression
-geoNodes: NodeList
-geoShape: NodeList
+parseObjElemLvl1(doc: Document, olProgram: OLProgram, parentFMol: OLFMolecule, fMol: OLFMolecule, node: Node, level: int)
+parseGMLElements(...)
+getGmlShapes(node: Node): String
+isGmlCalcReq(node: Node): boolean
+isGmlNodeExisting(node: Node): boolean
+getChildGmlNodes(node: Node): NodeList
+parseGMLElements(doc: Document, node: Node): Node
+getOuterEnvelope(doc: Document, envelopes: LinkedList<Node>): Node
+getGMLNodefromEnvelope(doc: Document, env: Envelope, prefix: String, crs: String): Node
+getGMLEnvelopeFromPoint(doc: Document, point: Point, sourceCRS: CoordinateReferenceSystem, targetCRS: CoordinateReferenceSystem): Node

Figure A.3: The UML class diagram of the detailed mapper implementation design.

Appendix B

Prototype Source Code

```xml
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/
    Transform"
 xmlns:fn="http://www.w3.org/2005/xpath-functions"
 xmlns:functx="http://www.functx.com"
 xmlns:uuid="java.util.UUID">

<xsl:output method="text" omit-xml-declaration="yes" indent="no"/>
  <xsl:template match="/">
    <xsl:variable name="fMolOpen" as="xs:boolean" select="false()"/>
    <xsl:for-each select="//*">
      <xsl:choose>
        <xsl:when test="count(ancestor::*) mod 2">
        </xsl:when>
        <xsl:otherwise>
          <xsl:call-template name="parseNode">
            <xsl:with-param name="node" select="."/>
          </xsl:call-template>

        </xsl:otherwise>
      </xsl:choose>
    </xsl:for-each>
  </xsl:template>


<!-- generate AIXM feature id or access existing one -->
<xsl:template name="parseNode">
    <xsl:param name="node"/>

    <xsl:call-template name="get-node-id">
    <xsl:with-param name="node" select="."/>
  </xsl:call-template>
  :
```

```
<xsl:call-template name="get-node-name">
  <xsl:with-param name="node" select="$node"/>
</xsl:call-template>
[
<!-- transform direct child attribute-elements -->
<xsl:for-each select="$node/*">
  <xsl:value-of select="local-name(.)"/>

  <xsl:choose>

    <!-- References to other AIXM features of the attribute-element --
        >
    <xsl:when test="count(./*) &gt; 0">
    ->
      <xsl:for-each select="./*">
        <xsl:if test="count(../*) > 1">
          {
        </xsl:if>

        <xsl:call-template name="get-node-id">
          <xsl:with-param name="node" select="."/>
        </xsl:call-template>
        <!-- <xsl:call-template name="parseNode">
          <xsl:with-param name="node" select="."/>
        </xsl:call-template> -->

        <xsl:if test="position() != last()">,</xsl:if>

        <xsl:if test="(count(../*) > 1) and (position() = last())">
          }
        </xsl:if>
      </xsl:for-each>
    </xsl:when>

    <!-- Simple text content of the attribute-element -->
    <xsl:when test="./text() != ''">
      ->"<xsl:value-of select="./text()"/>"
    </xsl:when>

    <!-- Attributes of the attribute-element -->
    <xsl:when test="count(./@*) &gt; 0">
      <xsl:choose>
        <xsl:when test="count(./@*) = 1">
          (

          <xsl:call-template name="get-node-name">
            <xsl:with-param name="node" select="./@*[1]"/>
          </xsl:call-template>
          ) -> "<xsl:value-of select="./@*[1]"/>"
          </xsl:when>
          <xsl:otherwise>
            many attributes!
          </xsl:otherwise>
        </xsl:choose>
      </xsl:when>

    </xsl:choose>
    <xsl:if test="position() != last()">,</xsl:if>
  </xsl:for-each>
  ].
</xsl:template>

<!-- generate AIXM feature id or access existing one -->
<xsl:template name="get-node-id">
    <xsl:param name="node"/>
    <xsl:choose>
    <xsl:when test="$node/@*[local-name()='id'] != ''">
      <xsl:value-of select="$node/@*[local-name()='id']"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:variable name="random" select="uuid:randomUUID()"/>
              uuid_<xsl:call-template name="replace-string">
        <xsl:with-param name="text" select="$random"/>
        <xsl:with-param name="replace" select="'-'" />
        <xsl:with-param name="with" select="'_'"/>
      </xsl:call-template>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

<!-- retrieve the node name in OL notation with or without a namespace
    -->
<xsl:template name="get-node-name">
    <xsl:param name="node"/>
    <xsl:variable name="prefix">
    <xsl:call-template name="get-prefix">
      <xsl:with-param name="qname" select="name($node)"/>
    </xsl:call-template>
  </xsl:variable>

  <xsl:if test="$prefix != ''">
      <xsl:value-of select="$prefix"/>#
  </xsl:if>

  <xsl:value-of select ="local-name($node)"/>
</xsl:template>


```

```
<xsl:template name="replace-string">
   <xsl:param name="text"/>
   <xsl:param name="replace"/>
   <xsl:param name="with"/>
   <xsl:choose>
    <xsl:when test="contains($text,$replace)">
         <xsl:value-of select="substring-before($text,$replace)"/>
         <xsl:value-of select="$with"/>
         <xsl:call-template name="replace-string">
            <xsl:with-param name="text" select="substring-after($text,
                $replace)"/>
            <xsl:with-param name="replace" select="$replace"/>
            <xsl:with-param name="with" select="$with"/>
         </xsl:call-template>
      </xsl:when>
      <xsl:otherwise>
  <xsl:value-of select="$text"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>


<!-- get the substring before the *last* colon (or empty string if no
    colon) -->
<xsl:template name="get-prefix">
    <xsl:param name="qname"/>
    <xsl:param name="prefix" select="''"/>
    <xsl:choose>
        <xsl:when test="contains($qname, ':')">
            <xsl:call-template name="get-prefix">
                <xsl:with-param name="qname" select="substring-after(
                    $qname, ':')"/>
                <xsl:with-param name="prefix" select="concat($prefix,
                    substring-before($qname, ':'))"/>
            </xsl:call-template>
        </xsl:when>
        <xsl:otherwise>
            <xsl:value-of select="$prefix"/>
        </xsl:otherwise>
    </xsl:choose>
</xsl:template>
</xsl:stylesheet>
```

Listing B.1: XSLT prototype implementation.

```
(: http://docs.basex.org/wiki/Repository :)
import module namespace functx = 'http://www.functx.com';

declare namespace myNs="www.googlle.at";
declare namespace uuid = "java:java.util.UUID";

(: get current formatted timestamp :)
declare function myNs:getTimestamp() as xs:string {
  (: "yyyy-MM-dd'T'HH:mm:ss.SSS'Z'" :)
  concat(substring(replace(string(current-dateTime()),"\+","0"),1,23),"Z")
};

(: parse the text :)
declare function myNs:getParsedText($text as xs:string) as xs:string {
  replace(replace(replace(replace(functx:trim($text), "\\n", "<br>"), "\\r\\n", "<br>"), "&quot;", "\\&quot;"), "\\s+", " ")
};

(: retrieves id of given element. if no id attr is existing, an uuid is generated. :)
declare function myNs:getId($elem as element()) as xs:string {
  if(empty($elem/@*[local-name()="id"]))
  then replace(uuid:randomUUID(),"_","-")
  else replace($elem/@*[local-name()="id"],"_","-")
};

(: retrieves id of given element. if no id attr is existing, an uuid is generated. :)
declare function myNs:getId($id as xs:string, $elem as element()) as xs:string {
  if(string-length($id)>2)
  then replace($id,"_","-")
  else
    if(empty($elem/@*[local-name()="id"]))
    then replace(uuid:randomUUID(),"_","-")
    else replace($elem/@*[local-name()="id"],"_","-")
};

(: parses the element and the first two levels. :)
declare function myNs:parse($id as xs:string, $notam as element()) as element()* {

  let $res := (<fmol>{

    (: id term :)
    if($id = "")
    then myNs:getId($notam)
    else myNs:getId(replace($id,"->",""),$notam),

    (: ol class:)
    concat(":",replace(name($notam),":","#"),"["),

    (: attribute of xml ol object:)
    if($id = "")
    then <objattr>{"detectionTimestamp -&gt;",myNs:getTimestamp(),","}</objattr> union (
     for $attrobj in $notam/@*
      return <objattr>{replace(name($attrobj),":","#"),"-&gt;&quot;",data($attrobj),"&quot;,"}</objattr>)
    else for $attrobj in $notam/@*
     return <objattr>{replace(name($attrobj),":","#"),"-&gt;&quot;",data($attrobj),"&quot;,"}</objattr>
    ,

    (: children of the xml element representing associations :)
    for $attr in $notam/*
    return
     <attr>{

      (: ol attribute name :)
      replace(name($attr),":","#"),

      (: ol attribute can have own attributes, text content or refer to other ol objects:)
      if(count($attr/@*)>0)
      then
       (: output attributes of the xml association element:)
       for $attrattr in $attr/@*
        return <attrattr>{concat("(",name($attrattr),")","->",data($attrattr))}</attrattr>
      else
       if(count($attr/*)>0)
       then
        (: lvl 2 - referred ol objects:)
        for $subobj in $attr/*
         return <toParse><subobjid>{concat("->",myNs:getId($subobj))}</subobjid><subobjname>{$subobj}</subobjname></toParse>
       else
        (: text value of xml attribute elements:)
        concat("->&quot;",myNs:getParsedText($attr),"&quot;"),

      ","
     }</attr>,
   "].

"
}</fmol>)

  return if($id = "")
  then $res
```

```
  else
    let $subres := (
      for $subobj at $pos in $res//toParse return
         myNs:parse($subobj/subobjid,$subobj/subobjname/*)
    )
    return <a>{$res, $subres}</a>
};

(: define database existing in BaseX:)
let $db := 'NOTAM'

(: query all notams encoded as AIXMBasicMessages :)
let $notams := (for $y in doc($db)//* where local-name($y) = "
    AIXMBasicMessage" return $y)

(: query all namespaces:)
let $prefixes := (
  for $elem in doc($db)//* return
    for $prefix in fn:in-scope-prefixes($elem)
      return <ns><prefix>{":- prefix ",$prefix,"="}</prefix><uri>{"&quot
          ;",fn:namespace-uri-for-prefix($prefix,$elem),"&quot;.
"}</uri></ns>
 )
let $distincPrefixes := fn:distinct-values($prefixes)

(: parse the first level :)
let $parsedNotams := (
    for $notam in $notams
      return myNs:parse("",$notam)
  )

(: retrieve all f-molecules of the child nodes:)
let $subFMols := (
for $subobj at $pos in $parsedNotams//toParse
  return myNs:parse($subobj/subobjid,$subobj/subobjname/*)
)

(: remove elements which where needed only for the recursion :)
let $cleanedParserNotams := (functx:remove-elements-deep($parsedNotams
    ,'subobjname'))
let $cleanedSubFMol := (functx:remove-elements-deep($subFMols,'
    subobjname'))

(: join the sequences to stirngs :)
let $output0 := string-join($distincPrefixes)
let $output1 := string-join($cleanedParserNotams)
let $output2 := string-join($cleanedSubFMol)
let $output := concat($output0,$output1,$output2)
let $output := replace($output,",\&#x005D;.","&#x005D;.")
```

```
return $output
```

Listing B.2: XQuery prototype implementation.

```
public class DOMParser {

  private OLProgram olProgram;

    public DOMParser() {
      olProgram = new OLProgram("olProgram");
    }

  public OLProgram startParser(String file) {

    DocumentBuilder builder;
      DocumentBuilderFactory factory = DocumentBuilderFactory.
          newInstance();
        factory.setNamespaceAware(true);
    Document doc = null;

    try {
      builder = factory.newDocumentBuilder();

      // create xml document in memory
      doc = builder.parse(new File(file));

    } catch (ParserConfigurationException | SAXException | IOException
        e) {
          e.printStackTrace();
    }

    // select root element. can be query, result, FeatureCollection
    Element root = doc.getDocumentElement();

        OLFMolecule parentFMol = null;

        parseNode(doc, parentFMol, root, 0, getNodeName(root));

    return olProgram;
  }

  /**
   * Recursive method which iterates top-down through all elements of
       an given node.
   * Creates and adds new f-molecules to the OL program.
   * @param parentFMol
   * @param node
   */
```

```java
private void parseNode(Document doc, OLFMolecule parentFMol, Node
    node, int level, String parent) {

  if(node.getNodeType() == Node.ELEMENT_NODE) {
    // only element nodes can be an object- or attribute-element

    if(!parent.equals(getNodeName(node.getParentNode()))) {
      level++;
      parent = getNodeName(node.getParentNode());
    }

    if(!(level%2==0)) {
      // object-element

      OLFMolecule fMol = null;

      fMol = initFMolecule(node);

      fMol.setAttributes(getNodeAttr(olProgram,node, fMol.
          getAttributes()));

      // add a detection timestamp to Notams aka AXIMBasicMessage
      String timestamp ="";
      if(node.getLocalName() != null && node.getLocalName().equals("
          AIXMBasicMessage") || node.getLocalName() != null && node.
          getLocalName().equals("Query")) {

        try{
          SimpleDateFormat formatter = new SimpleDateFormat("yyyy-MM-
              dd'T'HH:mm:ss.SSS'Z'");
              Date currentTime = new Date();
              timestamp = formatter.format(currentTime);
        }
        catch(Exception e){
          e.printStackTrace();
        }

        fMol.addFMolAttr("detectionTimestamp", new OLLiteral("
            timestamp", timestamp));
      }

      // parse child nodes - start recursion
      if(node.hasChildNodes()) {
        for(int i=0; i<node.getChildNodes().getLength();i++) {
          parseNode(doc, fMol, node.getChildNodes().item(i), level,
              parent);
        }
      }

      olProgram.addMolecule(fMol);
    }
    else if(parentFMol != null && !isEmptyAttrElem(node)) {
      // attribute-element

      // add node attributes to f-molecule attributes
      parentFMol.setAttributes(getNodeAttr(olProgram, node, parentFMol.getAttributes()));

      // handle html as text (formattedText), the child elements including html are not correctly parsed (multiple parsed)
      if(getNodeName(node).equals("formattedText")) {
        addFormattedText(olProgram, parentFMol, node);
        Element parentNode = (Element) node.getParentNode();
        parentNode.removeChild(node);
      }

      // second level
      for(int i=0; i<node.getChildNodes().getLength(); i++) {

        Node childNode = node.getChildNodes().item(i);

        // objects are added as attr -> obj

        if(childNode.getNodeType() == Node.ELEMENT_NODE) {

          if(!parent.equals(getNodeName(childNode.getParentNode()))) {
            level++;
            parent = getNodeName(childNode.getParentNode());
          }

          if(!(level%2 == 0)) {

            if(!isIdExisting(childNode)) {

              // add 'id' directly to the node which is then parsed correctly
              String id = ("uuid_".concat(UUID.randomUUID().toString())).replace("-", "_");
              ((Element)childNode).setAttribute("id", id);
              parentFMol.addFMolAttr((node.getPrefix() == null ? "" : node.getPrefix().concat(":")).concat(getNodeName(node)), new OLLiteral("id",id));
            }
            else {
              parentFMol.addFMolAttr((node.getPrefix() == null ? "" : node.getPrefix().concat(":")).concat(getNodeName(node)), getNodeId(childNode));
            }
```

```

                    // recursion starts here for new object
                    parseNode(doc, null, childNode,level, parent);
                  }
                }
                else if(childNode.getNodeType() == Node.TEXT_NODE) {
                  // Simple Elements - Text within elements and no children ->
                       getNodeValues
                  String text = getParsedString(childNode.getNodeValue());

                  if(text.length() > 0) {
                    parentFMol.addFMolAttr((node.getPrefix() == null ? "" :
                        node.getPrefix().concat(":")).concat(getNodeName(node).
                        replace(':','#')), new OLLiteral(getNodeName(node).
                        replace(':','#'),text));
                  }
                }
              }
            }
          }
        }

        ...
      }
```

Listing B.3: DOM XML prototype implementation.