Doctoral Thesis

Model-driven tools to support conceptual geospatial modelling

Author(s):
Schulze Althoff, Jan

Publication Date:
2011

Permanent Link:
https://doi.org/10.3929/ethz-a-006740897

Rights / License:
In Copyright - Non-Commercial Use Permitted
MODEL-DRIVEN TOOLS TO SUPPORT CONCEPTUAL GEOSPATIAL MODELLING

A dissertation submitted to
ETH ZURICH
for the degree of
DOCTOR OF SCIENCES

Presented by
JAN SCHULZE ALTHOFF

Dipl. Forstwirt, Universität Freiburg i. Brsg.
born 25 May 1971
citizen of Germany

Accepted upon the recommendation of
Examiner: Prof. Dr. Lorenz Hurni, ETH Zürich
Co-Examiner: Prof. Dr. Barbara Koch, Universität Freiburg i. Brsg.
Co-Examiner: Dr. Christine Giger

2011
CONTENTS

3.4.1 Transforming between Domain-Specific Language HML and UML ............................................ 93
3.4.2 Results ................................................................................. 100
3.5 Vertical Transformations .......................................................... 100
3.5.1 Transforming from Domain-Specific Language HML to GML Schema ........................................... 101
3.5.2 Results ................................................................................. 106
3.6 Summary ................................................................................. 106

4 Implementation ........................................................................ 109
4.1 Used Technology ...................................................................... 110
4.1.1 Java .................................................................................. 110
4.1.2 Eclipse .............................................................................. 111
4.1.3 Eclipse Modelling Framework (EMF) ................................. 111
4.1.4 Graphical Modelling Framework (GMF) ....................... 112
4.1.5 OpenArchitectureWare .................................................. 115
4.2 Domain Specific Language & Graphical Editor ....................... 116
4.2.1 Abstract Language Definition ......................................... 116
4.2.2 Generation of Core Code Fragments ................................ 116
4.2.3 Definition of Graphical Notation .................................... 117
4.2.4 Model Validation .............................................................. 122
4.2.5 Generation of a Graphical Model Editor ....................... 123
4.2.6 Description of the Graphical Model Editor .................... 123
4.2.7 Results ................................................................................. 127
4.3 Language-to-Language Transformations .................................. 128
4.3.1 Environment and Runtime .............................................. 128
4.3.2 Definition of the Transformation .................................... 128
4.3.3 Execution of Transformation .......................................... 137
4.3.4 Results ................................................................................. 137
4.4 Realisation of Vertical Transformations .................................. 138
4.4.1 Transforming between Application Specific Language HML and GML Schema ......................... 138
4.4.2 Generation of a HTML Documentation from HML Conceptual Models ......................................... 144
4.4.3 Results ................................................................................. 147
4.5 Summary ................................................................................. 148

5 Summary of the Results ............................................................. 151

6 Discussion ................................................................................. 155
6.1 Definition of Domain Specific Languages .............................. 157
6.2 Tools .................................................................................. 158
6.3 Horizontal Language-to-Language Transformations .............. 159
6.4 Outlook .................................................................................. 160
Abstract

The raising importance and fast development of national and international Spatial Data Infrastructures originate a need for harmonised geo data of different sources. Spatial datasets should not only be found, presented and processed via geo portals, but different information should also be capable of being integrated. Beyond pure technical and geometric issues like file formats, spatial resolutions or reference systems, this also includes informational aspects as data structures and semantic aspects.

A precondition for a tight integration of spatial data from different sources are precise definition of the used or targeted datasets. Such definition processes currently take place in the ‘INSPIRE data specifications’ or the Swiss efforts to create ‘minimum data models’ according to the Swiss federal act on geoinformation. In those initiatives groups of experts discuss about common information elements, how those elements are characterised in detail and relate to each other. The resulting data description is formalised in so called ‘conceptual data models’, which should be as independent of data formats, database systems or general computational aspects as possible.

Data models in the field of spatial information are characterised by specific aspects that are not common in other domains. Those are especially location with reference systems, different kinds of 2d or 3d geometries or measurements with units and value ranges. With common ‘general purpose’ modelling approaches like UML it is difficult to represent those aspects in a sufficient and user friendly way. Consequently, in recent years several specialised approaches were developed to support the creation of spatial models. There are extensions to UML, like GeoUML or the ISO/TC-211 standards, as well as independent languages like INTERLIS or MADS. Drawbacks of all specialisations are mainly a lack of suitable tools for modelling, limitations in scope or adaptability and poor interoperability between different approaches.

Such issues are addressed by new trends in the field of model driven software developments. After monolithic and inflexible approaches, new ‘language-centric’ methods evolve to support the adaptability of modelling languages, the creation of suitable tools and transformations between different types of models. A core technology for this is ‘meta modelling’ and describes the formal definition of modelling languages with standardised el-
Abstract

Meta models of modelling languages, which is used to efficiently generate software tools like graphical or textual editors, model validators or model transformations. This approach is highly efficient and not only used for existing languages, but also induced the creation of highly specialised ‘small’ (or ‘domain-specific’) languages, which are tightly adjusted to a field of work, a specific task or a user group.

This thesis explores to what extend those ideas of model driven software development can be applied to conceptual spatial data modelling. This could help to simplify modelling for domains expert by providing them suitable languages and graphical editors, enhance model quality with model validators and offer new ways of interoperability between different modelling languages.

Firstly it will be tested, if a common meta modelling approaches of software development can be used to sufficiently describe languages for conceptual spatial modelling. This is carried out for existing conceptual geospatial languages with Ecore, the most commonly used meta modelling approach. As result, the different languages can be represented on a common definition base. Additionally to already existing modelling languages, the idea of ‘domain specific languages’ is taken up to develop an exemplary modelling language ‘HML’ for the HUMBOLDT project. According tools like a graphical model editor and a validating parser are created in a highly automated way for this language.

Based on the developed meta models, transformations between different conceptual spatial modelling approaches are investigated in a second step. A language-to-language mapping between the domain specific approach of HML and ISO/TC-211 compliant UML is declared and executed. Such transformations between specialised languages and general purpose languages are of particular interest, because they offer ways of optimising modelling processes while staying interoperable to existing models and tools.

Finally vertical transformations, the possibility to create logical or physical models from the conceptual models, are addressed. Similar to the horizontal language-to-language transformations, standard tools from software development are used to derive GML Schema and textual model documentation.

All in all, this work proves along practical implementations how language based technologies of modern model driven software development can support conceptual spatial modelling. Especially the combination of ‘domain-specific’ modelling languages with according software tools and horizontal language-to-language tools offers possibilities to simplify modelling processes, increases model quality and opens a new field of interoperability between different conceptual modelling languages.

\[1^1\text{The HUMBOLDT project (EC contract SIP5-CT-2006-030962) of the European Commission addresses harmonisation issues in the context of INSPIRE and GMES} \]
Zusammenfassung


Eine Grundvoraussetzung zur Harmonisierung von Geodaten ist dabei die Definition der Dateninhalte, wie sie zur Zeit bei den 'INSPIRE Data Specifications' innerhalb der Europäischen Union oder den 'Minimalen Datenmodellen' der Schweiz stattfindet. In Expertengruppen werden die Elemente definiert, welche in den gemeinsamen Datensätzen verwendet werden. Es wird beschrieben, was sie charakterisiert und wie verschiedene Elemente zusammenhängen. Solche Beschreibungen werden als sogenannten 'konzeptionelle Modelle' erarbeitet und sollen möglichst frei von technischen Details wie Datenbanken oder Austauschformate sein.


Zusammenfassung

einfach verschiedene Werkzeuge wie Modelleditoren oder Modell-Checker erzeugen und effiziente Modelltransformationen definieren. Dieser Ansatz ist so effizient, dass neben bekannten Sprachen auch zunehmend hochspezialisierte 'kleine Sprachen' kreiert werden. Diese 'Small Languages' oder 'Domain Specific Languages' werden genau auf ein Problemfeld oder eine Anwendergruppe zugeschnitten.

In dieser Arbeit wird untersucht, inwiefern sich diese Ideen der modellgetriebenen Softwareentwicklung mit Ansätzen der konzeptionellen, räumlichen Datenmodellierung verbinden lassen. Das Ziel ist dabei, die Entwicklung angepasster Modellierungs- sowie Transformationswerkzeuge zu verbessern und eine Durchlässigkeit zwischen verschiedenen Verfahren zu fördern.

Zunächst wird geprüft, ob sich die sprachzentrierten Ansätze der modellgetriebenen Softwareentwicklung überhaupt auf diesen Bereich der Datenmodellierung anwenden lassen. Dazu werden bestehende räumliche konzeptionelle Sprachen mit 'Ecore', der meistbenutzten Meta-Modellierungssprache im Bereich der Softwareentwicklung, abgebildet. Zusätzlich wird die Idee der 'Domain Specific Languages' verfolgt und eine Modellierungssprache entwickelt, welche möglichst gut auf die Anforderungen des Europäischen Forschungsprojektes HUMBOLDT2 abgestimmt ist. Für diese Sprache, die 'Humboldt Modelling Language HML', werden auch entsprechende Werkzeuge wie ein prüfender Parser und ein graphischer Modelleditor erstellt.


Der letzte untersuchte Aspekt befasst sich mit sogenannten vertikalen Transformationen, also den Möglichkeiten, aus konzeptionellen Datenmodellen logische Modelle oder Datenstrukturen und -formate zu erzeugen. Auch hier werden die Transformationsverfahren der modellgetriebenen Softwareentwicklung genutzt um aus Modellen in der 'Humboldt Modelling Language' GML-Schemata und textuelle Modellbeschreibungen abzuleiten.

Zusammenfassend wird in der Arbeit anhand von konkreten Implementierungen gezeigt, wie sprachbasierte Ansätze der modellgetriebenen Softwareentwicklung auch im Bereich der konzeptionellen räumlichen Modellierung effizient genutzt werden können. Besonders die Kombination aus domänen-

---

2'HUMBOLDT' ist ein Europäisches Projekt über Daten- und Dienstharmesierung im Kontext von INSPIRE und GMES.
Zusammenfassung

spezifischen Sprachen mit optimierten Modellierungswerkzeugen und horizontalen Modelltransformationen zwischen verschiedenen Sprachen eröffnet neue Möglichkeiten. Die konzeptionellen Modellierungsverfahren können besser an die Bedürfnisse des Endanwenders angepasst bzw. vereinfacht werden ohne die Interoperabilität zu existierenden Verfahren oder bestehenden Modellen einzubüßen.
Chapter 1

Introduction

In the last few years, the development of Spatial Data Infrastructures (SDIs) significantly advanced. Currently there are several local, regional, national and international SDIs available or under development. They technically evolve from pure catalogue and simple data viewers toward a more service-oriented infrastructure with distributed data storage and processing. The technical possibilities together with a political wish for cross-border data brought up a new quality in the combination of spatial datasets. The necessity for up-to-date and integrated data sets from different sources is especially driven by topics like air and water pollution, traffic management, statistics or natural hazards, which can only be managed across national borders.

The INSPIRE (infrastructure for spatial information in Europe) initiative [Joint Research Centre, 2011] of the European Commission addresses accessibility and integration of transnational environmental information in Europe. This shall be achieved by setting up a legal framework, defining interoperability arrangements between member states and EC as well as establishing a common geoportal. Similar efforts on national scale take also place in Switzerland with the new federal act on geoinformation [Bundesversammlung der Schweizerischen Eidgenossenschaft, 2007] and the creation of a new national portal.

The wish for an easy and automated combination of data from different sources causes several problems (see e.g. [INSPIRE Drafting Team "Data Specifications", 2007]). Beside technical challenges like different databases and GIS formats or multiple spatial reference systems, there are also content-related problems. Depending on the responsible organisations and intended use, there are huge differences in the underlying information models. For example, one data provider might be responsible for the cartographic representation of streetdata whereas in another country this data is managed by the road authorities themselves. In the first case, only the map representation is of interest, whereas in the second case, all technical information on road width, surface or speed limitation is managed. The integration of the
1.1 Motivation and Current Situation

Against the background of those efforts, surprisingly little work has been spent during the last years in technology support for geospatial conceptual modelling. This is in contrast to major achievements in the field of software development.

In software development, there is a current strong trend towards simplifications and a new flexibility of modelling approaches.

Similar to new simplified coding languages also fresh, adjusted modelling approaches like SysML [OMG - Object Management Group, 2010] or BPMN

\footnote{In sum there are 34 INSPIRE data themes like `Transport Networks', `Hydrography', `Elevation', `Buildings', `Soil' etc. defined.}
1.2. HYPOTHESIS

[OMG - Object Management Group, 2011] and numerous small, highly-specialised languages with adapted modelling tools appear. Even the Object Management Group (OMG), the main driver of standardisation in the field of model-driven software development and maintainer of the Unified Modelling Language (UML), incorporate those new concepts in their strategies [Watson, 2008]. One key technology is called ‘Meta Modelling’ and describes the formalisation of modelling languages. In brief, meta models describe modelling languages and can be used to automatically derive tools for editing, validation and transformations with those languages. This is especially used for adapted conceptual modelling languages, so called ‘domain-specific languages’. They help experts during the modelling process by offering small and well-fitting concepts with a user-friendly terminology.

In geospatial modelling the trend seems contrary. After strong research activity on design of geospatial modelling languages in the late 1990s and early 2000s (see e.g. [Friis-Christensen et al., 2001], [Belussi et al., 2006] or [Bédard et al., 2004]), most modelling approaches disappeared with the finalisation of their research projects.

Beside the Swiss standard INTERLIS 2 is utilised in Swiss administration and offers some tool support. In addition, mainly the standards of the ‘ISO Technical Committee on Geographic Information/Geomatics (ISO/TC-211)’ [ISO/TC-211, 2002] are in practical use. The latter are based on UML, but add several rules and use large predefined models. The modelling support is limited to the standard UML editors, so that the complexity of the approach is still exposed to the domain experts. Model Validation is neglected and it is not possible to switch between the different modelling languages.

As consequence of that situation, this thesis tries to examine the potential of new model-driven technologies for the conceptual geospatial modelling. Especially ‘Meta Modelling’, the way of formalising languages in a standardised way is in the focus of this work. These techniques are not regarded to replace current modelling approaches, but to complement them with the potential to offer adjusted languages, suitable tools and to allow switches between different modelling languages.

1.2 Hypothesis

The usage of the new techniques from model-driven software development, especially ‘Meta Modelling’ will improve and simplify conceptual geospatial modelling.

As shortly touched in the introduction, meta modelling offers a basic foundation for different modelling languages and can therefore act as a kind of ‘glue’ between existing and new approaches. The optimal processability of languages based on formal meta models supports the creation of well-fitting languages, the development of suitable tools like model editors or valida-
tors, and enforces interoperability by language-to-language transformations. Therefore these subtopics will be examined in this work:

- **Meta models for existing conceptual geospatial modelling approaches.** Is it possible to redefine existing modelling languages like INTERLIS 2 with the help of standard meta modelling languages? This option would allow to combine already existing technology of spatial modelling with tools of model-driven software development.

- **Creation of domain-specific modelling languages for conceptual geospatial modelling.** Can a new spatial modelling approach be defined based on a set of requirements? Adjusted languages, either as simple limitation of existing languages, as ‘best-of-breed’ or complete new development, could offer modelling experts an optimised approach to formulate their informational needs.

- **Creation of a domain-specific modelling tool.** Can user tools be directly derived from a domain specific geospatial modelling language? User tools based on an automated process of current model-driven software development can be developed with little effort and high quality. This would be of special interest for small developer communities, like the geospatial.

- **Horizontal language-to-language transformations.** Is it feasible to define and execute transformations between different conceptual geospatial modelling approaches? Transformations between different modelling approaches would increase interoperability, so that e.g. an INTERLIS 2 model could be compared with UML models or a model in a highly specialised language could be processed with transformation tools of the ISO/TC-211 approach.

- **Vertical transformations from conceptual models to logical or physical models.** Can standardised model transformations from the field of software development be used to simplify the generation of data formats? Data formats are a prominent target for geospatial modelling and any simplifications in the transformations would be an asset.

### 1.3 Structure of the Thesis

This work is composed of six main chapters. In Chapter 1 a short introduction is given to guide quickly to the field of conceptual geospatial modelling, with its problems and the formulated working thesis.

Chapter 2 refers to the actual state of the art. The subchapters 2.1, 2.2 and 2.3 describe used approaches and technology for conceptual geospatial modelling, whereas subchapter 2.4 introduces terminology and standards of model-driven software development.
1.4. SUMMARY

After the formulation of the working thesis, chapter 3 explains the main ideas of this work. In this chapter the connecting elements, the metamodels of geospatial languages are created and the mapping concepts for horizontal language to language and vertical transformations are worked out and prepared for the implementation.

The used technologies and prototypical implementations are content of chapter 4. The implementation includes the creation of a domain-specific modelling tool, based on a model-driven development approach, the language to language transformation to link the domain specific language with mainstream UML modelling and finally vertical transformations to derive GML Schema and textual documentation of the conceptual models.

Chapter 5 comprises the results and in the final discussion of chapter 6, the used approach is reviewed and an outlook on potential usage is given.

1.4 Summary

This introduction points to the current needs for conceptual geospatial modelling, which are mainly driven by different SDI initiatives like INSPIRE or the Swiss national geportal. Despite to huge efforts in this field, little technical support is given to the modelling experts.

This is in contrast to the trends in model driven software development, where new technologies tend to support the modelling process strongly. One key technology for this is ‘meta modelling’.

In this thesis, the potential use of meta modelling techniques for conceptual geospatial modelling is examined. On one hand the theoretical base is tested, by setting up meta models of geospatial languages and on the other hand, according tools and transformations are implemented as prototypes.
Chapter 2

State of the Art

As stated in the introduction, this thesis tries to build bridges between new technologies and approaches in the field of model-driven software development on one hand and the current practices and needs for geospatial data modelling on the other hand. These two domains address quite different users; so terminologies, used techniques and expectations towards the approaches differ strongly. Beside the description of the actual state of the art for geospatial modelling approaches and model-driven software technologies, this chapter also tries to point out the particular strategies and terminologies. This helps in recognising the differences, the different intentions as well as commonalities or overlaps.

2.1 Modelling and Models

'Models can perform two fundamentally different representational functions. On the one hand, a model can be a representation of a selected part of the world (the 'target system'). Depending on the nature of the target, such models are either models of phenomena or models of data. On the other hand, a model can represent a theory in the sense that it interprets the laws and axioms of that theory. These two notions are not mutually exclusive as scientific models can be representations in both senses at the same time.' [Stanford University, 2010]

In this work only a small aspect of this very general model definition is regarded; the representation of geospatial phenomena in information systems, e.g. geospatial databases, interfaces for the exchange of spatial information or GIS information models.
2.1.1 General Data Modelling

For modelling of data for information systems, the definition is further refined by Simsion [Simsion, 2007]. Modelling as act of creating models can either be perceived as:

‘... a descriptive activity, the objective of which is to document some aspects of the real world ...’

or as

‘... a design activity, the objective of which is to create data structures to meet a set of requirements ...’

The difference is mainly philosophical and dependent on the authors’ interpretation: either the descriptive tasks [Halpin and Morgan, 2008] or more requirement-oriented approaches [Ambler and Sadalage, 2006] are emphasised.

Descriptive approaches focus on methods to transfer the ‘Universe of Discourse (UoD)’ [ISO/TC-211, 2002] to a formal model. The Universe of Discourse is the perceived subset of reality, which is defined by experts of the thematic field. The modelling process acts as a bridge between the individual (or group) perception of reality for a specific task and the information system.

Engineering-focussed approaches develop a formal model from requirements towards the demanded information system. The discipline of ‘requirement engineering’ is originated in modern software development (see e.g. [Sommerville, 2007], [Pagel and Six, 1994]). The concrete character of the targeted information systems can vary from data centric software applications over information models for complete enterprise businesses up to standardised data exchanges (see [Moody, 2005]).

Data models link views on reality with computer systems. The grade of abstraction of models indicate how ‘human understandable’ or how ‘computer processable’ they are. Already in the 1970s different abstraction levels for data models were standardised by the American National Standards Institute [American National Standards Institute, 1975]. Although nowadays it is good practice to use more and gradually evolving models ranging from simple concepts to concrete software implementations (see [Proper, 1997]), three types of models are usually differentiated: the conceptual models, the logical models and the physical models (figure 2.1).

Conceptual Models

The conceptual models describe the requirements of the end-users towards the data of an information system or, following the descriptive approach, the user-specific Universe of Discourse. Conceptual models should only reflect the needs towards the information and not consider any additional elements of relevance for practical implementation.
2.1. MODELLING AND MODELS

Figure 2.1: Differentiation of models into physical model, logical model and conceptual model

... conceptual schemas should deal only and exclusively with aspects of the Universe of Discourse (UoD). Any aspects irrelevant to that meaning, e.g. machine efficiency, should be avoided. [ter Hofstede and van der Weide, 1993]

Despite several guidelines on database modelling, conceptual models are not limited to a certain grade of completeness (e.g. it is often stated that conceptual models should only deal with entities or classes, but not contain further characteristics or attributes). In fact, the conceptual models should be used to obtain a complete reflection of the end-users’ knowledge in a formalised way. This includes important elements like required numerical precision or value ranges and spatial properties. To create such complete models, cyclic refinement processes are often used, where coarse models, containing just the basic information groups, are stepwise elaborated to be completed (e.g. [Balzert, 1995]).

For the creation of conceptual data models a variety of different approaches with specific ideas and rules were developed in the past 40 years. The most prominent representatives are:

- Entity Relationship Approaches (see [Chen, 1975]) describe the UoD by extracting concrete objects ('entities') and the references between them ('relationships'). Similar entities and relationships can be summed up to 'entity types' or 'relationship types'. Entity types are detailed by their attributes. Notations for entity relationship models are mostly graphical like the original notation by Chen [Chen, 1975] or IDEF1X [National Institute of Standards and Technology, 1993].

- Object Role Models (see [Halpin and Morgan, 2008]) follow also the concept of entities and entity types, but resolve all properties by using
‘roles’ and entities. This means that what is described as attribute in an entity relationship model appears as own (partially artificial) entity type connected by roles. An important notation for object role models is ORM2 (see [Halpin, 2005])

- Object-Oriented Modelling (see [Booch et al., 2004]) for static data is quite close to entity relationship modelling. Similar objects are grouped to classes and relationships can be defined between them. The additional concepts of inheritance and polymorphism allow simplifications in complex models. The best known notation is UML (see [OMG - Object Management Group, 2007b,d] and chapter 2.2), but there are also other languages like EXPRESS [ISO/TC-184, 2004] for the CAD/CAM domain or INTERLIS 2 [KOGIS, 2006b], which is tailored to geospatial applications.

It is remarkable, that the choice of the modelling approach and the expressiveness of the selected approaches and modelling languages directly influence the perception of the UoD and can result in strongly different information models [ter Hofstede and van der Weide, 1993].

**Logical Models**

Logical data models describe the conceptual model from an information system-related perspective. They already reflect key issues of the underlying software systems and aim to support the technical implementation as good as possible [Parent et al., 2006a]. Typically, the logical models contain system-specific data types, mechanisms to solve relations (like foreign keys or reference types) or resolved inheritances and complex associations.

The differentiation between conceptual and logical models can be complicate, because the same modelling language might be used for conceptual and logical models (esp. UML). Quite often cyclic refinements in modelling lead to gradually changing models from conceptual to logical models [Proper, 1997].

**Physical Models**

The physical models are intended to be the real implementation of the information system. Those models include concrete database schemas for specific database management systems, proprietary file formats for data transfer or code in programming languages. The physical models directly depend on the used hard- and software and must be adopted for every change in the infrastructure.

Typical languages for physical models are SQL, XML-Schemas, Mapping-Schemas like Hibernate Mappings [JBoss Community, 2011] or program code. Again, a differentiation between logical and physical models is sometimes
2.1. MODELLING AND MODELS

difficult; e.g. XML-Schema is a direct format description, but can also be regarded as tree-based logical model.

2.1.2 Geospatial Modelling

Geospatial Modelling is a specialisation of the general modelling processes. It always deals with at least two modelling aspects; aspects of location, shape and size in the real world\(^1\) and the thematic content [Abdul-Rahman and Pilouk, 2008]. The spatial aspects include the correct description of the earth's surface, the localisation on it, the description of metrics and the topology to describe spatial relations between objects.

Conceptual Modelling Approaches for geospatial data must offer ways to describe the concepts of thematic information and space in an appropriate level. The needs for modelling possibilities of space can vary strongly. Whereas several information systems only need simple support for spatial data types, cartographic applications or GIS analysis may need highly precise definition of space and topologies.

Traditionally, there are two main approaches for the perception of reality, either as continuous space, where all locations have specific properties, called field- or raster-based, or as an empty space where objects are placed at specific locations with specific extend, the feature- or vector-based approaches [Goodchild, 1992].

Field-Based

The complete space is divided in (irregular or regular) non-overlapping cells and the cells carry thematic information. The cells act as link between space and attributes. This approach is used for phenomena being continuous in space (e.g. temperature or radiation) or if object borders are difficult to determine (e.g. emission clouds or oil spills). Field-based GIS mostly store the information in flat tables.

Object- or Feature-Based

Feature-based approaches consider objects having specific characteristics including its location and size in an empty space [Goodchild et al., 2007]. Depending on the user perspective, the feature-based approach can be subdivided in more geospatial-centric or attribute-centric approaches. The first approach 'attaches' properties to a geometric element, while the other one 'attaches' geometry as a property to the object\(^2\). The feature-based approach

\(^1\)Other aspects like time, temporal changes, versioning etc. often also play important roles in geospatial modelling

\(^2\)In practise both approaches are difficult to differentiate. Objects or features are often distinguished by a combined definition of spatial and thematic aspects. E.g forest stands may need a minimum area and a specific thematic content like predominant species
is used in vector geographical information systems.

2.2 Conceptual Modelling Languages

As stated in chapter 2.1.1, the languages for conceptual modelling are the most important tools for the description of the universe of discourse or the requirements towards the information system. They must offer the modellers the possibilities to define their knowledge as precise and simple as possible. On the other hand the produced models must be understandable for developers or even be machine-readable as input for further software development.

Three established languages for conceptual modelling of geospatial data will be presented in this work to illustrate the actual state of the art. The approaches are chosen from a variety of geospatial modelling approaches (e.g. [Belussi et al., 2004], [Bédard et al., 2004] or [Borges et al., 2001]) as prominent representatives: the Unified Modelling Language UML, the Swiss standard INTERLIS 2 and the MADS approach for modelling complex spatial databases.

2.2.1 Unified Modeling Language

The Unified modelling Language (UML) is a graphical modelling language for software engineering based on object oriented concepts.

The objective of UML is to provide system architects, software engineers, and software developers with tools for analysis, design, and implementation of software based systems as well as for modelling business and similar processes. [OMG - Object Management Group, 2007b]

UML was created as combination of three competitive modelling approaches in the 1990s; the Object Modelling Technique (OMT) of James Rumbaugh, the Object Oriented Software Engineering (OOSE) from Ivar Jacobsen and the modelling notations of Grady Booch. In 1997 a first official release was published by an industry consortium ‘Object Management Group (OMG)’ and later also adopted by the International Standardisation Organisation [ISO JTC 1/SC 7, 2005].

With release of UML 2 in 2005 major changes took place with the aim to fix inconsistencies, include user feedback and to align the specification with the other standardisation approaches by OMG, esp. the Model Driven Architecture [OMG - Object Management Group, 2009]. Beside the drastic extensions of language size the specification formalism was strongly enhanced. The language specification is now completely built on a meta model

\*\*The complete specification of Infrastructure and Superstructure contains 13 diagram types for requirement analysis, static and dynamic models, collaboration and deployment and is over 900 pages\*\*
2.2. CONCEPTUAL MODELLING LANGUAGES

approach (see chapter 2.4) and language extensions by using ‘profiles’ are redesigned.

Specification Structure

The UML specification is split up in a series of specification documents.

The ‘UML Infrastructure’ [OMG - Object Management Group, 2007b] defines the language basics. That includes first the definition of basic modelling constructs, which were reused for the own specification and for other OMG specifications, secondly the modelling elements for the static conceptual models and finally the mechanisms for extending the language specification, esp. profiling (chapter 2.2.1).

The ‘UML Superstructure’ [OMG - Object Management Group, 2007d] defines all the other modelling elements, including the more specialised elements like collaboration or activity diagrams.

The ‘Object Constraint Language (OCL)’ [OMG - Object Management Group, 2006c] should be treated as essential member of the UML specification, as it allows to restrict elements of UML models to better fit the modelling requirements. OCL is defined with a textual notation.

The ‘Meta Object Facility 2.0 XMI Mapping Specification’ [OMG - Object Management Group, 2007a] describes the approach to store models in an XML format. The format specification is not bound to UML, but to the ‘Meta Object Facility (MOF)’ [OMG - Object Management Group, 2006b].

The ‘Diagram Interchange’ [OMG - Object Management Group, 2006a] specification of the Object Management Group is in an early state and allows the exchange of graphical model representations and the mapping to scalable vector graphics (SVG).

For conceptual data modelling, mainly the ‘UML Infrastructure’ as basics for static modelling and the ‘Object Constraint Language (OCL)’ for the necessary refinement of data models are of interest. The other specifications are usually beyond the scope of data modelling or are included in the used modelling software.

UML Compliance

UML2 is a huge and complex specification. For that reason OMG splits the specification into packages and defines subsets to control complexity. Based on those subsets four compliance levels are defined, ranging from basic static language elements at level 0 to the complete specification with enhanced features like templates at level 3. For conceptual spatial data modelling mainly the levels 0 and 2 are of relevance.

Level 0 is sufficient for standard conceptual data modelling level 0 and includes all elements for static object-oriented modelling (mainly the creation of ‘Class Diagrams’). Compliance level 2 is of much higher complexity
and includes especially the usage of UML profiles. This level is needed for geospatial modelling according to ISO/TC-211 (see 2.2.1).

For each compliance level three qualifiers are defined:

- Compliance to Abstract Syntax: The modelling elements of the level are available in the modelling environment and can be stored according to model exchange format.

- Compliance to Concrete Syntax: Additionally to the compliance of the abstract syntax is the notation according to the UML specification.

- Compliance to Diagram Format Specification: The graphical notation can additionally be stored and restored following the Diagram Interchange Specification [OMG - Object Management Group, 2006a]

**UML Class Models**

In UML 2.13 different diagram types are defined, classified into ‘structure diagrams’, ‘behaviour diagrams’ and ‘interaction diagrams’. For conceptual data modelling the structural diagram type ‘Class Diagram’ is of major interest. Class diagrams give a static view of the used information elements defined for the system. The following elements taken from the UML ‘core’ definition [OMG - Object Management Group, 2007b] are mainly used:

- **Type**: Types are named elements that can have properties and relations to other Types. Types cannot be used directly as they are defined ‘abstract’. Only the inherited modelling elements like ‘Class’ or ‘Data Type’ are used directly and inherit the properties of the ‘Type’ definition.

- **Class**: Classes are extensions of Types that are used to model groups of objects with similar properties. The key characteristics of Classes are their object-identities, i.e. every object of the class can be addressed by its own, and the encapsulated state of the objects, i.e. the objects carry the information (For details of those key concepts of object-orientation see e.g. [Balzert, 1995]). The notation for a Class is a rectangle with one compartment at the top for the name and at least one more compartment for the owned attributes.

- **Data Type**: Data Types are also extensions of Types, but only act as grouping container for properties. This means they don’t have an own identity and can only be used with other model elements. DataTypes have the same representation like Classes, but are marked with «Data Type» above the DataType name.

\*\*\*For analytical cases also the ‘Object Diagrams’ can be helpful. They illustrate the information state of object instances at a specific point of time.\*\*\*
• **PrimitiveType**: In UML there are only four basic primitive types defined: Boolean, Integer, String and UnlimitedNatural. Those types can be directly used without definitions.

If UML is used for logical modelling, platform primitives (e.g. Java Types like double, float, int, char, ...) can be defined as primitive types. This simplifies the platform-specific modelling, but limits the interoperability between different platforms. PrimitiveTypes are depicted as rectangle with the marker «PrimitiveType» and the name.

• **Property**: Properties are used to describe the characteristics of Classes and DataType. They are defined by their name and type and the values indicate the state of the objects. Properties are nested in the graphical representation of DataTypes, Classes or Associations in a special compartment.

• **Association**: Associations are similar to Classes, but contain at least two association ends, pointing to referring types. Associations can also point to more than two types and carry additional properties. Depending on the strength of the relationship between types, different association strengths are defined, 'Composition' for very strong dependencies, 'Aggregation' for close relationships and 'Association' for other relationships. The semantic differentiation between those association types is not clearly defined.

Associations are noted as simple line between the referenced types with labels for name, cardinalities and role names arranged along. The association strength is marked by a symbol at the associations ends, the filled diamond for compositions and the empty diamond for aggregations.

• **Generalisation**: Generalisation expresses the possibility to group types according to common properties and associations. Generalisations are directed associations and drawn as arrow with an empty triangle as arrow head.

• **Package**: Because UML models can be complex and large, packages are used to structure models and to define unique namespaces. Within one package all elements must differ in name and can be addressed via package name and element name. Packages are visualised as rectangle with a smaller one at the top left corner that contains the package name.

An example of an UML class diagram can be found in figure 2.2.
CHAPTER 2. STATE OF THE ART

Figure 2.2: Example of an UML Class Diagram (taken from the UML Infrastructure [OMG - Object Management Group, 2007b]).

UML Profiles

UML profiles offer limited possibilities in extending the language itself. With a profile it is possible to extend the Unified Modelling Language with new language properties. Profiles are often used in logical modelling, when specific platform constructs are represented in the logical model; e.g. profiles for Enterprise Java Beans [EJB 3.1 Expert Group, 2009] or persistence frameworks like Hibernate [JBoss Community, 2011].

A ‘Profile’ is a special package that contains all extensions in forms of ‘Stereotypes’ and ‘Tagged Values’. ‘Stereotypes’ extend language elements from the UML language definition (mostly the UML Class is extended). ‘Tagged Values’ serve as kind of new property types for the created Stereotypes.

The graphical definition of a profile is the UML Package with the «Profile» above the Profile name. The Stereotype takes the notation of Classes with the extra «Stereotype»; the Tagged values are listed within the Stereotype in a compartment. The base class of the Stereotype is linked with the ‘Extension’ arrow with the filled arrow head (see Figure 2.3). After the definition of the profile and the stereotypes, they can be used the applying

---

5Restrictions are also possible by using OCL, but complex, as language dependencies may be disturbed.
the profile and use the Stereotypes with the notation «Stereotype Name» in guillemets.

![Diagram](image)

**Figure 2.3:** Example of a simple Profile, containing one Stereotype ‘device’ for Classes with the 3 Tagged Values ‘author’, ‘color’ and ‘volume’ (taken from [OMG - Object Management Group, 2007b])

### UML for Geospatial Modelling

UML is the dominating language for several modelling purposes; it has got a clear notation, wide acceptance and several modelling tools are offered. For that reason UML class models are often also used to model geospatial information systems. UML is selected by the ISO Thematic Committee for Geographic Information/Geomatics (ISO/TC-211) [ISO/TC-211, 2005a] and by the American Federal Geographic Data Committee [Department of Homeland Security-FGDC, 2009] as basic language for their geospatial standards and models.

But the use of plain UML is not satisfying for the geospatial domain, because there are too many formulation alternatives and too high complexity in aspects like geometry, quality or space-and-time (see [Brodeur et al., 2000], [Casanova et al., 2000] or [Belussi et al., 2006]). For that reason ISO/TC-211 published a set of specification documents containing modelling guidelines, UML profiles, reusable UML packages and encoding rules:


- ISO-19103 [ISO/TC-211, 2005a], explains the usage of conceptual schema languages for geospatial modelling, including the profiling of UML for geospatial modelling.

- ISO-19107 [ISO/TC-211, 2001], contains the spatial schema, i.e. the classes and packages for spatial objects and relations.

- ISO-19109 [ISO/TC-211, 2005b], introduces the general feature model, which forms the conceptual basis for the creation of geospatial schemas.

- ISO-19118 [ISO/TC-211, 2005c], describes how UML schema for geospatial information models can be transferred to concrete formats.
• GML 3.2.1 [Open Geospatial Consortium, 2007] or ISO-19136 [ISO/TC-211, 2007], describe the XML format for storage and transfer according to ISO-19118.

**General Feature Model** The basic concept for modelling according to ISO/TC-211 is the definition of a ‘Feature’. A Feature describes one element of the real world, or more precisely the Universe of Discourse, with specific behaviour, state, relations to other Features and an own identity. Similar Features can be grouped to a ‘FeatureType’, which is characterised by sets of attributes.

![Figure 2.4: Extract from the General Feature Model (taken from [ISO/TC-211, 2005b])](image)

Those definitions are specified in ISO-19109 [ISO/TC-211, 2005b], the ‘General Feature Model (GFM)’. This model is very similar to parts of the core concepts of UML (compare figures 2.2 and 2.4); an Object matches a Feature, a Class matches a FeatureType, Inheritances and Associations are

---

6For dynamic behaviour also operations can be defined
defined similarly. But despite to UML, where those definitions (the abstract syntax) result in a complete modelling language, it is not intended to create a new modelling language, but to integrate into existing languages, preferably UML.

The mentionable problems of double definitions of concepts like properties or associations lead to unnecessary complexity and a formal weakness, which is described in the thesis of Einspanier [Einspanier, 2005].

**Conceptual Schema Language** ISO-19103 [ISO/TC-211, 2005a] describes one method to align UML to be used for geospatial modelling according to the General Feature Model. It consists of some general restrictions in the usage of UML Class Diagrams, a set of standard types to be used and some stereotypes with their meaning.

The most important restrictions to follow are avoiding multiple inheritances, multiple associations and association with attributes. Furthermore a strict package structure is suggested that limits the nesting of packages.

The type system introduces ‘Primitives’, including the sub-packages ‘Numerics’, ‘Text’, ‘Truth’, ‘DateTime’, ‘Multiplicities’ and ‘Enumerations’. The package ‘Implementation’ includes ‘Collections’, ‘Records’ and ‘Names’; the package ‘Derived’ introduces different basic units of measurement and fitting measurement types. Those units of measurement cover fields like area, time, speed, angles, length etc.

Stereotypes like «CodeList» or «Union» are introduced to indicate special behaviour of classes. They are used to impose the General Feature Model into UML. The stereotypes must especially be respected when translating conceptual models to logical or physical model. It is notable that the stereotypes of ISO-19103 are not defined as precise and valid profile for UML.

**Spatial Schema** A set of spatial types is described and defined as UML elements in the ISO-19107 Spatial Schema [ISO/TC-211, 2001]. The package ‘Geometry’ contains all types with their inheritance hierarchy, their relations to other definition elements and their spatial properties (see figure 2.5).

For most applications a small subset of the spatial types, limited to primitive geometry types, i.e. GM_Point, GM_Curve, GM_Surface and GM_Solid, is sufficient.

**Encoding** The encoding describes the process of translating the conceptual model to data formats. Beside ISO-19118 [ISO/TC-211, 2005c], mainly the Geographic Markup Language (GML) [ISO/TC-211, 2007] is of interest.

GML offers not only the way to define XML-Schemas for different kinds of geospatial information, but also includes rules to derive the Schema from
Figure 2.5: Basic geometric types of ISO-19107 and their inheritances (taken from [ISO/TC-211, 2001])
2.2. CONCEPTUAL MODELLING LANGUAGES

UML models.

**ISO/TC-211 Modelling Process**  The above described components build the minimum framework for conceptual geospatial modelling according to ISO/TC-211. The approach can be regarded as precise and sufficient for most modelling needs, but exposes the high complexity of multiple different standards to the user. In addition to that complexity, some topics are not solved satisfactory:

- The General Feature Model uses an additional abstract syntax to the UML modelling language, i.e. the classes and associations of UML can be overloaded with a different meanings and rules. This requires additional knowledge on the usage of UML and may cause information losses [Einspanier, 2005].

- ISO-19103 [ISO/TC-211, 2005a] defines several additional rules for the usage of UML, e.g. limitation to simple inheritances or the specific characteristics of code lists. These rules are neither reflected by the modelling language UML nor by the supporting tools, so that users are responsible for the correct implementation.

- The encoding rules defined in GML [ISO/TC-211, 2007] are not applied on a pure conceptual model, but on a logical model representation. The multiple stereotypes with the tagged values are adjusted to the production of GML Schema. The same is valid for structural limitations like the restrictions on associations, inheritances and packages. This contradicts the ‘pure’ ideas of conceptual modelling and strongly influences the perception of the Universe of Discourse.

**Experiences**

UML is the de-facto modelling standard in software development, although quite complex it is widely accepted and well-supported by many modelling tools. UML is designed as ‘General Purpose Modelling Language’, i.e. a language that could be used for all kinds of modelling purposes. This flexibility is achieved by defining just the language itself, but leaving the interpretation on how to apply open [Dobing and Parsons, 2005]. For spatial modelling, it is e.g. not defined how real world objects should be mapped to UML objects or when to use associations instead of inheritances. Another important instrument to achieve adaptability are UML profiles.

UML is also widely used in geospatial modelling; the American Federal Data Committee (FGDC) uses UML for their data models (e.g. [Department
of Homeland Security-FGDC, 2009)) as well as the European Commission in their INSPIRE process (e.g. [INSPIRE Drafting Team "Data Specifications", 2010]). Both models are completely incompatible as they use different declarations and naming for spatial types like 'Polygon' or 'GMSurface' and use different profiles with various stereotypes like «Feature» and «FeatureType» for marking real world object types.

The intention of ISO/TC-211 is to overcome such interoperability issues by standardisation. But the rules are rather complex and spread over several documents. Currently no adequate modelling environment covers the additional rules, the profiles and simple access to relevant spatial types. The responsibility to follow all rules is left to the modeller.

2.2.2 INTERLIS 2

The development of INTERLIS started in the early 1990s, when Swiss mapping agencies were in need to simplify the exchange and quality assurance process for land survey data. The requirements towards flexibility, multilingualism, versioning and extensibility were rather high (see [Steudler, 2006]) and led to the development of a conceptual modelling approach instead of the expected exchange format. INTERLIS covers the complete workflow from conceptual modelling over the encoding of concrete file formats to data exchange and validation (see Figure 2.6).

![Figure 2.6: The INTERLIS approach for data exchange (adapted from KOGIS, 2006b)](image)

The INTERLIS modelling language is the key element of the approach. It is a textual modelling language based on Entity Relationship concepts.
with the key modelling elements 'table' and 'relation' as well as basic types for points, lines and different polygons. With respect to the main target, the specification of exchange procedures for spatial data, constraints like maximum overlaps or intersections, bounding regions or spatial reference systems can be easily defined. After some years of practical experiences and investments in open-source software for support, the language was accepted as Swiss Standard SN 612030 and intensively used in Swiss land survey.

In 1998 the development of INTERLIS 2 [KOGIS, 2006b] was initiated. Based on the experiences of INTERLIS, additional language elements for structuring the models, better support for versioning and incremental updates and the possibility to define graphical representations were introduced. The modelling concept evolves from Entity Relationship to simple Object Orientation. The former ‘table’ element is matched to ‘object’ and inheritances as well as limited polymorphism were introduced. In addition to the textual language specification, which is well suited for precise definitions in legal texts, a graphical modelling tool was developed [Eisenhut, 2004] with a notation similar to UML.

**INTERLIS 2 Language Elements**

The INTERLIS 2 language is defined in the Reference Handbook [KOGIS, 2006b]. For all elements an explanation of the usage, the complete notation in Backus-Naur-Notation and explanatory examples are given.

- **Class**: The basic element for modelling is the Class. An INTERLIS 2 class is similar defined as an ISO/TC 211 feature type. The instances of classes, the concrete objects are individual and referable. The inheritance of Classes is limited to simple inheritance to only one superclass. A special form of inheritance can be used within topic inheritance (see below), where a Class can 'replace' its superclass. This form of inheritance is specially designed to support multilingualism and model refinements.

- **Structure**: A Structure is similarly defined as a Class, but intended to represent only a composed data type, without own identity; this might e.g. be an address type that is used for persons, companies or buildings.

- **Attribute**: Attributes define the state of classes or structures. They have name, cardinality and assigned type or domain. A special aspect of INTERLIS 2 attributes is the possibility to define or restrict types or domains inside the attribute definition. This creates anonymous types like illustrated in listing 2.1.

---

8 Additionally there is a self-definition of the language [KOGIS, 2006a] - i.e. the INTERLIS 2 language elements are expressed in INTERLIS 2
CHAPTER 2. STATE OF THE ART

CLASS Street =
  Name: MANDATORY TEXT +32
END CLASS;

Listing 2.1: INTERLIS 2 definition of a simple class with an inlined domain definition

- Associations: INTERLIS 2 offers a complete definition for associations. Associations can contain several association ends, own attributes and can be extended. The association ends carry the strength, role names and cardinalities.

- Topic: Topics offer grouping of elements similar to UML packages. They define local namespaces, but cannot be nested. Topics are semantically richer than UML packages. They support translations and inheritances. This allows the definition of multilingual models and the refinement of models. This is frequently used, e.g. to define a countrywide model and refine it for local use.

INTERLIS 2 is a specialised modelling language for spatial data. The type system strongly differs from the general purpose modelling language UML. UML has got a small set of standard primitives; all other types are either imported from the target platform or modelled from those primitives. INTERLIS 2 offers a different concept. Except for the boolean type, which can be used directly, all types must be defined on the model level. Specific ‘Domain’ elements exist for strings, numerics, enumerations and spatial types. For all domains, restricting inheritances can be defined; e.g. numeric values ranging from 1 to 100 can be restricted to 10 to 20, string values with maximum 20 digits can be reduced to 10, enumerations with five entries can be restricted to three of them.

- Text: A Text type is defined by its maximum length (see listing 2.2).

```
NAME = TEXT *255;
```

Listing 2.2: INTERLIS 2 definition of a textual type (taken from [KOGIS, 2006b])

- Numeric: Numeric types are specified with their maximum and minimum values, their precision and optional units. Additionally they can be assigned to reference systems (see listing 2.3).

```
normal = 0.00 .. 7.99;
exact EXTENDS normal = 1.0000 .. 7.9949;
lengthInM EXTENDS Distance = 0.00 .. 100.00 [m]
```

Listing 2.3: INTERLIS 2 definition of three numeric types (taken from [KOGIS, 2006b])
2.2. CONCEPTUAL MODELLING LANGUAGES

- Enumerations: Enumerations are types, that contain lists of literals. They can be used as choices for attributes. Enumerations can be nested and can extend or restrict other enumerations (see listing 2.4).

```
Weekdays = (Working days (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday), Sunday) CIRCULAR;
```

Listing 2.4: INTERLIS 2 definition of an enumeration with nesting and cyclic behaviour (taken from [KOGIS, 2006b])

The four geospatial primitives of INTERLIS 2 are also defined as a kind of templates:

- Coord: Coordinate or Point types are described by their axes. The axes are numeric types that usually have a value range, units and are assigned to concrete reference systems. The sample in listing 2.5 illustrates the definition of a two dimensional WGS 84 coordinate type. There are two axes defined with their value domains from -90 to +90 and 0 to 360, respectively. The notation of the coordinates is in degree and the reference system is WGS84 (see listing 2.5).

```
WGS84Coord = COORD −90:00:00 .. 90:00:00
[ Units . Angle _ DMS ] { WGS84[1] },
0:00:00 .. 359:59:59 CIRCULAR
```

Listing 2.5: INTERLIS 2 definition of a WGS 84 coordinate type (taken from [KOGIS, 2006b])

- Polyline: Polylines describe linear elements. They are defined by their vertices (using coordinate types), the used line-forms (e.g. straights or arcs) and additional tolerances for intersections or overlaps. These tolerances are esp. important for quality control of cadastral data (see listing 2.6).

```
WGS4Line = POLYLINE WITH (STRAIGHTS)
VERTEX WGS4Coord ;
```

Listing 2.6: INTERLIS 2 definition of a line string type using the WGS84 Coordinate type (taken from [KOGIS, 2006b])

- Surface and Area: The two types for polygons are defined similar as the polylines, but imply that the linestring is closed. The Surface is used for standard polygons, the area is intended for subdivisions of the area of interest. Those subdivisions are suited for cadastral parcels or land classifications and offer possibilities for automated plausibility checks. A further option is to attach properties to the bounding lines, without defining the boundary as own object (see listing 2.7).
Experiences

As Swiss standard for conceptual geospatial data modelling, INTERLIS and INTERLIS 2 are used in many data models, like in AV93 the official data model of the Swiss Land Survey (see [Steudler, 2006]) or the meta data model GM03 [KOGIS, 2003]. The capabilities in multilingualism and the easy way of extending the models are of practical use in defining regional or local models that add special aspects to national models (e.g. [Vermessungs- und Meliorationsamt Basel Landschaft, 2008]). The textual notation of INTERLIS 2 is well-suited for the precise description of large data models; it offers a compact thus readable form, especially for classes and data types. The shortcomings are the initial effort in learning the specific language and the non-intuitive representation of relations and hierarchies.\(^9\)

In summary, the INTERLIS approach with the set of supporting tools offers a rather complete mechanism for the exchange of geospatial data, including multilingualism, extensibility and precise definitions of value domains. The modelling language is highly adjusted to the needs of public administration. It directly contains concepts of space and supports typical key elements of public data, like quality constraints, precisions, value domains or reference systems.

INTERLIS 2 is a typical 'domain specific language' (see chapter 2.4.2).

2.2.3 MADS

Another specialised conceptual modelling language for geospatial data is the MADS ('Modelling for Application Data with Spatio-Temporal Features') approach [Parent et al., 1998]. It was developed in the late nineties at the Ecole Polytechnique Fédérale de Lausanne and the University of Lausanne. The approach addresses the design of complex, geospatial and temporal databases by offering a conceptual modelling language, a graphical schema editor, encoding scripts for databases and GI Systems as well as a querying tool.

The MADS language mainly describes Object Types and their Relations. Similar to INTERLIS, it was developed first in a relational manner, but later

\(^9\)The software INTERLIS UML Editor adds a more intuitive graphical notation [Eisenschütz, 2004]
extended to support object-oriented elements, like inheritance, object methods and attribute nesting. The approach introduces the idea of ‘orthogonality’: thematic, spatial and temporal aspects are regarded to be independent and have their own model representation. Beside typical geospatial data types, special focus is drawn on the definition of constraints and relations with temporal and spatial components. A textual and a graphical notation is offered, with a clear focus on the graphical representation. The concept of orthogonality is realised with special icons for different geometrical and temporal types.

The language definitions can be found in Annex A of the book ‘Conceptual modeling for traditional and spatio-temporal applications : the MADS approach’ [Parent et al., 2006a]. This book includes also a textual description of the language, with explanatory examples. The information about MADS presented in this work are mainly taken from this book.

MADS Language Elements

As mentioned, one key characteristic of MADS is the ‘orthogonality’ of thematic modelling and the aspects of space and time (in this work the temporal elements are out of scope). First the elements for specific thematic modelling are described, later the symbology and elements for spatial aspects.

- **Object Types** are used to group similar objects with identity from the Universe of Discourse. Object Types have attributes, methods and are addressable with an unique name. One or a combination of attributes can be declared as identifying key.

- **Relationships** describe the associations between objects or their according object types, respectively. The association ends, the ‘Roles’ carry information like cardinality and name. Each relationship should have an identifier for the database representation. To help modelling real world phenomena, the associations can have special association types: ‘Aggregation’ to indicate composing relationships, ‘Transition’ for time-related changes of object types and ‘Generates’ for time-related object generation.

- **Multi-Association Relationships** facilitate the description of multiple representations of objects in spatial databases. In a multi-association relationship, a group of objects from one object type can be associated to a group of objects of another object type. This is a common situation if objects are captured using different rules. In Figures 2.7 and 2.8 the simplification is illustrated with an example of groups of houses that represent the same real world entity.

- **Is-A Relationship** is used as refining or sub-grouping relationship for Object Types or Relationships. Despite to UML inheritance, MADS
Figure 2.7: Example of conventional mapping of related groups. Building representations for specific scales (1:15000 and 1:25000) are grouped together and mapped by the ‘Corresponds’ relationship. (taken from [Parent et al., 2006a])

Figure 2.8: The same association between grouped objects for building with the simplifying multi-association relationship type (taken from [Parent et al., 2006a])

precisely defines the rules applied, i.e. it is defined how attributes are inherited and how to solve conflicts in multiple inheritances. By either using a common inheritance arrow or separated arrows, it can be indicated if the inherited types are mutually exclusive or overlapping (see figure 2.9).

Figure 2.9: MADS Is-A relationship with two inheritance lines; private and public house are mutually exclusive, while private and living house may be overlapping (taken from [Parent et al., 2006a])

Spatial Types

Similar to other modelling languages MADS offers a limited set of primitive data types like Integer, Real, String or Boolean, which are designed analogous to common database types. Additionally a set of spatial (and temporal) attribute types is defined to meet all requirements for typical spatial data models. All spatial types inherit basic properties and behaviour from the
super-type ‘Geo’, including the minimum bounding box and the spatial reference system (see Figure 2.10). The simple types ‘Line’ and ‘Surface’ have refinements for directed lines and polygons without holes. The ‘Bag’ types are used for geometric collections.

Additionally to simply use geometric attribute types, it is also possible to directly mark object types with according symbols. This offers a very concise notation, as the spatial objects and their geometry types are directly visible in the models. Additionally MADS offers the possibility to describe spatial relationships, like ‘intersect’, ‘touch’ or ‘within’ in a simple manner already on the conceptual level (see Figure 2.11).
Experiences

Similar to INTERLIS 2, MADS was developed to cover specific spatial modelling tasks. One of the targeted application field was complex natural risk modelling in the MurMur project [Parent et al., 2006b]. On one hand a focus was set on complex modelling of spatio-temporal aspects and relations. On the other hand, especially problems resulting from different perceptions and representations for the same phenomena were addressed. MADS was designed for the production of database schemas, so that some model elements are more located on the 'logical' level; e.g. the primary keys or object- and relation-identifiers.

Although even some open source tools for modelling and schema generation were developed, MADS is not widely used. It might be too specialised for most application fields and is only driven by a small user and developer community.

2.3 Model Transformation

Model transformations are a key element in the work with conceptual models. They allow to derive target products in different forms from one or more input models (see e.g. Mens and Van Gorp [2006]). The variety of targets and transformation techniques is manifold. Depending on the target definition, the transformations can roughly be classified into horizontal and vertical transformations (see figure 2.12). The horizontal transformations deal with targets on the same abstraction level (in this work the conceptual models), whereas vertical transformations are needed to bridge the different abstraction levels. For all transformations it is obvious that information losses can occur, if the target structure is partially less expressiv, or information must be added in the transformation to fill information gaps in the source model.

In this chapter some transformations for geospatial models are presented.

2.3.1 Vertical Transformation

Most geospatial modelling languages are directly coupled with standard targets and according vertical transformation approaches. INTERLIS was mainly developed to optimise data exchange and generates standardised as well as proprietary exchange formats. MADS is optimised for the creation and maintenance of complex spatial databases and the ISO/TC-211 modelling with UML focusses strongly on the encoding of spatial data with GML.

UML to GML-Schema GML is an adopted ISO standard [ISO /TC-211, 2007] and actually the most prominent data format for vector data, especially due to the increasing importance of Web Feature Services [Open Geospatial
Figure 2.12: Transformations related to conceptual data models: (1) illustrates a horizontal transformation between datasets of different structures. The definition of the data transformation is done on the conceptual (or logical) level; (2) illustrates a vertical transformation to generate a logical or physical data schema from a conceptual model; (3) shows a horizontal transformation between two conceptual models of different modelling languages. Target of this transformation to switch between different approaches, while keeping the model content as far as possible.
Consortium, 2005]. It is based on XML technology and offers spatial and
temporal data types for exchanging spatial data. To use GML as file format
for data transfer, GML Schema must be defined. GML Schemas are simple
XML Schemas that follow specific creation rules and use predefined Schemas
with spatial and temporal types as imports.

Transformation rules to derive valid GML Schemas from ISO/TC-211
compliant UML models can be found in Annex E of the GML specification
document [ISO/TC-211, 2007], building on the experience of Grønmo et al.
[Grønmo et al., 2002].

To apply those transformation rules, the conceptual UML model must
be extended (‘marked’) by stereotypes and tagged values (see chapter 2.2.1).
Beside the manual transformation, some automated transformation engines
are available, e.g. shapechange\textsuperscript{10} or the fullmoon framework\textsuperscript{11}.

**INTERLIS Conceptual Model to INTERLIS Exchange Format**

INTERLIS was designed to allow complete or incremental transfer of geospa-
tial data in a platform-independent way. Therefore, the transformation
rules from the INTERLIS conceptual model to INTERLIS transfer format
are an integrated part of the specification document [KOGIS, 2006b]. The
INTERLIS specification splits the rules for encoding the conceptual model
to an exchange format into two sections. First the general requirements
towards an encoding and second the concrete transformation rules to the
exchange format. The actual official transfer format is defined as XML-
Schema with all type definitions and a backwards reference to the according
INTERLIS 2 model. All specific properties of the INTERLIS 2 model are
completely reflected in the exchange format.

Transformations of INTERLIS 2 to GML-Schema are also possible, but
not yet standardised [Eisenhut, 2010] as different mapping possibilities for
inheritances, nested types or relations exist. The generated GML-Schema
will not cover all information and capabilities of the INTERLIS concept
like multilingualism, spatial constraints or inheritances of complete models.
There is a major information loss for this transformation, due to the limited
expressiveness of GML.

The open source software ‘INTERLIS Compiler’ (http://www.interlis.ch)
can validate the INTERLIS 2 models and perform the transformation to
generate INTERLIS exchange formats as well as GML-Schema or other pro-
prietary formats like ‘Autocad dxf’ or ‘ESRI shapefiles’.

**MADS to Database**

MADS was created to support the design of spatio-
temporal databases. The intended target systems are primary (object-)

\textsuperscript{10}http://www.interactive-instruments.de

\textsuperscript{11}https://www.seegrid.csiro.au
relational databases and GI Systems. In [Parent et al., 2006a] the transformations between the MADS language models and relational tables are described in detail. This includes the description of information losses, due to insufficient ways of describing spatial or temporal types, constraints and associations in the logical models. Free Software for schema generation of database schemas for Oracle, MapInfo and ArcGIS is available.

2.3.2 Horizontal Transformation

According to current focus on data harmonisation, several research activities took place in recent years on the horizontal transformation of datasets (see [Humboldt Consortium, 2011], [Orchestra Open Architecture and Spatial Data Infrastructure for Risk Management, 2007] or [Staub, 2009]). The activities aim to define transformations on a high abstraction level and execute them on datasets (see (1) in figure 2.12). As examples, geospatial ETL tools and the Model-Driven Web Feature Server will be shortly described in this chapter, although those data transformations are not in the scope of this work.

Though another horizontal transformation is a focus of this thesis; the transformation of one conceptual model to another conceptual model of a different modelling language without changing the model content (see (3) in figure 2.12). Those 'content preserving' language-to-language transformations facilitate interoperability between different modelling approaches. As example, the UML import feature of the INTERLIS 2 UML editor is described.

**UML to INTERLIS 2** The open source tool ‘UML-Editor’\(^\text{12}\) tries to simplify the modelling process for INTERLIS 2 by offering a graphical interface based on UML notations. The tool utilises the similarity between both modelling languages and enriches the UML language with INTERLIS 2 specific elements. This results in an alternative, graphical representation of INTERLIS 2, which looks similar to UML.

A specific feature of the ‘UML-Editor’ is the import functionality for Class Diagrams of UML models. This import is limited to models in Rational Rose\(^\text{TM}\) UML format and only covers the basic UML constructs like package, class, association and attributes, but no specific geospatial aspects like ISO-19103 stereotypes or packages of ISO-19107.

**Geospatial ETL Tools** Extract-Transform-Load (ETL) tools have their origin in data warehouse applications, where different data sources must be analysed, merged and consolidated. With those tools different data sources can be accessed, structures can be analysed and directed mapping rules can

\(^{12}\text{http://umleditor.org}\)
be defined. With the defined mappings on the logical data models data transformations can be executed.

For geospatial data handling some commercial and very few open source tools are available (Safe Software FME\textsuperscript{TM13}, Talend Spatial Data Integrator\textsuperscript{14} or HUMBOLDT Alignment Editor\textsuperscript{15}. Those tools offer specialised data accessors and several spatial operations including buffering, intersections or type conversions.

**Model-Driven Web Feature Server** The Model Driven Web Feature Server (MDWFS) was a research project driven by the needs of public spatial data infrastructures [Donaubauer et al., 2007]. Target was the development of an adaptable service to deliver spatial data in different data structures, which might differ from the original one. Beside the specification of an extended WFS [Open Geospatial Consortium, 2005] interface, a mapping language for conceptual models was developed. The approach uses UML 2 activity diagrams as base for the mapping language (see [Staub et al., 2008]) that can be used for transformations on the physical level.

Although some promising prototypes were developed, neither detailed documentation nor usable implementations are available.

### 2.4 Model-Driven Software Development

Although main ideas and technologies of geospatial modelling are initially based on standards of model-driven software development, it is obvious that in software engineering big technical achievements have been reached, which are not fully reflected in geospatial modelling approaches. Especially working with languages and transformations evolved from rather static approaches to flexible and dynamic technologies. With these techniques it is now possible to offer better support for domain experts in the model creation and to simplify transformations towards other models or software products with little efforts.

In this chapter basic techniques and terms of actual model-driven software development will be introduced without turning too technical. More focus will be spent on the concepts of language design and meta modelling.

Working with models is a substantial part of modern software development. Compared to early approaches, where models were mainly used for rough design and software documentation, models are nowadays a main input for the creation and maintenance of software systems. Object-oriented programming languages, standardised multi-tier applications and reusable

\textsuperscript{13}http://www.safe.com/
\textsuperscript{14}http://www.spatialdataintegrator.com/
\textsuperscript{15}http://community.esdi-humboldt.eu/projects/show/hale
components and frameworks helped model-driven software development during the last 5 years to evolve from its niche to IT mainstream.

Due to the rapid developments in this field, there are several slightly different and overlapping definitions, terminologies and approaches. The most prominent are ‘Model-Driven Engineering’, ‘Model-Driven (Software) Development’ and ‘Model-Driven Architecture’, which are all closely related, but focus on slightly different aspects:

**Model-Driven Engineering (MDE)**

Model-driven engineering (MDE) is a discipline in software engineering that relies on models as first class entities and that aims to develop, maintain and evolve software by performing model transformations [Mens and Van Gorp, 2006].

This definition explains clearly the intention of models as main input for software systems. The model is not only used as temporal element during the development phase, but acts as the key element, which should also be used to modify the system. Using models just as design sketches or for documentation purposes is not sufficient from the perspective of MDE.

**Model-Driven Software Development (MDSD) or Model-Driven Development (MDD)**

Model-driven software development is set as general term for all techniques, which derive runnable software systems from formal models [Stahl et al., 2007]. It is focussed on the issue of software creation but also includes the possibility of interpreted models, which means the runtime behaviour of a software framework is controlled by models. In addition, models might only be used to describe only parts of systems, e.g. data access or user interfaces, but those parts must be defined completely, so that the produced software ‘fragments’ are closed and can easily be maintained by the model.

In sum, the MDSD definitions are very flexible and pragmatic.

**Model-Driven Architecture (MDA)**

The concepts of the Model-driven architecture [OMG - Object Management Group, 2009] are defined and maintained by the Object Management Group (OMG). MDA can be regarded as a specialisation of the MDSD [Fowler, 2010], using only technologies defined by OMG (UML - Unified modelling Language, OCL - Object Constraint language, QVT - Query View Transformation and MOF - Meta Object Facility). Although MDA offers clear concepts and good descriptions, which are also used in this chapter, the scope is often too limited. Beside the fixed technologies, also the scope of the transformations is reduced to the generation of code and code fragments from platform-independent and platform-specific models (see Chapter 2.4.1).
2.4.1 Terms

The main ideas of the presented approaches are all similar; deriving software code from domain-oriented models. Comparable to the abstraction levels of data modelling (conceptual, logical, physical) most approaches differentiate between computational-independent, platform-independent and platform-specific models (see figure 2.13).

**Computational-Independent Model (CIM)** The computational-independent model formalises the requirements from the user towards the software system, but does not include any implementation- or computer-related aspects. Those models are also called ‘domain models’ and are comparable with the ‘conceptual model’ from the field of data modelling.

Following the guidelines of model-driven architecture (MDA), the Object Management Group suggests the usage of UML for the dynamic and static viewpoints [OMG - Object Management Group, 2009].

**Platform-Independent Model (PIM)** The platform-independent model describes the targeted software system from a computational perspective without fixing to any underlying operating systems, frameworks or programming languages [OMG - Object Management Group, 2009]. It concentrates on the general adoptions of the CIM to information systems in switching terminologies, identifying information flows, adding typical design patterns or splitting into software components. Compared to data modelling, the PIM is located between conceptual and logical Model.

**Platform-Specific Model (PSM)** The platform-specific model includes the properties of the implementation base, in sense of related storage systems, programming language, operation systems etc.. The characterisation of those platform-specific properties is described in the ‘platform description model (PDM)’ that varies on platform properties. For UML models, the PDM can be supplied as UML profile and additional packages providing standard types and patterns. The PSM is comparable to the logical model in data modelling.

Similar to the classification into conceptual, logical and physical models, the classification into CIM, PIM and PSM cannot be strict. Depending on technical knowledge, on the used modelling language or the targeted system, implementation-specific aspects will be already included in the CIM. Also the differentiation between PIM and PSM can be ambiguous. The question, if specific concepts are platform-specific or valid for a group of platforms or for all platforms is difficult. Especially, the trend to use multiple slightly enriched models during a development process, e.g. first add implementation types, then add adopted structures, then add database access, etc. (see [Stahl et al., 2007]), makes clear differentiations difficult.
2.4. MODEL-DRIVEN SOFTWARE DEVELOPMENT

Figure 2.13: Computational-independent model (CIM), platform-independent model (PIM), platform description model (PDM) and platform-specific model (PSM); illustration of their dependencies and actors.

Transformations  Beside the different models, transformations are the other key element in model-driven software approaches;

Model transformation is the process of converting one model to another model of the same system. [OMG - Object Management Group, 2009]

Although a wide variety of transformations can be found in MDSD [Mens and Van Gorp, 2006], like merging of models, enrichment of models, restructuring or changes of modelling languages, most actual approaches strongly focus on vertical transformations to derive platform-specific models and software code from platform-independent models. Only very recent developments in domain-specific modelling languages (see chapter 2.4.2) also opened the scope for horizontal transformations between different modelling languages and include the computational-independent models in the production chain.

Figure 2.14: Vertical transformation of a platform-independent model to a platform-specific model (taken from OMG - Object Management Group [2009])
In figure 2.14 the MDA visualisation for transformations is depicted. It illustrates beside the vertical transformation, how additional information (the empty box) is added to the system. This is notable, as all transformations include information that enriches the models. The type of that additional information might vary from textual rules for manual transformations over built-in implicit mappings, which are hidden in ‘black-box’ implementations, to clear and formal ‘mapping rules’ in forms of mapping languages or mapping models. MDSD approaches should build on formal ‘mapping rules’.

**Mapping** As input for transformations, two main types of mappings can be distinguished (see OMG - Object Management Group [2009]); mapping on the level of models and mappings defined on the level of the modelling language:

- ‘Instance Mappings’ are mappings on concrete models. For the transformation either specific model patterns are used, e.g. specific combinations of elements, or the models are ‘marked’. Whereas the structural patterns are rarely used, marking is very common. Marking is done manually on the individual model, e.g. by using special name-prefixes or stereotypes with tagged values to indicate how a specific model element should be handled during the transformation process.

- ‘Type Mappings’ are a structural mappings defined on the level of the used modelling languages. Elements of the source model’s language are mapped to elements of the target’s model language. Those mappings can be used highly efficient for transforming models and do not require manual model manipulations (‘marking’) on the source models. But ‘type mappings’ can certainly only be applied for languages that are capable to carry all required input information.

A special type of those language-based mappings are the ‘Meta Model Mappings’ (see [OMG - Object Management Group, 2003]), which offer additional simplification of mappings for languages defined on the same meta-modelling approach (see figure 2.15 and chapter 2.4.2).

Generally, language-based mappings are considered to be highly efficient, as mostly simple and precise mappings can be defined. But they require as a prerequisite suitable languages for the input and output models, which are not always given. So, in practical use several mix-forms are used, like mapping one language element to a model pattern or vice versa. Also, often stepwise transformations are performed, first language based transformations and after that, detailed transformations are controlled by model mappings with markers.

Additionally, the mappings can follow different strategies; e.g. ‘procedural mappings’ defining step by step transformations while ‘relational mappings’ are declarative following the set theory. Mappings might be defined
2.4. MODEL-DRIVEN SOFTWARE DEVELOPMENT

Figure 2.15: OMG illustration of a model transformation using meta model mappings (taken from OMG - Object Management Group [2009]).

'bi-directional' also allowing reverse transformation or 'mono-directional' (see OMG - Object Management Group [2007c]).

2.4.2 Modelling Languages for MDSD

Modelling languages play an important role in MDSD. Especially for the computational-independent models, as bridging model between domain experts and software engineers, the requirements towards the used languages are high and manifold. On one hand the domain experts should be able to formulate precisely all their needs for a system and on the other hand software engineers should directly utilise the models in their development process. A conceptual modelling language should have following properties:

- expressive: The language should allow direct and compact modelling for the domain experts.
- complete: The modelling language should be capable of all demanded constructs.
- readable: The language should also be understandable, i.e. models described with the language should be self-explanatory.
- precise: The models created with that language should be described precisely and unambiguous.
- processable: The resulting models should be parsable by computer systems to allow automated processing.
CHAPTER 2. STATE OF THE ART

Domain-Specific Languages (DSL)

Domain-specific languages (DSLs) are languages tailored to a specific application domain. They offer substantial gains in expressiveness and ease of use compared with general-purpose programming languages in their domain of application. [Mernik et al., 2005]

Domain-specific languages exist for various fields and tasks; from simple languages for system configurations and scripting, over image processing languages like IDL, to special modelling languages as the Business Process Model Notation (BPMN) or the earlier mentioned geospatial modelling languages INTERLIS 2 or MADS.

In software development the last 15 years were characterised by a growing relevance of ‘general purpose’ languages like Java for object-oriented programming or UML for all-purpose modelling. The language sizes and complexities grew to fit mostly all tasks. Since a few years the trend is changing towards smaller and better adjusted domain specific languages.

This trend is initiated and fostered by different developments:

- Well-founded standards to define languages [OMG - Object Management Group, 2006b] allow to easily define concise languages
- New, highly flexible software approaches, like ANTLR\(^\text{16}\), Eclipse Modelling Project\(^\text{17}\) or Jetbrain’s Meta Programming System\(^\text{TM}\)\(^\text{18}\) support the creation of suitable language tools
- A shift of paradigms of the key player OMG [Watson, 2008] from extending UML towards additional languages opens the way to interoperability between new, small languages and general purpose languages

All in all, the definition and the usage of domain-specific languages is turning more and more simple.

Language Definition

In the definition of modelling languages, three aspects can be separated [Harel and Rump, 2000], [Karagiannis and Kühn, 2002]; the abstract syntax, the concrete syntax and the semantic description. Often, language definitions mix those aspects into one textual language description. But it is also possible to describe them separately as done in the UML specification. To illustrate the different aspects, parts of the UML class definition are cited as example:

\(^\text{16}\) [www.antlr.org]
\(^\text{17}\) [www.eclipse.org/modeling]
\(^\text{18}\) [www.jetbrains.com/mps/]}
2.4. MODEL-DRIVEN SOFTWARE DEVELOPMENT

- Abstract Syntax: The abstract syntax defines the properties of a language element and its relations to other elements. E.g. the textual description of the abstract syntax for a UML class is:

  Class is a kind of classifier whose features are attributes and operations. Attributes of a class are represented by instances of Property that are owned by the class. Some of these attributes may represent the navigable ends of binary associations.[OMG - Object Management Group, 2007d]

- Concrete Syntax: The concrete syntax contains the definition of the notation for the modelling elements. For a UML class it is:

  A class is shown using the classifier symbol. As class is the most widely used classifier, the keyword “class” need not be shown in guillemets above the name. A classifier symbol without a meta-class shown in guillemets indicates a class.[OMG - Object Management Group, 2007d]

- Semantic Description: The semantic description clarifies the usage and the context for the language elements. For a UML class only a rough, general context is described:

  The purpose of a class is to specify a classification of objects and to specify the features that characterise the structure and behaviour of those objects.[OMG - Object Management Group, 2007d]

For the technical implementation of tools, but especially for the definition of mappings, the abstract syntax is of major relevance. The language developments can be especially simplified if the abstract syntax is provided as formal ‘meta model’.

Meta Modelling

A meta model defines a language with another formal language. This means that the abstract syntax of an arbitrary language from the fields of software engineering is formally described.\(^{19}\)

Although this definition is a bit abstract, it offers practical advantages for transformations and language definitions, especially to build domain-specific languages in a fast and flexible way.

Not all modelling languages are defined with a formal meta model, often the abstract syntax is simply explained by textual descriptions.\(^{20}\) According

\(^{19}\)Although arbitrary languages like script languages, programming languages, modelling languages etc. can be described, this work only refers to modelling languages.

\(^{20}\)Also non-standard approaches or self-definitions are used.
to MDA terminology, those languages are called ‘Three Level’ approaches (see Figure 2.16). ‘Level 0’ indicating the object space (e.g. a specific real world object like ‘Bahnhofstrasse’ in Zurich), ‘Level 1’ is the model with its model elements (e.g. the ‘Class:Street’) and ‘Level 2’ is the language used for the model (e.g. INTERLIS 2).

![MDA model levels according to OMG, describing a three level approach with a non-formally described modelling language (like INTERLIS or MADS)](image)

Figure 2.16: MDA model levels according to OMG, describing a three level approach with a non-formally described modelling language (like INTERLIS or MADS)

Modelling languages following this approach can be precisely and well understandable described, but may not offer support for automations on language level. Especially building parsers, editors or transformers for languages can be complex and time-consuming.

If the abstract syntax of the modelling language is defined by another (simple) modelling language, this is called a ‘four-level’ approach. The additional level 3 can be guessed as modelling language for languages, the ‘Meta-Language’ or ‘Meta-Meta-Model’ (see Figure 2.17).

Those meta-languages can and should be much simpler than the usual modelling languages, as they only need to describe the static composition of the elements building the modelling language. For consistency, the meta-languages are usually self-defined, like MOF or Ecore, which are described below.

The most popular meta languages in model driven software development are currently the OMG Meta Object Facility (MOF) [OMG - Object Management Group, 2006b] and Ecore from the Eclipse Modelling Framework (EMF) [Eclipse Foundation, 2008b].

MOF is designed and maintained by the Object Management Group as part of their Model Driven Architecture. MOF is used in the UML specification and plays a prominent role in model transformations [OMG - Object Management Group, 2007c] and model storage [OMG - Object Management Group, 2007a]. The MOF specification is split up into the complete MOF (cMOF) and the subset ‘essential MOF (eMOF)’, which is sufficient to de-

---

21 An overview on other meta modelling approaches can be found in [Karagiannis and Kühn, 2002].
2.4. MODEL-DRIVEN SOFTWARE DEVELOPMENT

Figure 2.17: MDA model levels according to OMG, describing a four level approach with a formally described language (like UML)

scribe most modelling languages.

The Ecore language is a practical adoption of a meta-modelling approach within the open-source modelling platform EMF. This includes model serialisation, different model editors, model validation, model comparison and several transformation frameworks. The approach is proven to be stable and complete, so that several commercial and open-source projects utilise this technique for their modelling environments. Despite minor naming conventions and slightly different definitions, Ecore can be guessed equal to eMOF.

Figure 2.18: Simplified eMOF model depicted as UML Class Diagram (taken from OMG - Object Management Group [2006b])

In Figure 2.18 the general structure of the eMOF approach is depicted. The elements and the notation are comparable to a minimal version of UML class diagrams, concentrating only on the core, static modelling aspects.
But this simplicity is wanted and sufficient to model the abstract syntax of arbitrary languages.

The eMOF `Class' (in Ecore `EClass') is used to describe the elementary language constructs, the main 'vocabulary' of the language. The eMOF `Attributes' (`EAttributes') and `Associations' (`EReferences') allow the description of the language elements' properties and the relations to other language elements.

To illustrate the usage of a meta-modelling language, a snippet of a fictive modelling language is developed with Ecore. The language should contain an element 'myClass', similar to a primitive version of a UML Class. It should have a name, a text field to explain its purpose and may have attributes, inheritance and associations to other 'myClasses'.

Starting from the core properties, 'myClass' should have a text field for its name and a text field for its purpose, a simple Ecore model might look like Figure 2.19

![Figure 2.19: 'myClass' of a fictive modelling language modelled with Ecore](image)

To include the other properties, which are requested, the meta-model must be extended. Additional language elements for 'Attributes', 'Types' and 'Relations' must be modelled as Ecore 'EClasses' and associated with 'myClass' to achieve the requested behaviour. A sample meta-model, fitting the requirements is illustrated in figure 2.20.

![Figure 2.20: 'myClass' of a fictive modelling language with attributes and associations modelled with Ecore](image)

It should be emphasised that meta models defined in this way only carry information about the abstract syntax of the new language. The concrete
syntax is undefined, which means that multiple notations, e.g. textual, graphical, 3-dimensional or colourful graphics are possible. Also the semantics of the model is left open, e.g. an inheritance of ‘myClasses’ is defined in the ‘superclass’ association, but the inheritance behaviour is not clear.

2.5 Summary

Chapter 2 ‘State of the Art’ gives an overview of concepts and technologies relevant for this work.

The central term ‘conceptual modelling’ is defined as the activity of formally describing the universe of discourse, a subset of reality that is of relevance for the information system. The conceptual model is an important communication element between domain experts and software developers, a source of documentation and the main input for the subsequently derived logical models as well as the low level physical models.

Prominent conceptual modelling languages for the geospatial domain, the ISO/TC-211 adoption of UML, the Swiss standard INTERLIS 2 and MADS of the MurMur project are introduced and roughly described with their main characteristics. As each of the approaches focuses on specific tasks, none of them can be guessed as optimal for all purposes. Whereas ISO/TC-211 is the result of a long standardisation process and acts as a founding system for geospatial perception, INTERLIS 2 is a standard with a strong focus on correct data exchange and MADS is developed within a project for the creation of complex spatio-temporal databases. For those languages some techniques for horizontal and vertical transformations are described.

Those technical aspects of geospatial modelling were complemented by an overview of the main trends of current model-driven software development. After clarification of terminologies, the OMG levels of conceptualisation are introduced: ‘M-0’ for Instances, ‘M-1’ with the conceptual model, ‘M-2’ with the modelling language and ‘M-3’ for the meta modelling language. As most important meta modelling languages, eMOF and the implementation eCore are described.
Chapter 3

Concepts

Mentioned in the introduction, this work sets its main focus on the functionality of conceptual models as connecting elements between the domain experts and technical processing. It introduces new technologies from the field of model driven software development, especially to simplify modelling processed without loosing interoperability. Figure 3.1 focuses on the tools and processes between the different modelling levels, which were introduced in chapter 2.1.1 on general data modelling. To allow domain experts to reflect their knowledge efficiently as conceptual data models, suitable modelling environments must be provided (figure 3.1, (1)). Such a modelling environment consists of a fitting language, which allows to express all elements from the universe of discourse in a direct manner, and of at least one suitable modelling editor. The modelling editor supports the user, guarantees a minimum model quality and allows further automated processing of the model. The choice of an optimal modelling environment can vary depending on the task, the modelling domain and the user.

To achieve interoperability between different domains or tasks, it must be possible to switch between different modelling environments and therefore between different modelling languages. A model created in one conceptual modelling language can be processed after a ‘horizontal’ language-to-language transformation with tools provided for the target conceptual modelling language (figure 3.1, (2)). The main input for those horizontal transformations on the conceptual level are the source and target language with an according mapping.

The third marker in figure 3.1 illustrates a ‘vertical’ transformation to derive the targeted logical or physical model. Typical targets are GML-Schema documents for the exchange of data, but also data base structures or other file formats. Required for this transformation are tools to define and execute the transformations as well as the modelling language and a suitable format description.

Those tools and efficient languages play prominent roles in the over-
Figure 3.1: Adoption of figure 2.1 to depict the process steps between the universe of discourse and a logical or physical model. Step (1) illustrates the manual process of modelling with a helping model editor to create a conceptual model and step (3) shows the vertical transformation as a process with an according transformation tool. Process step (2) describes a horizontal transformation between different conceptual modelling approaches.
3.1. Problem Field

As described in chapter 2, there is a high degree of similarities and overlaps between Model Driven Software Development and approaches of conceptual geospatial modelling. Those similarities are certainly wanted and all approaches of geospatial modelling emphasise the adoption of model driven approaches.

ISO/TC-211 partly follows the OMG model driven architecture (MDA) and focuses on the usage of Unified Modelling Language (UML) for modelling purposes. That approach allows the modelling of spatial aspects, but the models tend to be complex and the modellers have to respect a long list of additional rules and several external sources must be included to result in consistent models. Even with UML being very huge and complex, the requested structure of ISO-19109 cannot be matched completely, because there are several conceptual overlaps between e.g. classes and feature types or package and application Schema [Einspanier, 2004]. Transformation techniques should be used according to MDA, but as the meta-model of UML and the extended UML does not cover all relevant aspects (e.g. spatial data types, spatial relations, measure types, ...) the transformations must rely on complex ‘instance mappings’. The actual existing transformation engines are designed as single-use, standalone applications and only provide a vertical transformation of the ‘marked’ UML model to GML Schema (see chapter 2.2.1).

Spatial schema languages like INTERLIS 2 or MADS are typical domain-specific languages. They are highly elaborated and their use is limited to very specific tasks or domains. Most available tools are quite good, but the maintenance and further development is limited due to the small user communities, especially for the open source parts. The model-driven approach is also emphasised for those languages, but the implementations of transformations are designed as specific software packages and only focus on the vertical transformations to exchange formats or database schemas.

Both general approaches, the use of adopted UML and the available geospatial conceptual schema languages are characterised by proprietary aspects like informal modelling rules, non-formalised modelling languages, specific tools etc.. Especially for model processing mainly proprietary and often outdated software is used and tightly coupled to other process components. This causes major problems in stability, reliability and maintenance of the processes; it is difficult to change any of the software components or modify
CHAPTER 3. CONCEPTS

the target definitions, e.g. by updating the GML versions or switching to other technologies like CORBA or JSON [Gronmo et al., 2002].

Those aspects are real burdens for interoperability and harmonisation of systems.

3.2 Working Thesis: Meta Modelling Techniques will Improve Usability and Interoperability for Conceptual Geospatial Modelling

New approaches in model-driven software development, with a strong focus on formalising modelling languages, allow concise definitions and processing of languages. This might also offer new possibilities in conceptual geospatial modelling. If those techniques can also be used in the geospatial modelling processes, interoperability problems might be leveraged.

To examine the suitability for a meta-modelling technique for geospatial modelling three assumptions will be tested:

• ‘Standard meta modelling approaches can be used to create well-defined modelling languages in the field of geospatial modelling’ (chapter 3.3). It should be tested, how existing or new conceptual geospatial modelling languages can be described by standard meta modelling approaches like eMOF or Ecore (figure 3.2, (1)). If this is feasible, according tools like editors, transformers, code generators can be used to simplify further processing.

• ‘Horizontal transformations between models described in different modelling languages can be performed with standard transformation tools’ (chapter 3.4). If the first statement is valid, standard transformation frameworks like OpenArchitectureWare [Open Architecture Ware, 2008], Atlas transformation system [Eclipse Foundation, 2008a] or QVT [OMG - Object Management Group, 2007c] can be used to transform a geospatial conceptual model defined in one language to an according model defined in another language, as long as both languages are based on the same meta modelling approach (figure 3.2, (2)). This might open interoperability between different modelling approaches and would allow to use very high specialised, user friendly modelling languages and respective tools. Models in those languages can then be translated to standard languages and be processed via established procedures.

• ‘Vertical model transformations, esp. transformations from conceptual to logical or physical models can be facilitated by the use of well-defined modelling languages’ (chapter 3.5). Transformations using language elements are simple to define with
modern code-generation utilities of MDSD. This can simplify transformations to common exchange formats like GML Schema or database definitions and reduces the dependencies on specific file formats or proprietary software products (figure 3.2, (3)).

![Figure 3.2: Detailation of figure 2.12 to illustrate the work assumptions of this thesis: (1) Different spatial conceptual modelling languages can be described with a common meta modelling approach. (2) Based on the defined meta models, transformations between different conceptual schema languages are possible. (3) The language based tools can also be used in vertical transformations between conceptual and logical/physical models.](image)

The assumptions are tested in this thesis, first on an exemplary analysis, to figure out the general potential and detect possible conceptual problems and second by implementing those examples with actual software tools. Such implementations help finding problems and allow estimations on complexity and efficiency.

Not within the scope of this work are any measures or cost-benefit analyses for different modelling languages, transformations or tool comparisons.
3.3 Meta Models of Conceptual Geospatial Modelling Languages

According to the first working thesis, it should be tested, if the abstract syntax of geospatial modelling languages can be sufficiently modelled by a common meta-modelling approach.

The selected modelling approach is Ecore, because it is on one hand nearly equivalent to eMOF and therefore compatible to the MDA approaches of OMG including UML. On the other hand Ecore offers optimal tools and an active community support, which is highly helpful for the implementations (see chapter 4).

First, it should be tested how an existing geospatial modelling language can be formalised with Ecore. For this task, the INTERLIS 2 language core is analysed and a meta model developed stepwise. A similar task is performed for the MADS modelling language and the according meta model can be found in Annex C.

Secondly a new modelling language is created to test the possibility of defining smaller, even higher specialised geospatial modelling languages. The presented language is designed on a set of requirements from the HUMBOLDT [Humboldt Consortium, 2011] research project. In chapter 3.3.2 the requirements towards the language and the developed meta model is described.

3.3.1 INTERLIS 2 Meta Model

In this chapter the INTERLIS 2 (see chapter 2.2.2) language elements are analysed and transposed to a formally described abstract syntax in Ecore. Beside the clear textual definitions of INTERLIS 2, also the concrete language syntax as Backus-Naur-Form [KOGIS, 2006b] and the INTERLIS 2 selfdefinition [KOGIS, 2006a] are considered in this step. Both formal language descriptions strongly help in the language analysis and the change to the common formalism of Ecore. This generated abstract syntax or language meta model can be used in the development of software components and for language-based transformations.

INTERLIS 2 has evolved over a long period and can be used to describe several aspects of geospatial information and data exchange. Due to the quantity of the language, only the most commonly used subset, according to the examples provided in [Dorfschmid and Brawer, 2009], is analysed. More specific aspects like the definition of graphical representations or specific object identifiers and the view concepts are not taken into account.

Model The overall root element of an INTERLIS 2 model is the ‘Model’ with a minimal set of metadata, including ‘name’, model ‘owner’, ‘version’ and a marker to indicate that further special contracts are available. There are two specialised types of models possible: pure type definitions or limited
to reference systems. A ‘Model’ might be a translation of another model, which is indicated by the self reference ‘translation’.

Other models can be reused by importing them. The keyword ‘qualified’ is set, if all elements are addressed with their complete name, including model name, topic name and element name.

![Figure 3.3: INTERLIS 2 ‘Model’ Element; abstract syntax defined with Ecore](image)

Figure 3.3 shows one possible meta model fragment of the described INTERLIS 2 ‘Model’ element. Several language properties are directly reflected in the illustrated Ecore model. Obvious are the associations ‘translationOf’, which allows to define one base model in another language, and the association to ‘Import’. The association ‘import’ is resolved as association class, because Ecore only supports simple binary associations without attributes and the ‘qualified’ attribute must be set.

**Topic**  The INTERLIS 2 ‘Topic’ is the language element, which is used for grouping models. It is functionally similar to a UML ‘Package’ and an INTERLIS 2 ‘Model’ can contain one or more ‘Topics’. Despite to UML ‘Packages’, no nesting is allowed, to keep model structures simple and to allow the inheritance of ‘Topics’.

The subset of the designed meta model in figure 3.4 shows the composing relation of ‘Topics’ to ‘Models’, which indicates the containment of topics inside the models. For ‘Topics’ a single inheritance is allowed; this is represented in the [0..1] cardinality of the ‘extends’ composition. The ‘depends’ property of the INTERLIS 2 specification is not included, as it is implicit set via referenced elements within the model.

**Class, Structure and Attribute**  INTERLIS 2 distinguishes between two main modelling elements with similar structure but different interpretation: ‘Class’ for real-world objects (objects with identity) and ‘Structures’ for helping structures, a kind of complex data type for elements without identity.

---

1The ‘qualified’ attribute is not a ‘correct’ part of the conceptual model. The property is mainly a marker for the model parser to interpret the model correctly.
‘Classes’ and ‘Structures’ are primary defined by their name and the list of contained attributes.

The additional property ‘final’ disallows any further extension, whereas ‘abstract’ indicates, that the Class or Structure must be extended and cannot be instantiated directly. The ‘extended’ attribute shows that a Class or a Structure is overwriting the according element with the same name. And the ‘extends’ relationship allows simple inheritance of ‘Classes’ or ‘ComplexTypes’ that keeps all attributes and relationships of the Superclass or Superstructure. As only ‘Class’ instances carry an identity, it is allowed for ‘Classes’ to extend either a Superclass or a Superstructure, but ‘Structures’ may only extend Superstructures.

The ‘extended’ and the ‘extends’ properties are mutual exclusive.

The attributes of ‘Classes’ or ‘Structures’ are primary defined by their name and their types. There is a difference between attributes with a type ‘Structure’ and attributes with types ‘Domain’ or ‘Type’. The latter are limited to single occurrence, while the first ones might also occur as ordered or non-ordered collection (List or Bag). For structure attributes also undefined types are allowed.

The INTERLIS 2 concrete syntax allows the refinement of attribute types within the attribute declaration. In the abstract syntax, these in-lined refinements are assigned to the ‘Type’ and ‘Domain’ definitions and therefore do not occur in the model-subset depicted in Figure 3.5.

**Relationships** Relationships are important in all object-oriented modelling approaches. INTERLIS 2 has got two major groups of relationships:
3.3. META MODELS

Figure 3.5: INTERLIS 2 basic structures of ‘Class’, ‘Structure’ and ‘Attribute’, modelled with Ecore

inherances and associations. The inheritance is already introduced in the previous sections and has got following specific properties in INTERLIS 2:

- The inheritance is in all cases limited to simple inheritance. This limits some specific modelling possibilities, but allows a clear and simple definition of the semantics for inheritances.

- The inheritance of complete ‘Topics’ is allowed in INTERLIS 2; all contained elements of the ‘supertopic’ are available in the topic and can be used, extended or replaced. This is handy for the refinement of a base topic, e.g. when a regional model contains more elements than a national.

- The ‘extends’ property of most language elements offers the overloading or replacement of language elements within an extended topic. The ‘extends’ marks the overwriting of elements for topic inheritances. The ‘extends’ property must not be used together with an ‘extends’ association, as it might cause inheritance conflicts.

Associations in INTERLIS 2 are defined quite complex with several options and possibilities. This complexity is reflecting the difficult real world associations (see esp. [Dorfschmid and Brauer, 2009]).

Because INTERLIS 2 differentiates between ‘Classes’ and ‘Structures’, also the associations need to reflect the difference between the helping structures and the classes with full identity. A ‘ReferenceAttribute’ is a simple association type for ‘Structures’, whereas the ‘Association’ is used for relations between ‘Classes’ and/or ‘Associations’.

A ‘ReferenceAttribute’ has one ‘reference’ to a ‘Class’ or an ‘Association’; this reference can be restricted with the ‘restriction’ member, to indicate, that only the listed sub-classes or sub-associations should be used. If a
CHAPTER 3. CONCEPTS

Figure 3.6: INTERLIS 2 ‘Associations’ and ‘Reference Attributes’, modelled with Ecore

‘Structure’ owns ‘Reference Attributes’, the extension to a ‘Class’ is forbidden, as ‘Reference Attributes’ are undefined for ‘Classes’.

‘Associations’ can carry specific names and have arbitrary attributes. They can define two or more ‘Association Roles’. An ‘Association Role’ carries several properties to define the association behaviour: a ‘name’, the ‘association Type’, the ‘ordered’ property and the ‘association Type’. The ‘association Type’ can be set either to ‘association’, ‘aggregation’ or the strong ‘composition’. Each ‘Association Role’ is associated with at least one ‘Class’ or ‘Associations’ as target element, but also more ‘references’ can be defined as choice. The ‘restricted’ association should be used similar to the ‘Reference Attribute’, where it indicates the allowed subtypes of the ‘references’.

For ‘extended’ ‘Associations’, the ‘Association Roles’ can also be ‘extended’ to refine their targets. If the property ‘hiding’ is set, then the according ‘Association Role’ of the super-association cannot be used anymore, only the specialisation can be applied.

The subset of the meta model including inheritances and the two different associations is depicted in figure 3.6.

Simple Types and Domains Types indicate what kind of values the attributes of ‘Classes’ or ‘Structures’ can carry. While general purpose languages like UML only have very limited numbers of simple types (Boolean, Integer, UnlimitedNatural, String) more specific modelling languages usually offer bigger sets of predefined types. Well defined defined types that are adjusted to the modelling domain can support human usability as well as automated model transformations.

INTERLIS 2 is designed for the exchange of geospatial information; the
3.3. META MODELS

type system is adjusted to real world observations, like measured values with units and precisions, alphanumeric values with defined length or the spatial types points, lines or areas.

In INTERLIS 2 the predefined types are not used directly, but must be concretised within the model; they can be regarded as ‘blueprints’ for the concrete types. Those concrete types are called ‘Domains’ and allow to specify precisely the properties of types like allowed numerical ranges or possible listed values (see the examples in chapter 2.2.2).

Figure 3.7: INTERLIS 2 base ‘Domain’ elements, modelled with Ecore

In Figure 3.7 the basic INTERLIS 2 Types are modelled. The abstract ‘Domain’ offers the ‘final’ and ‘abstract’ marker as well as the ‘mandatory’ value, to indicate if a ‘Domain’ also includes the ‘undefined’ or ‘Null’ value. Considering the fact, that the concrete INTERLIS 2 language allows anonymous domain definitions inside of ‘Classes’ or ‘Structures’, the ‘name’ property is not mandatory.

To define suitable types for modelling, the ‘Texts’, ‘Numerics’, ‘Enumerations’, etc. are inherited from the abstract ‘Domain’ and carry additional properties.

Textual Domains are modelled with ‘MText’ (with control characters) or ‘Text’ (only Unicode characters). If the ‘length’ property is not used, unlimited text length is allowed. Inheritance is defined for textual domains, but only allowing limitations of the text length.

‘Enumerations’ are offering highly sophisticated value lists. Especially the possibility to use ‘Enumerations’ as ‘EnumerationItems’ results in nested lists with tree-like structures. This possibility of choices is modelled in the meta model (see figure 3.7) by the inheritance between ‘EnumerationItem’ and ‘Enumeration’. Extensions of ‘Enumerations’ are possible and can be either restricting or extending. The property ‘circular’ even allows ordered, repetitive enumerations (“Monday”, “Tuesday”, . . . , “Sunday”).

‘Numeric’ domains are defined as decimal values with specific precision and value range. Cyclic domains are important in geospatial and temporal modelling. The property ‘cyclic’ can be used to indicate that after the maximum value of a defined numeric domain, the minimum value would follow.
Beside simple numeric domains, it is possible to indicate measured values by adding a reference to a ‘Unit’.

‘Units’ as depicted in figure 3.8 bridge numeric values to real world measures. In practice especially abstract units are important to define the general types of measurement; e.g. the Length can be defined as abstract unit and Meters extend Length. The derived ‘ComposedUnit’ and the ‘DerivedUnit’ allows direct relationships between different ‘Units’. Surprisingly INTERLIS 2 does not offer reference systems for numeric values on language level, this is shifted to standardised packages on the model level [KOGIS, 2006b].

![Figure 3.8: INTERLIS 2 ‘Units’, modelled with Ecore](image)

**Spatial Types and Domains** The spatial data types and domains are important for spatial modelling; INTERLIS 2 offers the same concept of ‘blueprints’ for well-defined types. ‘Coordinate’ Types are used to describe simple points, ‘Line’ Types for any linear objects and ‘Area’ or ‘Surface’ for polygons.

The ‘Coordinate’ is defined as ordered List of axis, which references to the ‘Numeric’ domains and opens the coordinate space. The concrete syntax of INTERLIS 2 only allows in-line definition of numeric domains for the axis. For that reason, the axis are reflected in the meta model (see figure 3.9) as composition indicating that the used ‘Numeric’ types are solely used by that ‘Coordinate’ type.

‘Lines’ and the polygon types ‘Area’ or ‘Surface’ are characterised by the coordinates of their vertices (‘coord’) and the shape of the line between the vertices (‘lineform’). Similar to the definition of the reference systems, the lineform must be defined on model level. The most common forms like straights, circles or splines are provided within the base INTERLIS 2 packages [KOGIS, 2006b].

---

2 For more complex relationships between units, the INTERLIS 2 element ‘Function’ must be used.
3.3. MET MODELS

3.3.2 Domain-Specific Meta Model

Following the basic idea that the new language oriented approaches would allow the easy creation of highly specialised modelling languages without losing interoperability, this chapter describes the creation of a new modelling language.

Highly specialised languages can be efficiently adjusted to the intended user communities, collect exactly the required information and present it in adequate form. The modelling language can be clean from superfluous elements and avoid complex constructs. A suitable language could guide to clean, compact and precise models.

If those languages are defined with according processable meta languages, user-friendly tools can be derived, including graphical or textual modelling environments, parsers and validators to ensure model quality.

In this work, an exemplary modelling language for the European HUMBOLDT project [Humboldt Consortium, 2011], the 'HUMBOLDT Modelling Language (HML)', is developed. The HUMBOLDT project aims at providing tools for the harmonisation of geospatial data in the context of the initiatives of INSPIRE and GMES [European Commission, 2011]. One key element of the project is the harmonisation and the transformation of data in different conceptual models. Well known conceptual models of sources and targets are requested as requirement for the definition of those transformations. The newly defined specialised modelling language, which is elaborated
in this chapter and the derived tools help in the specification of such models. Parsers and checkers ensure basic quality and produced code fragments can directly be utilised in the developed transformation software.

Requirements

To develop suitable modelling languages, the requirements towards that language must be clarified. Those requirements lead to the definition of the language properties, the needed modelling elements, the requested types and the used terminology. For the HUMBOLDT modelling language HML, the requirements result from a workshop on HUMBOLDT data models held Technical University of Delft 2008 and the analysis of the data specifications of the HUMBOLDT scenarios [Humboldt Consortium, 2011]. Without doing in depth analysis, which would be beyond the scope of this thesis, the requirements can be grouped according to their main origin or functionality:

- Requirements from the INSPIRE modelling approach, which is primarily based on the ISO/TC-211 standards. Mainly the General Feature Model [ISO/TC-211, 2005a] and the spatial data types [ISO/TC-211, 2005b] should be considered to avoid conflicts between existing and newly generated models.

- Requirements for data processing for harmonisation as investigated in the HUMBOLDT project. Those requirements include the wish to reduce the flexibility of modelling in senses of offering only limited modelling alternatives for most real world situations. This results in more comparable conceptual models that could be processed easily in the harmonisation steps. Specially requested features are elements for reference systems, units and minimum meta data about the model.

- Requirements from the user perspective. The modelling language should also simplify the modelling process, offering easy-to-use environment with validation support and good visualisation [Khatri et al., 2006].

Developed Language Properties. From the collected requirements towards the new HML, the basic language properties, the main elements and the basic terminology are developed. Considering also the established languages UML and INTERLIS 2, with their commonly accepted concepts and established structures, following properties are defined:

1. Using an object-oriented approach to reflect real world objects: The object-oriented approach should be used in compliance to the general feature model of ISO/TC-211. The main element should be a Class (Feature Type) representing descriptions of real world entities. Those classes should have an identity, i.e. the objects of those classes
exist and can be addressed individually; they have a name, attributes and relations to other objects. As the modelling is focussed on the data perspective, the definition of methods is superfluous.

2. **Simple Inheritance:** Although in real world modelling multiple inheritances often occur, they should be avoided due to their complexity. Targeting the simplification of modelling, only simple inheritances are allowed.

Examples for multiple inheritances are frequent; a *bridge* is a part of a *street* and a *building*. Depending on the definitions of *street* and *building* this might cause problems in implementation. Is a *name* attribute defined for *streets* and for *buildings*, it is not clear how this property will be inherited. For that reason multiple inheritances are often avoided. In the example *bridge* might inherit from *building* and be connected to *streets* with an association *connectedStreet*.

3. **Definition of Attributes:** Attributes are used to describe the specific properties of objects. The definition of those attributes is done in the according classes. Following common practise in object-oriented modelling, attributes should have name, a type and cardinality. In several cases, the order of attributes is of relevance.

4. **Complex Types to describe repetitive patterns:** Often several classes carry similar attribute groups, e.g. addresses. To simplify maintenance and reduce model complexity, it is allowed to define complex types. Those can be used like normal types and are defined similar to classes, but do not have identities. To avoid confusions, inheritances of complex types and classes are separated. This means classes can only be derived from classes and complex types can only be derived from other complex types.

Depending on the perspective, the attributes *streetname*, *housenumber* and *city* can be guessed as ‘real world’ class *AddressClass* or as a simple collection of attributes *AddressComplexType* that is only used within other ‘real world’ classes like *House* or *Customer*. Obviously both representations should not be mixed, as the first *AddressClass* has an identity, whereas the *AddressComplexType* has not. This differentiation must remain also for subtypes and subclasses. There can be rules to avoid such problems, but preferable Classes and ComplexTypes should not be mixed in their inheritance hierarchies.

5. **Simple Associations:** Associations describe the relations between different objects. Although it is a key feature of several modelling languages, more advanced associations with attributes and inheritances are rarely used in practise. The common workaround to handle more complex associations is to define (virtual) association classes. As the
modelling approach aims at simple models and user interfaces with limited sets of elements, this approach should be used.

A common example of a more complex association is a sells association between two objects of the class Person and one object of the class Product with an attribute price. This is a ternary association with the roles seller, buyer and item. Most likely it may be useful to subtype this association; e.g. bidding with more persons involved etc.. The common way to limit the complexity is to create a helping or association class purchase (see [Poels et al., 2005]). From a conceptual point of view this is a workaround, but very common in practical work, because it reduces the number of modelling elements as well as it does limit the variability of models.

6. **Association Strength:** The strength of associations is used to indicate how dependent related objects are. The range should be according to common practise from association over aggregation to composition. This classification of associations is of relevance for the understanding of conceptual structures as well as for the maintenance of data.

7. **Support for Type Definition:** The knowledge of the value domain for attributes is substantial for the implementation of database structures or exchange formats. Several modelling approaches exclude the setting of value domains from conceptual modelling [Balzert, 1995] and regard it to be part of logical or physical modelling. But the knowledge of typical values and value ranges is part of the knowledge of domain specialists. For that reason value domains should be modelled on the conceptual level and the language should support this; e.g. an air temperature measurement should not be expressed as Java ‘Float’ Data Type ranging from $-3.4028235\times10^{38}$ to $3.4028235\times10^{38}$ but as a Celsius Temperature ranging from $-45^\circ C$ to $60^\circ C$. The requested type-system should be flexible enough to reflect such experts’ knowledge. For implementation-independent modelling, the following basic types must be supported: Boolean, Numeric Values (with definable ranges and precisions), Strings (with definable length) and Enumerations (with definable values).

Additional to those basic types, in geospatial environments often measured values are to be supported. So numeric types can be related to units of measurement and to reference systems (with associated axis)\(^3\).

Finally it happens quite often, that specific attributes can either have one type or another, e.g. a value expressed as measured value or as classified value from an enumeration. The modelling language must support these Union types, expressing structural choices.

\(^3\)Aligned with the concepts of INTERLIS 2 domain definitions.
8. **Spatial Data Types**: Key elements for spatial data modelling are of course the spatial data types. In many applications only essential types are needed; point, line as well as polygon types with and without overlaps. Following the concept of refining data types with domain experts’ knowledge, it must be possible to express those types with settings for precision, coordinate value domains or allowed reference systems.

9. **Packaging**: A simple packaging concept should be available to structure models and keeping them readable. Packages should allow nesting of packages and define a ‘namespaces’ concept. This means that the complete name of an element is the combination of package and element name, so within a model elements might have the same name, as long as they are located in different packages.

10. **Model as root element**: The reusability of elements from different models is an essential requirement in modern data modelling to achieve interoperable data models. It is important to have a well defined root or top-level element to organise the interaction between models. The model as top level element should be used for maintaining the models and should therefore carry additional maintaining information like actual model version or model creator.

**Language Metamodel**

Following the defined properties for HML, an abstract language syntax is defined and explained in this chapter. The definition follows a stepwise development process, starting with the structural elements for model and packages, continuing with class definitions including attributes and relationships. Finally the type system for simple and spatial types is explained. Again the Ecore metamodel approach is used for this abstract language syntax. The complete meta model can be found in Annex B.

**Model** According to the need of a defined root element for conceptual models, the ‘Model’ element is defined (see figure 3.10). It contains the metadata elements to describe information regarding maintenance and management of the model. In the described context of the Humboldt project, those elements are limited to ‘name’, ‘version’ and ‘creator’ as free text and a ‘timestamp’.

**Package** As formulated in language properties, only a simple, nesting package structure is needed. It is prescribed, that each model contains exactly one root-package and an arbitrary number of nested packages with arbitrary depth.
Figure 3.10: Humboldt Modelling Language elements ‘Model’ and ‘Package’ elements, modelled in Ecore.

Figure 3.10 illustrates how these properties are reflected in the abstract syntax. A HML ‘Model’ contains the properties ‘name’ (from ‘ModelElement’), ‘creator’, ‘version’ and ‘timestamp’. For each ‘Model’ exactly one ‘modelpackage’ is specified, that may contain an arbitrary number of ‘sub-packages’. Except for their ‘name’ ‘Packages’ carry no properties.

Class, Complex Type and Attribute  Similar to INTERLIS 2 and to the concepts of the General Feature Model (ISO/TC-211 2005) there is a differentiation between classes for objects with identity and complex types that only act as structures for attribute collections.

Figure 3.11: ‘Classes’, ‘ComplexTypes’ and ‘Attributes’ of the HML meta model, expressed in Ecore.

The ‘Class’ is a named element and contains a set of attributes, with each of the attributes having a ‘name’, a defined ‘cardinality’ and an indicator,
if the attribute values are ‘ordered’ in cases of higher cardinality. Every attribute is further characterised by its ‘type’. A type might be simple, a geometric type or a composed `ComplexType` type.

As defined in the language properties, inheritance is only defined as simple inheritance within ‘ComplexTypes’ or ‘Classes’. Those inheritances for ‘ComplexTypes’ and ‘Classes’ are defined in the meta model as simple self-reference ‘superclass’ or ‘supertype’ with a limited cardinality of zero to one (see figure 3.11).

**Relationships** The modelling of relationships is simplified according to common modelling practise. While some conceptual modelling languages, in particular INTERLIS 2, offer complex approaches, like multiple association ends, relationship attributes and inheritances, the HML approach limits it to simple binary relationships.

‘Relations’ are defined by their association ends, ‘fromClass’ and ‘toClass’, the according ‘role-names’ and their ‘cardinalities’. An additional name can be omitted, because the role-names describe the association sufficiently. As it is required to differentiate the strengths of relations, the ‘Relation’ element is declared abstract and only the inherited elements ‘Association’ for loosely coupled elements, the ‘Aggregation’ for strong relationships and the ‘Composition’ for dependent relations can be used (see figure 3.12).

| Figure 3.12: HML meta model; different relationships modelled with Ecore. |

**Simple Types** As already described in chapter 2.2 the definition of types is especially important for domain-specific modelling languages. In the definition of the language properties for the Humboldt Modelling Language it is stated that the type-system should not be determined by the underlying implementation, but reflect the expert’s knowledge of type and domain of the data. This is covered by a system of templates, which allows the users to define the types precisely instead of matching their needs to a fixed typeset.
Figure 3.13: Simple Types for the HML meta model; modelled with Ecore.
All templates for the definition of such simple types inherit from the ‘SimpleType’ element (see figure 3.13). It offers the properties ‘name’, ‘abstract’ and ‘base’ inherited from ‘DefinitionElement’ and owns the property ‘definition’. The ‘definition’ property was designed as simple textual element to allow users storing additional context information or describing the types.

Structurally simple is the definition of ‘StringTypes’. Beside the properties of the ‘SimpleTypes’ a ‘StringType’ is only defined by the potential length of strings. ‘StringTypes’ can also inherit from other ‘StringTypes’ by defining their ‘supertype’ relationship. The inheritance between ‘StringTypes’ was designed to be either restricting in length or deriving a concrete type from an abstract.

The ‘EnumerationType’ was mainly defined as container for ‘EnumerationItems’. An ‘EnumerationItem’ is a simple textual entry. There are two possibilities of inheriting enumerations; restricting the ‘EnumerationItems’ to a subset of the items or define a new ‘Enumeration’ by merging enumeration items. Both possibilities are used quite frequently and therefore are included in the metamodel.

The ‘NumericType’ offers several possibilities for defining the concrete numerical types. Beside the inherited properties of ‘SimpleType’, the ‘length’, ‘precision’, ‘min’ and ‘max’ allow the definition of value ranges and precisions. The Boolean property ‘cyclic’ is used to indicate repetitive value ranges. This is very important for temporal and several geographical values.

According to the main usages of numeric types for modelling real world observations, an optional ‘Unit’ can be referenced by the numeric type.

Inheritance is indicated by a ‘supertype’ relation and should be restrictive in senses of reducing the domain or precision, limiting to inherited units or simply deriving concrete subtypes from abstract ones.

The ‘Unit’ element is exclusively used in the context of numeric types for measurements. Additionally to the derived properties from the ‘DefinitionElement’ there is ‘shortUnit’ for the standard abbreviation of a unit and ‘definition’ for additional user information. Units can also be inherited using a ‘superunit’ relationship. This allows the creation of unit systems including abstract units for e.g. ‘length’ or ‘time’ and their concrete units like ‘metres’, ‘inches’, ‘seconds’ or ‘years’.

**Union Type** In real world modelling a very common situation is that certain properties can be expressed as either one type or another. To allow such choices, the ‘UnionType’ is defined as a specialisation of ‘Type’, which acts as collection of references (‘unionTypes’) to other types. The inheritance of ‘UnionTypes’ is again defined to be simple and restrictive in reducing the associated types (see figure 3.14).

---

4This definition of how an inheritance is used, is part of the semantics and cannot be reflected in the abstract syntax.
Spatial Types  The logical structure of the spatial type model is split into two fields; on one hand the different ‘GeometryTypes’ and on the other hand elements dealing with spatial referencing. As described in the language properties, the HUMBOLDT Modelling Language is limited to the three basic geometry types. Additionally to that, it is possible to set spatial reference systems for the geometries.

In figure 3.15 the according subset of the developed Humboldt Modelling Language is illustrated. The abstract ‘GeometryType’ is the root for all geometry types and the inherited ‘SimpleGeometryTypes’ cover ‘PointType’, ‘LineType’ and ‘PolygonType’. They only own the inherited properties ‘name’, ‘base’ and ‘abstract’ from the general ‘DefinitionElement’ but are connected to indicate their topological relations. For the specific purpose of this language, no further properties on the spatial types are required, although several extensions might be possible, e.g. for all PointTypes indicating minimum spatial resolutions, different geometric forms for the LineTypes like arcs, splines or polynomial functions and for PolygonTypes maximum overlapping, minimum areas or indicators for space-filling of the polygons.

Each of the ‘SimpleGeometryTypes’ allows simple inheritance in restricting its supertype. A PolygonType can only inherit from a PolygonType with the same or a subtyped LineType, a LineType can only inherit from a LineType with the same or a subtyped PointType etc. Additionally to the ‘SimpleGeometryTypes’ there is also a ‘MultiGeometryType’ defined for models handling spatial attributes that might contain different simple geometry types; e.g. a building represented as simple point or polygon, but it is also possible to refer to points either in one or the other reference system.

The set of elements defined for the spatial referencing is based on a simplified, but for this application sufficient model. A ‘Coordiatesystem’ is defined by the general inherited properties of the ‘DefinitionElement’ and an additional ‘definition’ property to support human understanding of the
element. But structurally it is defined by the (up to three) axis that are opening the coordinate space. No additional properties are included into the ‘Coordinatesystem’, because the definition of spatial systems is usually not of interest for the modelling experts.

The ‘Coordinate’ element includes also references for up to three ‘CoordinateNumericTypes’, which are representing the coordinate values. The ‘axis’ relationship of a ‘CoordinateNumericType’ offers the possibility to bind the coordinate values to the according axes of ‘Coordinatesystem’. By bridging the ‘CoordinateNumericType’ to the ‘Axis’ instead of referring the ‘Coordinate’ to one according ‘Coordinatesystems’, it is possible to allow Coordinates with mixed reference systems. E.g. a three-dimensional coordinate can contain two coordinate values of a 2-d ‘Coordinatesystem’ and the height component from a 1-dimensional height system.

![Figure 3.15: Spatial Types for HML; modelled with Ecore (to simplify, the following inheritances are omitted: From ‘Type’ to ‘GeometryType’; from ‘SimpleType’ to ‘NumericType’ and from the abstract ‘DefinitionElement’ to ‘Unit’, ‘Coordinate’, ‘CoordinateSystem’ and ‘Axis’)]

**Standard Type Libraries** As described in previous sections the developed HUMBOLDT modelling language does not contain a fixed set of primitives, but a system to declare types independently of the underlying implementations. Nevertheless it is handy for several purposes to have a set of standard types to reduce efforts in definitions and increase interoperability
between models.

For the given purpose a simple type library is set up with packages for standard types, for common units and for time specific elements. The standard types (figure 3.16) include the enumeration type for ‘Boolean’, ready to use ‘Integer’ and ‘Float’ types as well as string types expressing ‘Name’ and ‘URI’. Additionally there is a standard type introduced to indicate object identifiers ‘OID’. The object identifier is designed as a union type pointing to three standard implementations: ‘StandardOID’, which is a simple text identifier; the ‘UUIDOID’ also a string identifier marking a UUID [IETF - Internet Engineering Task Force, 2009] and a simple integer identifier ‘IntegerOID’. To offer at least the minimum geospatial types, simple types for ‘Point’, ‘Line’ and ‘Polygon’ were created. Those abstract spatial primitives carry no information on extents, used reference systems or numeric types and can only be used as place holder and root element for an inheritance hierarchy.

![Figure 3.16: Standard predefined types for HML (Graphic created with the HUMBOLDT GeoModel Editor as described in chapter 4.2.5)](image)

The ‘BaseUnits’ package (figure 3.17) contains some often used units. The concept is always to define an abstract unit first that indicates what to measure and a concrete unit derived from that. For the first basis, those units are mainly defined by the international system of units [Bureau International des Poids et Mesures, 2008]. Further commonly used units are defined in the ‘ExtendedUnits’ package, that contains ‘Volume’, ‘Area’, ‘Speed’, ‘Pressure’ etc. as well as further frequently used units like ‘Kilometre’, ‘Degree Celsius’
3.3. META MODELS

and ‘Degree’ for the measurement of angles.

Figure 3.17: Base Units defined in HML (Graphic created with the Humboldt GeoModel Editor as described in chapter 4.2.5)

3.3.3 Results

INTERLIS

The abstract syntax for the chosen subset of INTERLIS 2 can be derived easily from the descriptive language reference (the complete meta model can be found in Annex A). Most specific properties of INTERLIS 2 are reflected well in the meta model: e.g. the differentiation between Structures and Classes, with their different relationships; the concept of defining domains on model level and the adjustment to real-world phenomena with coordinates, measurements and typical enumerations or the handy definition of packaging and the model inheritances.

One language aspect cannot be reflected well in the meta model. INTERLIS 2 includes the possibility to handle language extensions on the model level⁵. This is e.g. used for the definitions of reference systems or line forms. An inclusion of those extendable elements into the meta model of INTERLIS 2 would be more consequent and probably more easy to use within software tools and transformations.

Finally it must be noted, that the presented meta-model is only one possible representation of the abstract language syntax of INTERLIS 2.

---

⁵Similar to UML stereotypes
Domain-Specific Language HML

For a given set of requirements, like those from the HUMBOLDT project, a specialised domain specific geospatial modelling language can be easily designed. With the use of eMOF (Ecore) as standard meta-modelling approach, all requirements could be reflected well and with little effort in the language.

For the designed language some concepts of INTERLIS 2 and ISO/TC-211 were utilised. This follows the best-of-breed ideas of domain-specific languages: Optimising existing languages for specific purposes, so that users can rely on established approaches with common terminology where they fit and on modifications where necessary.

The developed language consists of a package model with nesting, clearly differentiated ‘Classes’ with identity and ‘ComplexTypes’ as structure. Inheritances and Associations are limited in possibilities and a flexible, but precise typesystem is developed. Beyond the language specification, a base library for commonly used types and units is introduced.

3.4 Horizontal Transformations between Modelling Languages

Following the second working thesis (‘Horizontal transformations between models described in different modelling languages can be defined with standard transformation tools’) the usage of meta models can simplify the translation between models of different modelling languages. With the abstract syntax as formal meta models defined in the same metamodel approach (Ecore), the definition and execution of horizontal language-to-language transformations can be simplified. Compared to transformations on the model representation (the ‘model files’), no specific efforts for language parsing or serialisation are needed and mapping definitions can be supported by user friendly tools.

The reuse of existing components, like GML Schema generation or model validation procedures, can be enhanced, because the models will be translated to the according model language. If a domain-specific geospatial modelling language could be translated to ISO/TC-211 compliant UML, the existing procedures can be reused.

For the transformation between languages different typical situations can occur, dependent on the grade of similarity between the language concepts:

- **Direct 1:1 mapping:** The elements of the source language can be directly translated to according elements in the target language; e.g. language element ‘Class’ in the Humboldt Modelling Language can be directly translated to UML ‘Class’ elements with the ISO/TC-211
3.4. **HORIZONTAL TRANSFORMATIONS**

stereotype «FeatureType». Such direct mappings indicate the same concepts of the languages.

- **Filtered 1:1 mapping**: The elements of the source language can be mapped to according elements of the target language according to source’s attribute values; e.g. the language element ‘NumericType’ in the HUMBOLDT modelling language can be mapped either to the UML ‘Integer’ or ‘Unlimited Natural’ depending on the property ‘precision’. The selection of the correct target type is a simple filtering on the property values. Those filtered mappings indicate similar concepts, but differ in coding information either as property or as different types.

- **Mapping of patterns**: There is no direct mapping from one source language element to one language element of the target language, but language elements or specific patterns in the source mode can be mapped to other patterns in the target model. Such restructuring transformations can be found when language concepts are only available in one language and must be substituted by a structure in the target model; e.g. INTERLIS complex associations must be mapped to association classes in UML. Those pattern mappings indicate different modelling concepts of source and target language and especially appear in mappings towards general purpose modelling languages.

- **Mapping between language elements and standard model elements**: Language elements from the source model are mapped to specific predefined elements of the target model; e.g. the language element ‘LineType’ of the defined HUMBOLDT modelling language is mapped to the imported «Datatypes» ‘GM_Curve’ or ‘GM_MultiCurve’, that are defined in ISO-19107 [ISO/TC-211, 2001]. Similar to the mapping of model patterns, these mappings occur, when general purpose modelling approaches are enriched with additional concepts in form of standard libraries.

- **Mapping not possible (information cannot be represented accordingly)**: Some information cannot directly be mapped to another modelling language; e.g. Inheritances and Merging of ‘Enumerations’ in HML cannot be reflected in the ISO /TC-211 approach for UML modelling, as there is no defined way of handling inheritances for classes with «Enumeration» stereotype. In that example all ‘Enumeration Items’ can be retrieved, but the knowledge on the relations between enumerations is lost after the transformation. Those information losses are frequent if source and target languages are different in their modelling scope.
3.4.1 Transforming between Domain-Specific Language HML and UML

To examine the feasibility of transformations based on a meta model approach, a mapping from the defined domain-specific language HML to the most frequently used standard approach of ISO/TC-211 is analysed. This transformation is chosen as example to test how highly adapted modelling languages fit into common developments for geospatial modelling.

For the description of the mappings between source and target language a simplified pseudo-code with similarities to QVT [OMG - Object Management Group, 2007c] is used. The source modelling language as defined in chapter 3.3.2 is noted with the prefix `HML', the target language is Unified Modelling Language ‘UML’ with the stereotypes used by ISO /TC-2116.

In chapter 4.3 the pseudo-code of this chapter is translated into a concrete mapping language as proof of feasibility.

**HML Model** The ‘Model’ element in HML is used as root element for the handling of imports and to add meta-information to the model. This information is currently limited to ‘name’, ‘creator’, ‘version’ and ‘timestamp’.

Such close integration of meta data of the models is not foreseen in the ISO/TC-211. Meta information can only be added on the level of implementation in form of ISO-19115 [ISO/TC-211, 2003] or Dublin Core [Dublin Core Metadata Initiative, 2010] meta data. For that reason, no mapping from HML ‘Model’ to UML is defined and it is sufficient to use the HML ‘modelPackage’ as input for the UML root element (listing 3.1).

```plaintext
// No Transformation for HML Model
HML_Model -> null
```

Listing 3.1: Pseudocode for the ‘Null’-transformation of HML Model

**HML Package** The HML Package offers functionality to structure elements. It can be nested, defines closed namespaces and acts as a container for all other modelling elements. The functionality is similar to UML, except for the HML root package, which is of the same functionality as the UML Model.

The properties ‘name’ of the HML Package and the UML Package are the same and can be directly mapped; the similar relations for nesting Packages are called ‘subpackages’ for HML and ‘nestedPackage’ for UML (see listing 3.2).

\[6\] In this pseudo-code the complexity of handling UML Stereotypes is not taken into consideration. Due to the definition, Stereotypes can only be applied on existing UML elements. This means, the creation of stereotyped model elements must be a two phase process: first creating an element with all required properties in the profiled model and secondly applying the stereotype and tagged values to the element.
3.4. HORIZONTAL TRANSFORMATIONS

The ISO/TC-211 approach overloads the structuring pattern of packages with the stereotypes «ApplicationSchema» and «Leaf». «ApplicationSchema» indicates a thematically closed subset of models and «Leaf» is used for simple, unnested definition packages. In common practice, «Leaf» is rarely used for domain models and a similar element is not present in the Humboldt approach.

Summarised, HML root packages are mapped to UML models, the top level packages to packages with the stereotype «ApplicationSchema» and all deeper nested packages are mapped to simple UML packages.

```java
// HML Model package transformed to UML Model
IF (HML_Package == HML_Model.modelPackage) THEN {
    HML_Package -> UML_Model
    (HML_Package.name -> UML_Model.name,
     HML_Package.subpackages -> UML_Model.subpackages)
} // HML Top Level Packages transformed to UML Packages

ELSE IF (HML_Package IS MEMBER OF HML_Model.modelPackage.subpackages) {
    HML_Package -> UML_Package <<ApplicationSchema>>
    (HML_Package.name -> UML_Package.name,
     HML_Package.subpackages -> UML_Package.subpackages)
} // Other HML Packages mapped to UML Packages

ELSE {
    HML_Package -> UML_Package
    (HML_Package.name -> UML_Package.name,
     HML_Package.subpackages -> UML_Package.subpackages)
}
```

Listing 3.2: Pseudocode for the transformation of HML Packages to UML Model and UML Packages

**HML Classes, Complex Types and Attributes**

The domain-specific language HML and the ISO/TC-211 modelling approach with UML use similar concepts for modelling features and complex structures, although HML is more restrictive in its possibilities. Due to the modelling focus on real world situations, both approaches have the differentiation between helping structures (HML ComplexTypes, ISO/TC-211 DataTypes) and real-world representations (HML Classes, ISO/TC-211 FeatureTypes).

The transformation in listing 3.3 from HML Classes to UML Class with ISO/TC-211 stereotype «FeatureType» is straightforward. The properties 'name' and 'abstract' are mapped to the according counterparts. The HML 'superclass' is transformed to one entry in the generalisation list 'general' in UML. The members of the HML Class 'attributes' can be directly mapped to the UML Class attributes, but each HML 'Attribute' must be transformed to one UML 'Property'.

This transformation is also simple: HML Attribute 'name' is mapped to UML Property 'name', the 'minCardinality' and 'maxCardinality' to 'lower'
and ‘upper’, the ‘type’ property is mapped to ‘type’. The mapping from HML types to UML types must be handled in a dedicated way as described later.

```
HML_Class -> UML_Class <<FeatureType>>
(HML_Class.name -> UML_Class.name,
HML_Class.abstract -> UML_Class.abstract,
HML_Class.superclass -> UML_Class.generation[1]
HML_Class.attributes[] -> UML_Class.attributes[] {
    HML_Attribute -> UML_Property
    (HML_Attribute.name -> UML_Property.name,
    HML_Attribute.minCardinality -> UML_Property.lower,
    HML_Attribute.maxCardinality -> UML_Property.upper,
    HML_Attribute.type -> UML_Property.type)
})
```

**Listing 3.3:** Pseudocode for the transformation of HML Class to UML Class with «FeatureType» stereotype

The transformation of HML ComplexType to UML Class with according stereotype «DataType» is similar to the described transformation of HML Class to UML Class with stereotype «FeatureType» as shown in listing 3.4.

```
HML_ComplexType -> UML_Class <<DataType>>
(HML_ComplexType.name -> UML_Class.name,
HML_ComplexType.abstract -> UML_Class.abstract,
HML_ComplexType.supertypes -> UML_Class.generation[1]
HML_ComplexType.attributes[] -> UML_Class.attributes[] {
    HML_Attribute -> UML_Property
    (HML_Attribute.name -> UML_Property.name,
    HML_Attribute.minCardinality -> UML_Property.lower,
    HML_Attribute.maxCardinality -> UML_Property.upper,
    HML_Attribute.type -> UML_Property.type)
})
```

**Listing 3.4:** Pseudocode for the transformation of HML ComplexType to UML Class with «DataType» stereotype

**HML Relations**  The relations within the Humboldt Modelling Language are designed simple and restrictive compared to UML relations. The limitations led to easy transformations from HML to UML.

In listing 3.5 HML ‘Associations’, ‘Aggregations’ and ‘Compositions’ are transformed to UML Associations with the property ‘name’ mapped directly. As UML ‘Associations’ are build from collections of association ends, the HML ‘fromRole’, ‘fromCardinalityMin’ and ‘fromCardinalityMax’ properties must be transformed to a UML AssociationEnd with the according properties for ‘name’, ‘lower’ and ‘upper’. The ‘to...’ properties are mapped similarly. The strength of the relations, expressed in the HML inheritance hierarchy is translated to the values ‘none’, ‘shared’ and ‘composite’ for the property ‘aggregation’ of the first AssociationEnd.
3.4. HORIZONTAL TRANSFORMATIONS

Listing 3.5: Pseudocode for the transformation of HML Association, Aggregation and Composition to UML Associations with different association ends.

HML Simple Types The handling of types strongly differs between the Humboldt Modelling Language and the ISO/TC-211 approach of using UML. HML does not offer any standardised types, but provides templates to define them. ISO/TC-211 uses the standard UML primitives (Boolean, Unlimited natural, Integer and String) and a huge set of predefined classes and types (esp. from Spatial Schema [ISO/TC-211, 2001]). Due to this difference the mapping between HML types and according UML types needs detailed treatment.
For ‘String’ types, the HML modelling language offers the possibility to limit the string length and also introduces inheritance. The ISO/TC-211 modelling does not allow those possibilities\(^7\), so that all HML string types are directly mapped to the UML String primitive (listing 3.6).

\[
\text{HML\_StringType} \rightarrow \text{UML\_Primitive\_String}
\]

Listing 3.6: Pseudocode for the transformation of HML String Types to UML String primitives

The ‘Numeric Type’ in HML is designed to reflect as good as possible real world usages of numerical elements with potential units, value ranges and precisions. The transformation targets are the UML primitives ‘integer’ or ‘unlimited natural’ and the ‘GML Measure Types’ or its subtypes for numeric values with units of measure. The code for the decisions can be found in listing 3.7.

```
IF (HML\_NumericType. unit == NULL)
THEN {
  IF (HML\_NumericType. precision == 0)
  THEN {
    HML\_NumericType \rightarrow \text{UML\_Primitive\_Integer}
  }
  ELSE {
    HML\_NumericType \rightarrow \text{UML\_Primitive\_UnlimitedNatural}
  }
}
ELSE {
  IF (HML\_NumericType. unit == 'Length' or 
      HML\_NumericType. unit. anySuperUnit () == 'Length')
  THEN {
    HML\_NumericType \rightarrow \text{ISO\_19103\_LengthType}
  }
ELSEIF (HML\_NumericType. unit == 'Time' or 
         HML\_NumericType. unit. anySuperUnit () == 'Time')
  THEN {
    HML\_NumericType \rightarrow \text{ISO\_19103\_TimeType}
  }
ELSE {
    HML\_NumericType \rightarrow \text{ISO\_19103\_MeasureType}
  }
```

Listing 3.7: Pseudocode for the transformation of HML numeric types to UML primitives or ISO/TC-211 MeasureType

The simple HML ‘Numeric Types’ are mapped to the UML primitive ‘integer’ if the precision property is zero, otherwise it is mapped to ‘unlimited natural’. In both cases the information on value ranges, type inheritance and potential cyclic behaviour is lost.

HML numeric types with related units are mapped by default to the ISO-19103 Measure type. If the related HML unit is taken or inherited from the

\(^7\)The only possibility to restrict String length or value ranges for UML would be the usage of OCL constraints.
provided HML standard library, a deeper mapping can be provided; numeric
types referring to any length unit are matched to an ISO-19103 Length type,
HML numeric types with angle units will be matched to an ISO-19103 Angle
type, etc.

**HML Enumeration Types**  The HML Enumerations are simply offering
a controlled list of text enumeration items. There are two types of inher-
tances allowed, one extending the enumeration items and one restricting
them. The ISO/TC-211 approach uses two different elements for similar
lists: «Enumerations» stereotypes are used for closed collections of values;
«CodeList» stereotypes are similar, but the items may change during model
maintenance. For both types no inheritance or extensions are allowed.

Such differentiation on maintenance strategy is not part of the Humboldt
Modelling Language, so all HML ‘Enumerations’ are translated to UML
‘Classes’ with the «Enumeration» stereotype. Furthermore, the enumeration
items of the HML inheritance hierarchy must be resolved to retrieve all
equivalent members for the according UML enumeration (listing 3.8).

All HML ‘Enumerations’ with the name “Boolean” are matched to the
Boolean primitive in UML.

```plaintext
IF (HMLEnumeration.name == 'Boolean')
 THEN {
 HMLEnumeration -> UMLPrimitive_Boolean
}
ELSE {
 HMLEnumeration -> UMLClass <<Enumeration>>
 (HMLEnumeration.name -> UMLClass.name,
 HMLEnumeration.items[] -> UMLClass.attributes[])
 (HMLEnumItem.text -> UMLProperty.name)
 // also add the items for the merge enumerations
 HMLEnumeration.merge[].items[] -> UMLClass.attributes[]
 (HMLEnumItem.text -> UMLProperty.name)
}
```

Listing 3.8: Pseudocode for the transformation of HML Enumerations to
UML Classes with stereotype «Enumerations»

**HML Union Types**  The HML UnionTypes are used to express exclusive
choices of types, which are defined by the relation ‘union type’. In ISO/TC-
211 a stereotype «Union» is used to mark standard classes as union types.
The allowed types are to be inserted as attributes with arbitrary names and
the correct type\(^8\).

\(^8\)In practise the «Unions» stereotype is rarely used. Mostly the choices are moved to
the level of attributes in the feature types.
In listing 3.9 the HML UnionType is therefore mapped to a UML class with stereotype «Union». The relations to the unionTypes are resolved as simple attributes. The attribute name is arbitrary chosen as the lower camel case name of the according type and the attribute type is mapped to the correct type.

```plaintext
HML_UnionType -> UML_Class <<Union>>
(HML_UnionType.unionTypes[]) -> UML_Class.attributes[] {
  HML_Type -> UML_Property
  (HML_Type.name.toLowerCaseCamelCase()) ->
  UML_Property.name,
  HML_Type -> UML_Property.type}
```

Listing 3.9: Pseudo code for the transformation of HML UnionTypes to UML Classes stereotyped with «Union»

**HML Spatial Types**  The rich spatial types are a key characteristics of the Humboldt modelling language. Compared to ISO/TC-211, where predefined types of ISO-19107 must be used, HML offers a rich content with the possibilities of defining reference systems and value domains.

The resulting transformations are simple but with a notable information loss (listing 3.10). HML PointTypes are mapped to ISO-19107 ‘GM Point’. HML LineTypes or HML PolygonTypes are mapped to ‘GM Curve’ and ‘GM Surface’.

```plaintext
HML_PointType -> ISO-19107_GMPoint
HML_LineType -> ISO-19107_GMCurve
HML_PolygonType -> ISO-19107_GMSurface
```

Listing 3.10: Pseudo code for the transformation of the HML spatial types

### 3.4.2 Results

Most elements of the defined domain-specific language HML and the ISO/TC-211 modelling approach are quite similar. Simple 1:1 mappings or filtered 1:1 mappings can be defined for most elements, e.g. packages, classes, complex types, attributes and relationships.

The mappings of the basic types cannot be defined as direct one-to-one mappings. HML defines the properties for used types on the level of the language definition whereas ISO/TC-211 focuses on the re-usage of standard import packages and specific rules for imported types. This causes partly a loss of information, esp. for the spatial types or the numeric types, where precision and value domains cannot be reflected within the UML models. The mapping rules for types turn out to be quite cumbersome, because simple assignments of language elements have to be completed with evaluation of property values.
3.5. **VERTICAL TRANSFORMATIONS**

To generalise the results, it can be stated that mapping definitions between different modelling languages are simple for similar concepts. This fits to the statements of OMG about ‘type’ and ‘instance mappings’ (see 2.4). But for some mappings, the concepts of source and target language differ strongly so that complex mappings and information losses cannot be avoided.

### 3.5 Vertical Transformation between Conceptual and Logical or Physical Models

The third working thesis to be checked, is that conceptual data models with geospatial content could also be transformed to logical or physical models utilising the techniques of model-driven software development. Although this assumption is quite likely to be fulfilled, because the main target of model-driven software development are vertical transformations and the production of code or format specifications, it should be controlled for a typical target of the geospatial domain.

#### 3.5.1 Transforming from Domain-Specific Language HML to GML Schema

Main targets for model transformations in the geospatial domain are currently format descriptions for data exchange; the most prominent is GML Schema. GML Schemas are XML Schemas that reference predefined XML Schemas for geospatial types, collections, time and other needed structures. A GML Schema derived from a conceptual model must be able to reflect as precise as possible all entities, relations, types and domains.

Again a pseudo-code illustrates the mapping between the input elements of the domain-specific language and the GML Schema elements.

**HML Model**  Similar to the mapping to the UML model, the top-level model element is not mapped to a target element (listing 3.11).

```plaintext
// No Transformation for HML Model
HML_Model -> null
```

Listing 3.11: Pseudocode for the transformation of HML Model

**HML Packages**  For the mapping of HML package structures to XML Schema only two dissatisfying solutions with information losses can be found.

Either the nested structure of packages is kept by using `xsd:complexType` or each package is used with its name to generate one own XML Schema. In the first solution all packed elements are nested in complex types, resulting
in deep, complex structures with the information loss of the package names and the namespaces.

For the actual transformation, the second solution was selected, because the naming of packages is regarded to be of major relevance and in practise, deep package nesting is rarely used. Each package is mapped to an own file and GML Schema; dependencies between packages are resolved with `xsd:import` statements.

Listing 3.12: Pseudocode for the transformation of HML Packages to GML Schemas

In listing 3.12 the transformation rules from HML ‘Package’ to `xsd:schema` are illustrated. With `[namespacePrefix]` a property must be set, which is not available in the source model. Thus the mapping definition, the source must be ‘enriched’ with external information. Usually the `[namespacePrefix]` is a URL like 'http://www.ethz.ch/ika/hml'.

**HML String Type** Contrary to UML, the ‘HML String Type’ can be reflected very well in GML Schema. The String type is mapped to `xsd:SimpleType` restricting the `xsd:string` with the length (listing 3.13).

Listing 3.13: Pseudocode for the transformation of HML String to XML Simple Type

**HML Numeric Type** The numeric types should be either mapped to the predefined types `xsd:integer` and `xsd:double` or to a type restricting a `gml:MeasureType`. It is notable that in GML Schema, no units can be provided for `gml:MeasureType` on the model level, this can only be done for the data. This mapping implies a loss of information from the conceptual model to the XML Schema definition (listing 3.14).

Listing 3.14: Pseudocode for the transformation of HML Numeric Type
3.5. VERTICAL TRANSFORMATIONS

Listing 3.14: Fragment of pseudocode for the transformation of HML Numeric Types to XML Simple Types or GML Measure Types

**HML Enumeration** The ‘HML Enumeration’ is a collection of ‘HML Enumeration Items’, which are simple String values. In XML Schema this can be represented as `xsd:simpleType` restricting `xsd:string` with the list of valid values. If ‘HML Enumeration’ has a ‘super-type’ then this one is used as restriction base instead of `xsd:string`. The ‘merge’ property of ‘HML Enumeration’ cannot be represented accordingly in XML Schema. For those cases the enumeration items are simply resolved and any information about the relations between enumerations are lost (listing 3.15).

Listing 3.15: Pseudocode for the transformation of HML Enumerations to XML Simple Types

**HML Union** The ‘HML Union Type’ is used to allow ‘either or choices’ between different types. In XML those kind of choices are packed in `xsd:ComplexTypes` with a `xsd:choice` that is pointing to the according types. Inheritance of ‘HML Union Types’ are not reflected directly in the complex types, so that the choices of the super-types are added to the type (listing 3.16).

Listing 3.16: Pseudocode for the transformation of HML Unions to XML Complex Types
CHAPTER 3. CONCEPTS

Listing 3.16: Pseudocode for the transformation of HML Union to XML Complex Types

-HML Spatial Types- In the HML modelling approach, several properties for spatial types, like reference system or value domains can be specified on model level. In GML, those properties are defined on the dataset level. A specific mapping is therefore not possible and it is sufficient to use the GML Types gml:PointPropertyType, LinePropertyType or SurfacePropertyType for attributes instead. Therefore, the mapping is handled in the transformation of 'Classes' and 'ComplexTypes' (see listing 3.18).

-HML Class- HML Classes are mapped in listing 3.17 to an xsd:complexType and the according xsd:element of that type. Following the rules for GML Schema, those complex types should inherit directly or indirectly from the root element gml:AbstractFeatureType and the elements from gml:-abstractFeature.
3.5. VERTICAL TRANSFORMATIONS

Listing 3.17: Pseudo code for the transformation of HML Class to XML Complex Type

```
<xsd:element abstract="false" name="[HML.Class.name]"
    substitutionGroup="gml:AbstractFeature"
    type="[HML.Package.name]:[HML.Class]+'Type'/">
```

The attributes are nested in a `xsd:sequence` and each HML attribute is resolved as `xsd:element`. The setting of attribute name and the cardinalities are taken from the properties of 'HML Attribute'.

The attribute type is either the according reference to the created type or to the substituting standard types of GML Schema like `gml:PointPropertyType`, `gml:LinePropertyType`, `SurfacePropertyType` or `xsd:integer` and `xsd:double` (listing 3.18).

```
HML_Attribute.type =>
    [IF HML_Attribute.type INSTANCE OF HML_Point THEN]
       gml:PointPropertyType
    [ELSE HML_Attribute.type INSTANCE OF HML_Line THEN]
       gml:LinePropertyType
    ...
    [ELSE]
       [HML_Attribute.type.package.name]:[HML_Attribute.type.name]`
```

Listing 3.18: Pseudo code to resolve the type of HML Attributes

HML ComplexType The ‘HML Complex Type’ is mapped similar to the ‘HML Class’, but the defined type does not inherit from `gml:AbstractFeatureType` and the defined element should substitute `gml:AbstractObject`.

HML Relations The Humboldt Modelling Language limits its ‘HML Association’, ‘HML Aggregation’ and ‘HML Composition’ to binary relations without attributes. In GML several alternatives to deal with relations can be defined. For 1-to-1 or 1-to-many relationships, references can be nested in the feature types. Many-to-Many relationships must always be resolved as separate ComplexType pointing to the according feature types.

In this work a similar mapping to INSPIRE is used [INSPIRE Drafting Team "Data Specifications", 2007]⁹, because it is a widely used approach. All relationships are mapped to separate complex types referencing source

⁹In the author’s opinion, the used structure to define relations within GML Schema is an unlucky choice in several points. The target element is nested as `<gml:targetElement>` in the `<xsd:annotation><xsd:appinfo>`, but `<xsd:annotation>` is initially intended to add comments and documentation to the XML code. This clearly means those sections should not contain key structural content, because software (esp. parsers) are not obliged to process those elements. The nested `xsd:appInfo` allows to define application-specific code (helping constructs for the software like parser code statements or parser hints) and is even less the place for real content. Finally the referenced element is placed as simple string within the tag and therefore cannot be validated by processing software.
and target feature types. The structure to define those references is quite complex and deeply nested as illustrated in listing 3.19. The binary HML relation endings are mapped to a sequence of two elements, with the `element` tag carrying the information on rolename and cardinality. The link to the associated class is packed in an annotation element.

```
Listing 3.19: Pseudocode to resolve HML Relations
```

3.5.2 Results

The domain-specific modelling language HML and the general purpose format specification XML differ quite strongly in concepts and structure. XML, therefore GML as well, is designed in strong hierarchies, following a tree concept and with a focus on the textual representation. This is due to the fact that XML is optimised for machine reading and not for human comprehension. All entities from the conceptual input must be mapped to structures of the format entities and complete models end up in complex definitions in XML Schema, having most of the information hidden in deep structures.

For the mappings from HUMBOLDT Modelling language to GML Schema, this results in a more template-oriented approach; mostly the target structures are fixed and information from the source model is only filled in as completing elements. Resulting from the fact that GML Schema is a representation of the physical structure, the relationships and references are resolved as textual representation; i.e., a reference to any element of the conceptual model will result in a ‘qualified name’ as textual string. In the definition of the pseudo-code, this results in long and redundant mappings.

3.6 Summary

In this chapter, the main working thesis ‘Meta Modelling Techniques will Improve Usability and Interoperability for Conceptual Geospatial Modelling’
3.6. SUMMARY

is introduced. This statement is validated along three sub-statements:

- ‘Standard meta modelling approaches can be used to create well defined-modelling languages in the field of geospatial modelling’

- ‘Horizontal transformations between models described in different modelling languages can be performed with standard transformation tools’

- ‘Vertical model transformations, esp. transformations from conceptual to logical or physical models can be facilitated by the use of well-defined modelling languages’

For all of the statements, a conceptual proof is made in this chapter and in chapter 4 according software realisations are carried out.

The first statement lead to the main question, if conceptual geospatial languages can be designed with a standard meta model approach. To test this, the abstract syntax of the INTERLIS 2 conceptual language is analysed and step wise described with the Ecore meta model language. Although Ecore was designed for model-driven software development, all aspects of the geospatial language could be reflected well in the Ecore meta model.

Additionally, the abstract syntax of a new geospatial modelling language (HUMBOLDT Modelling Language HML) is designed from a set of requirements. This language is not created as replacement of existing geospatial languages, but as a higher-specialised ‘domain-specific’ modelling language for a clearly defined task. It utilises ideas of common approaches like INTERLIS 2 and UML, but reduces and simplifies wherever possible. The resulting language is small and compact, but completely sufficient for the requested modelling environment. This new language is also precisely defined via its Ecore meta model.

The second statement about horizontal transformations between languages is especially of relevance for the concept of ‘domain-specific languages’, because this allows to use small languages without loosing compliance to established approaches. For that reason, transformation rules from the Humboldt Modelling Language to an ISO/TC-211 compliant UML approach are investigated and defined in a pseudo code, similar to the Query View Translate (QVT) standard of OMG. The created mappings utilise the language meta models and are simple and compact, especially if language elements can be mapped directly; e.g. HML:Class to UML:Class with «FeatureType» or HML:Attribute to UML:Property.

Finally the potential for vertical transformation, from conceptual models to logical models, is examined with the example of an HML model to GML Schema transformation. GML Schema is currently the most prominent exchange format for spatial data and is characterised by complex XML Schema structures and a huge set of predefined types and elements. Again, a pseudo-code for the transformation is created, but the mapping is more
complex compared to the horizontal transformation rules. This is mainly due to the little conceptual overlap between both approaches and the complex target structures that must be created for simple source elements.
Chapter 4

Implementation

The realisation of the concepts described in chapter 3 with concrete standard technologies of model-driven software development is an important proof of feasibility in this work. It should be demonstrated that the language design and the model transformations could result with reasonable efforts in suitable processes and software. In parallel to the described concepts of the previous chapter and the three main assumptions about usability of meta modelling approaches for geospatial modelling (see chapter 3.2), also three implementation steps were be carried out and are described in this chapter.

- ‘Standard meta modelling approaches can be used to create well-defined modelling languages in the field of geospatial modelling.’

In chapter 3.3 the abstract syntax of INTERLIS 2 and of an application specific modelling language HML were created in a standard meta-modelling language (eCore). Those formalised language concepts can be used to derive several software components in a highly automated way. To prove this potential of deriving well-adjusted tools, some software products, especially a graphical modelling tool for HML, should be generated in an efficient model-driven approach (chapter 4.2).

- ‘Horizontal transformations between models described in different modelling languages can be defined with standard transformation tools, if the languages are based on the same meta-modelling approach.’

For the success of domain specific languages in geospatial modelling, the ability for model exchanges between different approaches will be crucial, especially the transfer to the ISO/TC-211 UML. This would allow to use conceptual modelling tools for domain-specific languages and still be compliant to more general approaches.

In chapter 3.4 according mapping rules are described in a simple pseudo-code. A translation of these mapping rules to concrete and executable mappings is done in chapter 4.3. Special attention is drawn on situ-
ations, when rules are very simple or when meta-modelling does not help in the definition of mappings.

- 'Vertical model transformations from conceptual models to logical or physical models can be carried out by utilising techniques of the model-driven software development'

Logical models or physical structures, esp. the generation of GML Schemas or textual documents, are frequently requested outcomes from modelling tasks. Those transformations can be carried out e.g. as a two-step process, by performing first a horizontal transformation to a standardised modelling approach (ISO/TC-211) and after that a vertical transformation to the needed logical model. One-step transformations as alternative can be more simple and direct with the benefits of reducing errors and information losses, especially as there is still a lack of suitable vertical transformation software (see 2.3.1).

A realisation of the one-step vertical transformation from HML to GML Schema (see chapter 3.5) is described in chapter 4.4.

All software fragments\(^1\) of this work are designed as proof of concept in a sense of complete implementation of functionality. But those fragments are not ready to be used as out-of-the-box software with guaranteed stability, error freeness or advanced usability. Furthermore no quantitative analysis on spent efforts, development costs or efficiency was made.

### 4.1 Used Technology

The technologies in the field of model-driven software development are evolving rapidly. The choice of the selected products and frameworks reflects therefore only a snapshot of actual open source technologies at the time of this work.

#### 4.1.1 Java

Java was initially developed by Sun Microsystems in the late 1990’s for embedded systems, but evolved quickly to a popular object-oriented software language. The language is designed to be independent of the underlying hardware architecture and operation systems. The source code is precompiled into Java ‘byte code’ and then interpreted\(^2\) by the ‘Java Virtual Machine’, which is available for several platforms. The execution within the

---

\(^1\)Software fragments are pieces of a software system, which are guessed as completed in itself. Those fragments might be software libraries, but also services or even configuration files.

\(^2\)Actual Java Virtual Machines mostly run in a mixed mode ('Hot Spot Optimisation'), where time critical or repetitive operations are compiled before execution, while the rest is interpreted.
Java Virtual Machine guarantees high stability and excellent security control. In 2007 Sun opens the version 1.6 as open source project, which is now maintained by Oracle.

The used Java version is 1.6.

4.1.2 Eclipse

Eclipse was initially developed by IBM as integrated development environment (IDE) under the name VisualAge for Java. The core components were given in the early 2000’s to the open source Eclipse Foundation. Especially in Europe, Eclipse is the most popular Java IDE and several open source initiatives as well as commercial companies develop and maintain components of Eclipse. Eclipse is written mostly in Java and built on a specific component model, which allows recombining, reusing and extending those elements. This component model and several core elements are also used to create completely different applications such as mail clients (Lotus domino), desktop GIS (uDig) or reporting tools (BIHF report designer).

The used Eclipse version is ‘Ganymede’ 3.4.

4.1.3 Eclipse Modelling Framework (EMF)

The Eclipse Modelling Framework (EMF) is an open source framework to facilitate model-driven software development. It is based on the component architecture of Eclipse and well integrated in the developers’ community. There are several subprojects under the umbrella of EMF, like model comparison, model storage, model querying or different transformers. The core components offer an architecture and a complete code generation process for software systems in four steps:

1. **Definition of an abstract modelling language** A language designer creates the abstract syntax for the intended language and tools. The size and complexity is adjusted to the planned tasks and users. For the definition mostly the EMF standard eCore (see chapter 2.4.2) is used. Other possibilities are XML-Schemas or annotated Java Classes.

2. **Generation of software tools** The defined meta model is directly used to generate tools and software fragments.

   - **Internal Model Representation** as structure of Java Interfaces, Classes and Factories for their creation. This model representation is the basis for all created products; it offers a runtime structure for models following the defined meta model.
   - **XMI Schemas** and software for reading and writing. From the meta model a standard XMI file format and a library to store and read those files can be derived.
• The EMF Tree Editor. This application is a minimal tool for the creation of models according to the defined meta model. It only exposes the language structure, as defined in the meta model, in an editable tree structure. Of course this kind of editor does not support intuitive modelling and usually only acts as a minimum placeholder during software development. Mostly, the editor is replaced by a graphical editor built with GMF (see chapter 4.1.4), but also textual editors might be created with tools like Xtext\(^3\).

3. Definition of Models Modelling experts can use the languages and the editing tools to define their models. In software development, such models may represent simple configurations of applications, more complex information models for storage systems or complete workflows for business applications.

4. Code Generation From the created models, different software fragments or documentations can be derived. The base technology of EMF is the template transformation with JET (Java Emitter Template). This technique is a rather simple template tool and can be used for the creation of arbitrary texts from the models. In practise this simple technique is also mostly replaced with more sophisticated frameworks like OpenArchitectureWare (see chapter 4.1.5) or Atlas transformation system\(^4\).

![Figure 4.1: Simplified roles and products in an EMF software generation](image)

Figure 4.1 gives an overview of the steps in the EMF software production chain. A 'language designer' defines (at least) the abstract syntax of

\(^3\)http://www.eclipse.org/Xtext
\(^4\)http://www.eclipse.org/atl
the modelling language with support of ‘modelling experts’ and ‘software engineers’. The preferred meta modelling approach is the usage of eCore.

The ‘modelling experts’ define an arbitrary number of domain models in that domain-specific modelling language5.

The ‘software engineers’ utilise the available language meta model to define transformation processes for software (fragments) or documentation. Those transformations can then be applied to an arbitrary number of domain models.

This style of software production with strict role definitions and the ability to create multiple, similar fragments are often called ‘software factories’ [Greenfield et al., 2004].

4.1.4 Graphical Modelling Framework (GMF)

The Eclipse Graphical Modelling Framework (GMF) is a framework to facilitate the creation of graphical editors, which can be used instead of the simple EMF tree editor. GMF acts as a bridging element between the Eclipse Modelling Framework (EMF) and the Eclipse Graphical Editing Framework (GEF), which offers general elements for graphical tools like painting panes, defined graphical primitives or window management.

GMF is built on the techniques of EMF and utilises its components. Figure 4.2 illustrates the key components and the overall generation process to build a graphical editor for a defined language. The ‘Domain Metamodel’, taken from the EMF processes, describes the abstract syntax of the new modelling language. It is the base input for the generation process. The ‘Tooling Model’ describes menus as well as basic behaviour for the planned graphical editor. In the ‘Graphical Model’, the appearance of all model elements and the symbology is defined.

The ‘Mapping model’ combines all three input models for the generation process. This means it sets the links between elements of the domain meta model, the graphical descriptions and the menu descriptions. The combined models are used to build software libraries and configuration files that are used within the GMF and EMF runtime. The product is a complete graphical model editor that can either be used as standalone tool or as ‘plug-in’ for other Eclipse based software.

4.1.5 OpenArchitectureWare

OpenArchitectureWare is one of the most popular open-source projects dealing with model transformations6. It can be used instead of the simple JET

5There are also predefined general purpose languages available like UML or XML Schema languages.

6Since the latest release, OpenArchitectureWare is integrated together with the ‘ATLAS’ transformation system in the Eclipse Modelling Framework and maintained by the
Figure 4.2: GMF components for the creation of a graphical domain-specific editor (white elements are project-specific components; grey components are standard framework components from EMF and GMF)
4.1. USED TECHNOLOGY

transformations and offers highly sophisticated developer support. Instead of plain template approaches, the transformations can be split up into a series of model-to-model and model-to-text transformations, controlled by a workflow engine. Tools with validation, highlighting and code completion support the creation of the transformations.

The most relevant framework components are:

**oaow check**  OpenArchitectureWare check is a simple model validation component. It allows to perform quality checks at any stage of the process, e.g. integrated in the graphical model editor, to give the users live warnings during model creation or before executing the transformations to guarantee transformation quality. Typical validation rules deal with naming conventions or uniqueness constraints.

The oaow check-language is a simple combination of OCL constraints with according warning levels and messages.

**oaow Xtend**  Xtend is a textual language to define model-to-model transformations. The transformations with Xtend are meta model aware, which means that transformations are defined and executed directly on the model itself and not on any model formats, like text files or XML representations. Xtend allows transformations between models or between different modelling languages (based on Ecore meta models) and it supports the combination of different models or enriching models with external information.

Xtend is a declarative language and easy to learn and understand. In some rare cases, when the language expressiveness is not sufficient, it is possible to call special transformation classes coded in Java.

**oaow XPand**  XPand is the model-to-text language of OpenArchitectureWare. It is a modular template approach to define target products that contain static text and dynamically created elements from the input model. The language is meta model aware, contains different kinds of loops, expressions and the option of template nesting. Again, external Java classes can also be called directly.

**Workflow Definition**  The execution of the single components is controlled by workflow definition files. These XML files can be composed of several substructures and be externally parameterised. A typical workflow may include following steps: (1) configuration of the transformation environment with the meta models of the used modelling languages, (2) reading and validation

---

Eclipse Foundation. This results in slight modifications of structures and terminology. In this work all implementation is done with OpenArchitectureWare Version 4.3 and all used terminology sticks to this.
of the input model, (3) model-to-model transformations, (4) model-to-text transformation and (5) storage of the target products.

4.2 Implementation of a Domain-Specific Modelling Language and a Graphical Modelling Editor

This chapter describes a concrete implementation of a domain-specific modelling language with an supporting graphical model editor. The chosen examples are developed around the abstract syntax of the Humboldt Modelling Language as defined in chapter 3.3.2.

The realisation utilises EMF for the core components, GMF for the concrete syntax and the graphical editor as well as OpenArchitectureWare 'check' for some basic consistency control.

4.2.1 Abstract Language Definition

The main input for the Eclipse Modelling Framework is the abstract language definition (or the meta model), preferably as Ecore model. In chapter 3.3.2 this meta model is stepwise developed and is directly used as input for the software generation process (in Annex B the complete meta model is available).

4.2.2 Generation of Core Code Fragments

From the abstract language definition in forms of an Ecore meta model an intermediate 'generator model' is derived. This intermediate model acts as an enriched model for the code production process. It allows manipulations like changing or renaming package structures, introducing individual factory classes or simply adding comments and copyright notes to the generated Java classes. In this work, only simple copyright notes are added.

From the generator model Java packages for internal model representation, model storage and basic editor functionality are derived in an automated standard process (see also chapter 4.1.3).

4.2.3 Definition of Graphical Notation

In contrast to the abstract language syntax, there is no standard notation for a concrete graphical syntax. In this chapter a textual description with explanatory illustrations is given for the graphical appearance of the model elements.

In general, the chosen graphical notation is close to UML, but with specific iconisation and colours to enhance readability.

- Package: A ‘Package’ may have two representations. Firstly it can act as graphical container; i.e. the representation is the background pane,
containing all packed elements. Secondly it can appear as rectangle with two nested rectangles (compartments) inside; the small, yellow compartment at the top containing the standard symbol and the local package name and the big central one can contain all packed elements (see figure 4.3).

**Figure 4.3:** Graphical representation of the model element ‘Package’.

- **Class:** Similar to UML classes the representation is a rectangle with two main compartments. The green, small top one contains the standard icon, the name and a boolean field to mark ‘abstract’ or not. The big, central compartment comprises a list of the member attributes. The attributes are listed with their cardinality and the attribute name (see figure 4.4).

**Figure 4.4:** Graphical representation of the model element ‘Class’.

- **Complex Type:** ‘ComplexTypes’ are also defined in similarity to UML classes. The graphical representation is similar to HML ‘Class’ elements, but with an orange top compartment as used for all HML ‘Types’ (figure 4.5).

**Figure 4.5:** Graphical representation of the model element ‘ComplexType’.
• Enumeration: Like all ‘Types’, the ‘Enumeration’ is designed as rectangle with an orange top compartment, containing icon and name. The main compartment is a simple list of enumeration items (see figure 4.6).

![Figure 4.6: Graphical representation of the model element ‘Enumeration’](image)

• Simple Types: The simple types ‘String’, ‘Numeric’ or ‘Union’ as well as the spatial types ‘Point’, ‘Line’ and ‘Polygon’ are orange in their top compartment (see figure 4.7) and carry their specific properties in the main compartment; like string length or numeric precision.

![Figure 4.7: Graphical representation of simple types; ‘UnionType’, ‘StringType’, ‘NumericType’ and spatial types ‘PointType’, ‘LineType’, ‘PolygonType’ and ‘SurfaceType’.](image)

• Unit: The specific element for numerical units is depicted as rectangle with a cyan top compartment. This compartment contains the icon, the name and abbreviation (see figure 4.8). The property for additional definitions is placed in the main compartment.

![Figure 4.8: Graphical representation of the model element ‘Unit’.](image)

• Coordinate System: The definition of a coordinate system is illustrated by a rectangle figure with a dark green top compartment and a special icon. Additionally to the name, the definition (mostly EPSG code) is visible (see figure 4.9). The central compartment contains the ordered list of the coordinate ‘Axes’.
4.2. DOMAINSPECIFIC LANGUAGE & GRAPHICAL EDITOR

Figure 4.9: Graphical representation of the model element 'Coordinate System'.

- Coordinate: The graphical representation of a Coordinate Element is a rectangle with a grey top compartment, in which the icon and coordinate name are placed (see figure 4.10). The central compartment lists the used numeric types.

Figure 4.10: Graphical representation of the model element 'Coordinate'.

- Inheritance: The representation of inheritance as simple triangular arrow is taken from the UML definition (see figure 4.11).

- Enumeration Merge: The merging inheritance for enumerations is depicted as dotted arrow line with triangular arrow head.

- Union Connector: The union relation of the Union Types is a simple dotted line.

- Association: An association between classes is illustrated as simple line. The association ends are labelled with according role names and cardinalities (see figure 4.11).

- Aggregation: The appearance of aggregations are defined like associations, with one aggregation end marked as unfilled diamond (see figure 4.11).

- Composition: The graphical representation of a composition is defined like an aggregation, but with a filled diamond end.

Those graphical definitions must be translated to a GMF graphical model (see chapter 4.1.4). The graphical modelling structure itself is defined in Ecore, so that the simple tree editor of EMF can be used to define the representation (see figure 4.12). On the CD in Annex ?? the complete graphical model can be found.
CHAPTER 4. IMPLEMENTATION

Figure 4.11: Graphical representation of the model elements ‘Inheritance’, ‘Association’ and ‘Aggregation’.

Figure 4.12: Screen shot of the Eclipse Modelling Environment with the definition of the HML concrete syntax.
4.2.4 Model Validation

For additional validation of models the OpenArchitectureWare ‘Check’ component is selected (see chapter 4.1.5). The definition of constraints is straightforward as a combination of simple OCL phrases combined with according alert actions. In listing 4.1 two constraints are declared; the uniqueness of element names within a package and minimum spelling rules for object names. This could be easily extended by adding further rules.

```plaintext
context hml::Attribute WARNING
'Attribute '+this.name+' should start with lower Case':
this.name.toFirstLower() == this.name;

context hml::Class ERROR
'Class '+this.name+' names must be unique in the package':
((hml::Package) this.eContainer).classes.
    select(e|e.name == this.name).size == 1;
```

Listing 4.1: Two constraints defined with oaw ‘Check’ language

4.2.5 Generation of a Graphical Model Editor

With the elements of the abstract syntax and the concrete syntax, a complete domain-specific modelling language is defined. But for the creation of the requested graphical model editor, additional information is needed as shown in figure 4.2.

The GMF ‘Tooling Model’ is a very simple information model containing a lists of entries and icons. This model is used to generate standard and context sensitive menus. This and all other sources are provided on the CD in Annex ??.

For a model editor, the abstract syntax, the concrete syntax and the tooling are linked in the ‘Mapping Model’. This model contains e.g. the links between the ‘Class’ element of the abstract syntax, the according menu entry in the ‘Tooling Model’ and the appearance in the ‘Graphical Model’, including all mappings between their properties and diagram labels.

Those four linked models are combined in an automated transformation process to a complex ‘Graphical Editor Generator Model’. The functionality of this intermediate model is similar to the EMF ‘Generator Model’, but adjusted to the GMF build process. With this model, direct influence can be taken on the generation process and the package structures. Additional documentation or copyright notes can be added here.

In this work the ‘Graphical Editor Generator Model’ is used to add simple copyright notes and model validation as described in chapter 4.2.4. For this modification, the implementation packages of OpenArchitectureWare are published together with the validation files to the generator framework. The validation behaviour (‘on-the-fly’ or ‘on-demand’ validation) is then declared in the intermediate model.
The ‘Graphical Editor Generator Model’ is finally transformed to a complete Eclipse plug-in, the HUMBOLDT GeoModel Editor’. This editor can either be executed standalone or be integrated into existing Eclipse runtimes. Figure 4.13 illustrates the HUMBOLDT GeoModel Editor as standalone application with integrated additional plug-ins for the help system and a minimal file tree. The plug-in technology also allows the integration of other editor tools (e.g. XML Editors, GML Editors), connectivity to model repositories (e.g. simple subversion support) or transformation engines.

![Figure 4.13: Screen shot of the ‘HUMBOLDT GeoModel Editor’ as standalone application](image)

A complete and runnable version of the Graphical Model Editor can be found on the CD in Annex ??.

### 4.2.6 Description of the Graphical Model Editor

The Graphical Model Editor as complete software product is composed of basic Eclipse components dealing with general management of windows, files and menus plus some plugged-in components like the help system, the file tree and finally the generated plug-in for modelling. The modelling plug-in consists of the tab-able main editor panel, an palette of available modelling elements and a property panel for the detailed settings of all elements.
4.2. DOMAIN SPECIFIC LANGUAGE & GRAPHICAL EDITOR

Usage of the Graphical Editor

The main functionalities of the editor are described along typical tasks of geospatial conceptual modelling. The single tasks focus on the user interaction and do not describe a continuous and complete model (examples for complete models can also be found on the CD in Annex ??).

Create new model: To create a new data model, the ‘new-file’ wizard is called via the menu entries. In the creation process, two file locations are requested from the user; one for the model file as XMI representation and one file to store the state of the graphical editor. An editor panel opens with an empty rootpackage, which can be renamed in the properties panel.

Add model elements: New model elements can either be added by selecting them in the palette or by using a context menu in the graphical pane as shown in figure 4.14. Both options use the underlying meta model, so only valid elements can be selected in the given context; this means, it is possible to add classes or data types to packages, but no attributes or enumeration items. For every element the properties view can be used to define all details.

![Figure 4.14: Screen shot of a ‘Context Menu’ in the HUMBOLDT Geo-Model Editor](image)

Add associations: The creation of different associations is quite similar; either by using the palette or the context menu for any connections. Again, the associations or inheritances can only be drawn for valid structures. This means it is prevented by the editor to define mixed inheritances or relations between data types and classes.

Import: The modelling environment supports two types of imports; ‘loading of resources’ and ‘creation of shortcuts’. ‘Resources’ are used to grant access to complete other model files. The loading of resources is reasonable for the utilisation of standard libraries as described in chapter 3.3.2.

‘Shortcuts’ offer the possibility to import single elements into models. Those elements are displayed with a marker in the graphical model (see figure 4.15). Shortcuts should be used to build associations or inheritances between different models.
CHAPTER 4. IMPLEMENTATION

Model navigation: To structure models the use of packages is foreseen in the design of the modelling language. To support this structure in the editor, separate and synchronised editor panels for each package can be started by double clicking on the package.

For navigation, the ‘Outline Panel’ can help; this can be switched between a graphical overview of the editor pane or the tree view of all model elements.

Model validation: The model validation is available as entry in the ‘Diagram’ menu. Warnings and errors are directly marked within the diagram as yellow alert sign or red error markers. A mouse-over tool-tip shows the according error message (see figure 4.16).

Model storing and loading: As already noted in the creation of new models, the model editor stores two files, one containing the model and a second to store the graphical state. To restore a model, it is sufficient to open the file with the graphical state, as it contains a reference to the model file. But it is also possible to simply load a model file with a default graphical representation by using the ‘initialise hmldiagram_file’ option.

4.2.7 Results

The presented approach to create a domain-specific modelling language for geospatial data models together with suitable editing tools is useful. The language and the according software tools can be derived in a completely model-driven way from a standardised meta model approach. The generated software tools are rich in functionality, stable and easy-to-use.

In the production process, the definition of the abstract syntax is the key element. It is rather straight forward and well supported by tools. The generation of the graphical editor is more complicated; the set-up of the
4.3. LANGUAGE-TO-LANGUAGE TRANSFORMATIONS

software environment and the combination of the tools is still rather complex and not feasible without sound knowledge of the underlying frameworks and software processes. Also the configuration of the graphical appearance and the validation framework is a bit complex, despite to supporting tools.

In general, the learning curve of this approach is steep and most effort is spent in the set-up of the production process. But after this effort, the generation and adoption of modelling languages with according tools is a fast and agile procedure. The integration into the stable Eclipse components offer reliable functionality and easy integration with other tools such as version control, external editors or deployment mechanism.

4.3 Realisation of Horizontal Language-to-Language Transformations

A feasible transformation between different modelling languages would be a real benefit compared to current practises. This would open compatibility between several approaches. But those transformations are usually difficult to define and execute. In chapter 3.4 it is assumed that transformations between languages based on the same meta model approach could be carried out more easily, because of the existence of meta-model-aware transformation tools from model driven software development.

To test this assumption, a meta-model-aware transformation is set up to generate an ISO/TC-211 compliant UML models from models created with the HUMBOLDT GeoModel Editor in the Humboldt Modelling Language of chapter 4.2. The transformation rules are already defined in the pseudocode of chapter 3.4. This code must be transposed to an executable code in a fitting environment.

4.3.1 Environment and Runtime

As environment for defining and executing the transformation, Eclipse with OpenArchitectureware is selected\textsuperscript{7}. As input for the transformation process arbitrary conceptual models are defined within the graphical editor and stored as local files. After the transformation, the generated UML model can be opened and processed with standard UML tools\textsuperscript{8} (see figure 4.17).

4.3.2 Definition of the Transformation

The complete transformation procedure is split in several single process steps. Those steps with correct parameters are defined in the workflow definition\textsuperscript{7} In a productive environment, the runtime could be e.g. integrated in the HUMBOLDT GeoModel Editor.

\textsuperscript{8}In the test set-up, the UML editors Enterprise Architect (http://www.sparxsystem\textregistered.com) and the open source editor Papyrus (www.papyrusuml.org) are used.
Figure 4.17: Set-up for the transformation process. The Graphical Model Editor is used to define a conceptual model. The OpenArchitectureWare installation acts as mapping environment and transformation engine. The resulting UML model can be visualised with standard tools.

For the given task, the workflow consists of four major steps: (1) loading of the input model, (2) validation of sources, (3) transformation and (4) storage of the target model.

Loading of Sources

The processing of all data in the OpenArchitectureWare framework is done in so-called ‘slots’ or processing pipes. The reading components just open files for stream-reading into slots, which can then be used by other components. Beside model files, input readers can also be defined for in-memory models or database access.

Listing 4.2: Opening a model file of an Ecore based language (snippet of the oaw workflow file). The generic configuration structure consists of the reader class ‘oaw.emf.XmiReader’ with properties for filename and slot-name.
4.3. LANGUAGE-TO-LANGUAGE TRANSFORMATIONS

Beside the input model, an UML template file is loaded in listing 4.2. This template is an empty UML class model with imports of standard ISO/TC-211 packages. The template is loaded into the slot called 'umlmodel' and will be enriched in the following transformation processes.

Model Validation

Before performing any transformation steps, the input model should be validated to ensure structural and logical correctness. This is configured in the workflow definition file by calling a validation component 'oaw.check.CheckComponent' (Listing 4.3).

Firstly the model in the input slot 'hmlmodel' is validated against the meta model 'geomodel.ecore' and secondly it is tested against the oaw check 'file hml2umlCheck' that is defined similarly to chapter 4.2.4.

Listing 4.3: Snippet of oaw workflow file to validate the HML input model against the metamodel and an additional check file

```xml
<component class="oaw.check.CheckComponent">
  <metaModel id="geomM" class="org.openarchitectureware.type.emf.EmfMetaModel">
    <metaModelFile value="geomodel.ecore"/>
  </metaModel>
  <metaModel id="ecoreMM" class="oaw.type.emf.EmfMetaModel">
    <metaModelPackage="org.eclipse.emf.ecore.EcorePackage"/>
    <checkFile value="check::hml2umlCheck"/>
    <emfAllChildrenSlot value="hmlmodel"/>
  </component>
```

Model Transformation

The transformation itself is defined as a set OpenArchitectureWare Xtend functions that are called and parameterised from the workflow. The Xtend functions can be stored in one or more Xtend files, which makes it easy to structure and maintain the sources.

As described in chapter 4.1.5, oaw Xtend is a specialised meta-model-aware, functional language designed for the transformation of models. Practically a jump-in function is called with parameters from the workflow execution; within this function model elements are processed, manipulated, removed or created and other subfunctions are called. This can be nested arbitrarily. Xtend especially allows the processing of complete collections and offers limited polymorphism of functions, so that the resulting code is compact, functionally rich and mostly avoids procedural loops or case statements.

Xtend transformations always strictly follow the abstract syntax of the model languages. This means, at every step of the transformation process,
always complete and valid models are handled. This ensures a high quality of the transformation, but can complicate the mappings, because all structural rules of the model creation must be followed; e.g. firstly a UML ‘model’ element must be created, before ‘packages’ can be added. The same is valid for ‘classes’, ‘associations’, ‘attributes’ and all other UML model elements. In a consequence, this is especially difficult for the ISO/TC-211 stereotypes and tagged values, that can only be applied after packages, classes and associations are completely generated.

**Workflow Configuration** In the first transformation step, the UML model template is loaded and the base structures of packages and classes are created. The execution component ‘oaw.xtend.XtendComponent’ calls the ‘createPackage’ function of the Xtend mapping file with the source model in the input slot ‘hmlmodel’ and the target template in the output slot ‘umlmodel’ (Listing 4.4).

---

```xml
<component class="oaw.xtend.XtendComponent">
  <metaModel idRef="geoMM"/>
  <metaModel idRef="ecoreMM"/>
  <metaModel idRef="umlMM" class="oaw.uml2.uml2MetaModel"/>
  <invoke value="libs/packages::createPackages(hmlmodel, umlmodel)"/>
</component>

...<component class="oaw.xtend.XtendComponent">
  <metaModel idRef="geoMM"/>
  <metaModel idRef="ecoreMM"/>
  <metaModel idRef="isoMM"/>
  <invoke value="libs/hml2profile::hml2profile(hmlmodel, umlmodel)"/>
  <globalVarDef name="xsdVersion" value="${xsdVersion}"/>
  <globalVarDef name="targetNamespacePrefix" value="${targetNamespacePrefix}"/>
  <globalVarDef name="targetXsdDirectory" value="${targetXsdDirectory}"/>
</component>
---
```

Listing 4.4: Snippet of oaw workflow file to execute the step-by-step transformation from a HML model to the UML model

Further transformation calls for inheritances and associations are executed after this step, but not listed here (the complete transformation can be found on CD in Annex ??). The final step in this transformation is to apply the ISO/TC-211 profile for geospatial models. Again the ‘oaw.xtend.Xtend’ component is used to call the Xtend function ‘hml2profile’. Interesting is the introduction of the external variables ‘xsdVersion’, ‘targetNamespacePrefix’ and ‘targetXsdDirectory’, which are used to enrich the target model with
information that cannot be found in the source model. The values for the variable are stored in simple properties files.

**UML Model and Packages** The first transformation deals with the creation of the UML root ‘model’ element and ‘packages’. In chapter 3.4.1 the mapping rules are defined; the ‘rootpackage’ of the HML model will be mapped to a UML ‘model’ element. The nested packages of the source model will accordingly be mapped to UML ‘packages’. The adding of the «ApplicationSchema» as well as all other ISO/TC-211 stereotypes will be carried out in a separate final step.

```xtend
// The hml root package will be used as uml model element
uml::Model createPackages (hml::Package hModel, uml::Model uModel ) :
  uModel.setName(hModel.name) =>
  uModel.nestedPackage.addAll(hModel.subpackages.collect(
    e | createMember(e, uModel)));

// Nest the subpackages in a recursive manner
create uml::Package createMember (hml::Package hPackage, uml::Model uModel) :
  this.setName(hPackage.name) =>
  this.nestedPackage.addAll(hPackage.subpackages.collect(
    e | createMember(e, uModel)));
```

Listing 4.5: Snippet of the Xtend mapping for the creation of the UML model element and the packages

The mapping concept is realised with two Xtend mapping functions (listing 4.5). First `createPackages` is called to handle the root package of the source. The slot with the source model and the slot with the template UML are given as parameters. In `uModel.setName(hModel.name)` the name of the already existing UML ‘model’ element is set to the HML root package name.

The next statement collects all subpackages of the source root package and calls for each the second function `create uml::Package createMember`, which creates for each source ‘package’ of the source model one according UML ‘package’. The returned UML ‘packages’ are all added to the UML ‘model’.

The creation of the nested packages in the second function is quite similar. Here UML ‘packages’ are freshly created instead of manipulating the existing UML model. The nested structure of the packages is handled by the recursive behaviour of the code.

**UML Classes and Relations** Chapter 3.4.1 defines the mapping rule for all source ‘classes’, ‘complex types’, ‘enumerations’ and ‘union types’ to UML classes. For all of them the name is mapped to the UML ‘class’ name. The needed differentiation to the according stereotypes «Feature-
Type», «DataType», «Enumeration» and «Union» is performed and described in the final profiling step.

```java
uPack.ownedType.addAll(hPack.types.typeSelect(hml::ComplexType).
    collect(e|createClass(e)));

uPack.ownedType.addAll(hPack.types.typeSelect(hml::Enumeration).
    select(e|e.name.toUpperCase().trim() != "BOOLEAN").collect
    (e|createClass(e)))

uPack.ownedType.addAll(hPack.types.typeSelect(hml::UnionType).
    collect(e|createClass(e)));

uPack.ownedType.addAll(hPack.classes.collect(e|createClass(e))); // ComplexType to ClassDummy
create uml::Class createClass (hml::ComplexType hComplex):
    this.setName(hComplex.name) ;
    this.setIsAbstract(hComplex.abstract);

// EnumerationType to ClassDummy
create uml::Class createClass (hml::Enumeration hEnum):
    this.setName(hEnum.name);
...

// UnionType to ClassDummy
create uml::Class createClass (hml::UnionType hUnion):
...

// Class to ClassDummy
create uml::Class createClass (hml::Class hClass):
...
```

Listing 4.6: Snippet of the Xtend mapping for the creation of the UML classes from HML Class, HML ComplexType, HML UnionType and HML Enumeration.

In listing 4.6 the implementation of the main mapping rules towards UML ‘classes’ is listed. The control statements collect all source elements of ‘complexType’, ‘enumeration’ etc., call the create function `create uml::Class createClass` for the according UML ‘class’ and add them to the UML ‘package’. The special treatment of the HML ‘Boolean’ is solved with a `select` filter operation in the control statement.

After the basic structure is created all relations, inheritances and attributes are added. The setting of the simple properties like role names or cardinalities are straightforward and comparable to the procedures of setting UML ‘class’ or ‘package’ names. More effort is needed to create the references between UML model elements; e.g. set the ‘superclass’ in cases of inheritance. For this operation, a helper function `getCorrespondingUML` (listing 4.7) needs to be written to retrieve a reference to an existing UML ‘class’ of a given source package and name of a source type or class.
4.3. LANGUAGE-TO-LANGUAGE TRANSFORMATIONS

Types
As already obvious from the transformation rules in chapter 3.4.1, the mapping of types is the most cumbersome task in defining the transformation process. Only the complex structures of the previous chapter are mapped to new UML 'classes'. All other types are replaced by UML primitives or imported types from the ISO/TC-211 packages.

For the property 'type' of HML 'Attributes', the references must be redirected accordingly. This is simple for 'StringTypes', but more complex for numerical types, with several mapping options; to 'Integer' for types with precision zero or to 'UnlimitedNatural' or to different ISO-19103 'Measures'.

Listing 4.7: Snippet of the Xtend helper to retrieve a reference to an existing UML class for a given source element

```java
// Get uml class for hml element
uml::Class getCorrespondingUML (hml::ModelElement
  hElement, uml::Model uModel):
  uModel.allOwnedElements().typeSelect(uml::Class).select(
    e | e.name == hElement.name && e.owningPackages().name.
      containsAll(createPackageList(hElement).name)).first();
```

```java
// Match String types
private uml::Type getBaseType (uml::Model model,
  hml::StringType type,
  uml::Property prop):
  getImportedPrimitiveType(model, "String");

// Split Numeric Primitives and MeasureTypes
private uml::Type getBaseType (uml::Model model,
  hml::MeasureType type,
  uml::Property prop):
  type.unit == null
    ? getNumericPrimitiveType (model, type, prop)
    : getNumericMeasureType (model, type, prop);

// Split Integer and Natural
private uml::Type getNumericPrimitiveType (uml::Model model,
  hml::MeasureType type, uml::Property prop):
  type.precision == 0
    ? getImportedPrimitiveType (model, "Integer")
    : getImportedPrimitiveType (model, "UnlimitedNatural");

// Lookup different MeasureTypes
private uml::Type getNumericMeasureType (uml::Model model,
  hml::MeasureType type, uml::Property prop):
  switch {
    case (hasOneSuperUnit(type.unit, "Area")):
      getImportedType(model, "Area")
  }
```
case (hasOneSuperUnit(type, unit, "Angle")):
    getImportedType(model, "Angle")
...
default : getImportedType(model, "Measure")
}

Listing 4.8: Snippet of the Xtend function to map attribute types to different UML types

For the mapping of attribute types the function `getBaseType` is overloaded with different signatures. In the Xtend code snippet of listing 4.8 the functions for `hml::StringType` and `hml::NumericType` are illustrated. An HML ‘string’ type is simply transposed to the imported primitive UML type ‘String’, which is retrieved by a simple lookup helper function.

The ‘numeric’ attribute type is handled stepwise, inspecting first potential ‘units’ and then the ‘precision’ of the type. For the differentiation of the ‘measure’ types, the inheritance hierarchy is examined. If an abstract super type from the standard base model (see chapter 3.3.2) can be found in the inheritance hierarchy, then a transformation to a more specific ‘measure’ type can be made.

The transformation of spatial types follows the same mechanism (listing 4.9). Again `getBaseType` functions are used with according signatures. To differentiate between simple geometry types (‘GM_Point’, ‘GM_Line’, etc.) and multiple geometries (‘GM_MultiPoint’, ‘GM_MultiLine’, ...), the ‘cardinality’ of the attribute in the source model is analysed.

```xtend
case (type, type) :
    prop.upper > 1 ?
        (getImportedType(model, "GM_MultiPoint") ->
        prop.setUpper(1))
    : getImportedType(model, "GM_Point")
```

Listing 4.9: Snippet of the Xtend function to map attributes with geometry types

### Applying the UML Profile

The last transformation step is the application of the ISO-19103 profile on the created UML elements.

```xtend
package expandApplicationSchema (hml::Package hModel,
    hml::Package app$) :
    app$.applyStereotype("Iso19103Profile::ApplicationSchema") ->
    app$.setTaggedValue("Iso19103Profile::ApplicationSchema", "xsdDocument",
    targetXsdDirectory() + app$.name) ->
    app$.setTaggedValue("Iso19103Profile::ApplicationSchema", "targetNamespace", targetNamespacePrefix() + app$.name) ->
```

...
4.3. LANGUAGE-TO-LANGUAGE TRANSFORMATIONS

Listing 4.10: Snippet of the Xtend code for applying stereotypes and tagged values from the ISO-19113 profile on packages and enumerations

In the code snippet of listing 4.10 the stereotype «ApplicationSchema» is applied on the top level packages and a function for the «Enumeration» stereotype is shown.

Obvious in the code fragment is the weak support for UML profiles: all stereotypes and tagged values must be set via generic functions as simple string values, which are of course not checked by validators and not strongly supported by editors.

Storage of Target Model

The last step in the processing pipe is the storage of the generated UML model. The output slot with the completed UML model is stored with the help of the OpenArchitectureWare ‘org.openarchitectureware.uml2.UML2Writer’ to a file. The format is the standard XMI representation of UML that can be read by most UML tools (listing 4.11).

Listing 4.11: Snippet of the oaw workflow to store the generated UML model in a file

4.3.3 Execution of Transformation

The transformation is executed by calling the OpenArchitectureWare workflow runner from the Eclipse platform. During transformation a log is produced and can be monitored in an execution console. The complete generation process is executed within a few seconds, depending on the size of the model and the amount of log statements. The produced XMI file representing the complete UML model can now be opened by an arbitrary UML editor.

4.3.4 Results

In this chapter the realisation on a horizontal model-to-model transformation from the domain-specific HML approach to the standardised ISO/TC-211 approach for UML was shown. With techniques from model-driven software development the mappings and work-flows were defined and executed. The
The functionality of the used software framework, OpenArchitectureWare was completely sufficient and mappings on base of Xtend could be defined quite clear and structured.

The stepwise transformation controlled by a workflow definition is very useful, it includes the possibilities of validation for input models and of stepwise processing with reduced complexity. The tool support for defining and executing the mappings is very good. There are meta model aware editors with code completion and syntax highlighting for model elements, their properties and for all control statements of the mapping language Xtend.

But the definition of the transformations itself is not trivial:

- The transformation language follows the ideas of set processing; this means, the programming paradigm differs from object orientation or procedural programming. On one hand this leads to compact and functionally rich mapping code that is very similar to the pseudocode developed in chapter 3.4. But on the other hand, most developers feel more comfortable with procedural or object-oriented approaches.

- The construction of the outcome model always works on the complete and correct model. This helps in the production of good and high quality models, but is quite often complex whenever associations between objects are affected.

- The UML meta model is very complex and sometimes not intuitive. For the definition of the mappings sound knowledge of UML is needed. Even the excellent tool support cannot always compensate the complexity of that meta model.

- The usage of stereotypes adds an additional transformation step to the complete process. This complicates the transformations, especially as UML profiles are only weakly supported by the editor tools of Open-ArchitectureWare.

### 4.4 Realisation of Vertical Transformations

To check the potential of the used frameworks also for vertical transformations, two important examples will be implemented.

One transformation follows the mapping rules from chapter 3.5 for a transformation from the application specific modelling language HML to GML Schema. The other transformation is a very simple approach to derive a HTML documentation of the model.
4.4. REALISATION OF VERTICAL TRANSFORMATIONS

4.4.1 Transforming between Application Specific Language HML and GML Schema

For the transformation of input models to GML Schema, the mapping rules from chapter 3.5 should be transposed to real code fragments of the OpenArchitectureWare framework. The transformation to that target format uses a special capability of the oaw framework: although XML Schema (and therefore GML Schema as well) defines a physical file structure, a metamodel of XML Schema and its serialisation is already predefined within the framework. Additionally, some helper libraries for some complex issues of XML Schemas, like resolving on qualified names or working with collections, are available. As a result, the vertical transformation from the conceptual model to a GML Schema can also be executed as a model-to-model transformation with a default serialisation.

Definition of the Workflow

The general execution of the mappings is defined as oaw workflow file similar to the horizontal mapping. First the input file is loaded and validated with oaw check files, then the transformation itself is executed and finally the resulting model is serialised to an XML Schema file.

Definition of the Mapping

The technique and the general usage of the transformation language Xtend is very similar to the horizontal mapping between two conceptual modelling languages of chapter 4.3. But the semantics of the source and target language are quite different; the application specific language is used to describe a model of reality, while XML Schema describes a dataformat. The following code samples and explanations will focus on the peculiarities of that vertical transformation aspect. The full mapping files can be found on the CD in Annex ??.

GML Schema  The single HML packages are mapped to separate GML Schema files. Each of the target schemas gets a namespace consisting of a prefix, which is defined as external property, and the HML package name. A set of standard schemas (http://www.opengis.org/gml/3.2, http://www.w3.org/1999/xlink and http://www.w3.org/2001/XMLSchema) is included first.

```java
/* create a XML Document
create XmlSchemaDocumentRoot this
   createDocumentRoot(hml : : Package pack) :
setSchema ( createSchema (pack) ) ->
this .XMLNSPrefXMap. put ("xlink", "http://www.w3.org/1999/xlink") ->
this .XMLNSPrefXMap.put
```
CHAPTER 4. IMPLEMENTATION

In listing 4.12 the function `createRootDocument` is called from the workflow for each ‘HML Package’. This function creates the root document element of the XML Schema, adds namespaces and calls the `createSchema` function. Before adding all nested `xsd:simpleTypes` and `xsd:complexType` to the schema, the needed imports are resolved and included.

**HML String Type to Simple Type**  The mapping function of the HML String Type to `xsd:simpleType` contains a nested function call to create the restriction type `xsd:string` (listing 4.13).

```
SimpleType createStringType(hml::StringType str):
    let r = new TopLevelSimpleType:
        r.setName(str.name) ->
        r.setRestriction(createRestrictionType(str)) ->
        r;

RestrictionType1 createRestrictionType(hml::StringType str):
    let r = new RestrictionType1:
        r.setBase(createQName("http://www.w3.org/2001/XMLSchema", "string")) ->
        // the length restriction should be set here
        // r.setMaxLength(str.length) ->
        r;
```

Listing 4.13: Fragment of the Xtend mapping file for HML String Types to XSD Simple Types

Although this mapping appears to be easily realised, it causes problems and cannot completely be mapped. The underlying XSD metamodel of the framework is incomplete and not documented for the restriction of simple
types. For that reason only a restricted type can be produced, but no length limitation can be added.

**HML Numeric Type to Simple Type or GML Measure Type** The mapping of HML Numeric Types in listing 4.14 includes two decisions; for Numeric Types with units, the `gml:MeasureType` is extended and for the others either `xsd:integer` or `xsd:double` are restricted. Unfortunately, for the simple types the same problem occurs as for the String Types; no concrete restriction can be created due to the framework limitations.

With that problem, the simple numeric types are not mapped to `xsd:simpleTypes` with a restriction on value domain, but the attributes use directly the `xsd:integer` or `xsd:double` instead (see the mapping of attributes).

```xml
xmlSchema::ComplexType createMeasureType (hml::NumericType num):
let type = new TopLevelComplexType:
  type.setName(num.name+"Type") ->
  type.setAbstract(num.abstract) ->
  switch {
    case (num.supertype == null):
      type.setSimpleContent(createMeasureSimpleContentDefault (num))
    default:
      type.setSimpleContent(createMeasureSimpleContentInherited (num))
    }
  ->
  type;

private xmlSchema::SimpleExtensionType
  createMeasureExtensionBaseDefault (hml::NumericType ft):
let r = new SimpleExtensionType:
  r.setBase(createQName(‘http://www.opengis.net/gml/3.2’,
    ‘MeasureType’)) ->
  ...
```

Listing 4.14: Fragment of the Xtend mapping file for HML Numeric Types to XSD Complex Types

For the complex Measure Type, the mapping restriction base is set either to `gml:MeasureType` if no inheritance is defined on the HML ‘Numeric Type’ or the according supertype is chosen.

**HML Enumeration to Simple Type** Following the rules described in chapter 3.5, the transformation can easily be defined similar to ‘HML String Types’ as a restriction of the `xsd:string` (listing 4.15).

For ‘HML Enumerations’ two different kinds of inheritances are defined:

1. ‘supertype’ limits the used ‘enumeration items’ to a subset of the supertype. For ‘supertypes’ the `extensionBase` of the `simple type` is set accordingly.
2. 'merge' is the product of all 'enumeration items'. For this kind of inheritance, all 'enumeration items' must be resolved and added to the 'enumeration'.

```java
xmlSchema::: SimpleType createEnumeration (hml:::Enumeration enum):
  ...
  // adds all enumeration items
private xmlSchema::: RestrictionType1 addItem
  (xmlSchema::: RestrictionType1 r, hml:::Enumeration enum):
  enum.merge.size != 0
  ? enum.merge.collect(e | r.addItem(e))
  : r ->
  r.enumeration.addAll(enum.item.collect(e | createEnumFacet(e)))
  ->
  r;
private xmlSchema::: NoFixedFacet createEnumFacet (hml:::EnumItem it):
  let f = new NoFixedFacet:
  f.setValue(it.text) ->
  f;
```

Listing 4.15: Fragment of the Xtend mapping file for HML Enumeration Types to XSD Simple Types. The Code covers the resolving of merged Enumeration Items and in the function `createEnumFacet' the restriction of an XSD Simple Type.

In the mapping code the collecting of items is realised in a recursive `addItem` function, which calls itself for all 'merge' entries of the 'HML Enumerations'. Those operations on collections can be formulated very concise and short\(^9\).

Contrary to the missing support for string length or minimum/maximum restriction, the `xsd:enumeration` item can be defined as restriction. However, the property used for this is called `xsd:NoFixedFacet` and neither intuitive nor well-documented.

**HML Union Type to Complex Type**  The implementation of the mapping for 'HML Union Types' is straightforward and comparable with 'HML Enumeration'.

**HML Class to Complex Type**  Like all other elements, the transformation function `createFeatureType` for the HML Classes is called as a nested function from `createSchema`. It creates a new Complex Type directly at the top level and sets its basic properties 'name' and 'abstract'.

The mapping function in listing 4.16 for 'HML Class' uses mainly code as already described for other elements before.

\(^9\)Unfortunately no empty collections can be handled, so that all enumerations have to be checked against the collection size 0 beforehand.
4.4. REALISATION OF VERTICAL TRANSFORMATIONS

Listing 4.16: Fragment of the Xtend mapping file for HML Classes

```java
xmlSchema::ComplexType createFeatureType (xml::Class ft):
    let type = new TopLevelComplexType:
        type.setName(ft.name:"Type") ->
        ... // adding of attributes as collections
        eg.element.addAll(ft.attributes.collect(e|createAttribute(e)));

Listing 4.17: Fragment of the Xtend mapping file for HML Attributes of HML Classes and HML Complex Types

Again the mapping of standard properties like name and cardinality is simple; the main effort is spent in resolving the right types. In the code section of listing 4.17 intensive use is made in overloading the mapping function 'getType'. The 'getType' is tested sequentially for the right signature and if no specialised type is found, the last, default function is used. This function resolves the name as qualified name', combining package namespace and type name.

HML Complex Type to Complex Type  The mapping code for 'HML Complex Types' is nearly the same as for 'HML Classes', except for the missing supertype.

HML Relationship  As explained in chapter 3.5, the mapping of 'HML Relations' to xml:complexType is simple in principle, but is using very uncommon types in the GML Schema target. In the practical implementation
of listing 4.18, several problems occur in the transformation to those elements with the used version of the OpenArchitectureWare XSD meta model adapter.

Firstly, the usage of the `gml:targetElement`, a simple `xsd:string`, is the only GML imported element on specification level. This requires to load the complete GML Metamodel into the transformation framework. Due to its complexity and size, the GML metamodel can currently not be parsed into memory completely. This problem could be easily solved with small adoptions of GML Schema.

Secondly, the usage of XML tags nested in the `xsd:appinfo` tag is not supported in the used version of the XSD meta model adapter.

To handle these problems without manipulations of GML Schema and without specific adaption of the OpenArchitectureWare XSD adapter, a simple workaround was chosen. Instead of the nested `gml:targetElement`, the standard `xsd:documentation` element is created, carrying the same information of the related class. In a following simple XSLT script, the elements are replaced.

```xml
xmlSchema::ComplexType createRelationType (hml::Relation ft):
    let type = new TopLevelComplexType:
        switch {
            case (ft.name.length > 0): type.setName(ft.name+"Type")
            default : type.setName(ft.fromRole+"_"+ft.toRole+"Type")
        } ->
        type.setComplexContent(createRelationComplexContentDefault(ft)) ->
        type;

    // Elements with rolename, a ReferenceType and an appinfo, which type should be linked
    LocalElement createAssoEnd(String name, int cardMin, int cardMax, hml::Class class):
        let el = new LocalElement:
            el.setName(name) ->
            el.setMinOccurs(cardMin) ->
            el.setMinOccurs(cardMax) ->
            el.setTypeName(createQName(GmlNamespac, "ReferenceType")) ->
            el.setAnnotation(createAnnotation(class)) ->
            el;

    // Annotation type
    AnnotationType createAnnotation(hml::Class class):
        let an = new AnnotationType:
            an.documentation.add(createDocu(class)) ->
            an.appinfo.add(createAppinfo(class)) ->
            an;

    // Use the Documentation instead of GML appinfo
    DocumentationType createDocu(hml::Class class):
        let doc = new DocumentationType:
```
4.4. REALISATION OF VERTICAL TRANSFORMATIONS

In the main function `createRelationType` the base `xsd:complexType` is created and if no name is given in the source model, a combination of ‘fromRole’ and ‘toRole’ is chosen.

Execution of the Transformation

The transformation is executed by calling the OpenArchitectureWare workflow runner with the workflow file. Again, the transformation process takes only a few seconds depending on the size of the models. The GML Schema can be opened and validated with an arbitrary XML editor.

4.4.2 Generation of a HTML Documentation from HML Conceptual Models

In this chapter an additional vertical transformation to a basic model documentation is realised. This second transformation focuses on the practical needs in daily data handling. At least as important as processable formats like GML schema or database schemas are human-readable documentations; e.g. in the INSPIRE process the official regulations and the feature catalogue are mainly derived from the conceptual model.

In this work a limited html documentation with all HML ‘Classes’, HML ‘Types’ and HML ‘Relations’ in tabular form is generated for HML models.

Technically this transformation is of interest, because a template-based model-to-text approach is used. The used template language is OpenArchitectureWare XPand.

Definition of the Workflow

The workflow for this model-to-text transformation is basic, because no validation and only one transformation step is included. The validation is skipped by intention, to allow users to generate model documentation already at an early stage, where the models may still be incomplete.

In listing 4.19 the execution component ‘oaw.xpand2.Generator’ is called with the parameters of the HML meta model, a location, where the documentation shall be stored and the template file ‘HmlDocu_html’ with the input model slot ‘hmlmodel’.

```xml
<component id="trafo" class="oaw.xpand2.Generator">
  <metaModel id="goMM">
    class="org.openarchitectureware.type.emf.EmfMetaModel">
```
Definition of the Template

The OpenArchitectureWare XP and template language, mentioned in chapter 4.1.5, offers a simple mechanism to mix static text with parts, that are dynamically created from an input model. The language is integrated in the ow framework and can use all of its options like integration into workows, external call of XTend or even Java functions.

In a XP and template one or more procedure are dened, which can be called from outside (the workow execution) or from within the template. The procedure consist of control code, responsible for the dynamic component, and static content, which is not interpreted by the template engine. All XP and code within the templates is marked with guillemets.

Listing 4.20: Snippet of a XP and template, which creates an empty html file for a HML model

Listing 4.20 illustrates the basic notation of XPand. The rst line starts with the XPand «IMPORT» expression to make the HML meta model available for processing. The «DEFINE ... FOR» statements deine procedures with input properties, in this case the procedure root to handle the HML root 'package'. Beside the mixture of static HTML content and access to properties of the HML package, another procedure (the package procedure in listing 4.21) is called by the «EXPAND» statement.
4.4. REALISATION OF VERTICAL TRANSFORMATIONS

Listing 4.21: Snippet of a XParad template. The template functions create the documentation for a HML package with all HML classes as tables.

Listing 4.21 shows especially the processing of collections with the 「FOR-EACH」 keyword. For all HML ‘classes’ of a package, a new table is created and for all HML ‘attributes’ of a ‘class’, a table row is defined. This kind of template processing makes it easy to create structured documentations. The complete XParad template can be found on the CD in Annex ??.

4.4.3 Results

GML Schema

The automated generation of GML Schema from HML models is feasible and can be realised with acceptable efforts. Compared to transformations between different conceptual modelling languages as described in chapter 4.3 more problems and a higher complexity arise. But not all problems are related to the meta model approach:

- The conceptual differences between source and target language are huge. The input language HML offers the possibilities for very simple conceptual modes. XML Schema is a very large and elaborated language for general purposes of format descriptions. As those concepts must be bridged, complex mappings are required.

- The strict tree like structure of XML must also be followed in the creation process; this causes partly deeply nested mapping code.
• The definition of relationships in the INSPIRE usage of GML Schema is complex and problematic as described in chapter 3.5. The required usage of nested \texttt{xsd:appinfo} is not supported by the used transformers.

• In contrast to the other components of the OpenArchitectureWare framework, the XML Schema adapter is rather incomplete and weakly documented. This was problematic in simple type restrictions and the definition of relationships.

Despite to the problems for this vertical transformation, the approach offers comparable performance and quality to other actual software products, like ‘ShapeChange’ or ‘FullMoon’ (see chapter 4.3).

\textbf{HTML Documentation}

The generation of documentations with template-based transformations is very easy and direct. No major challenges occur in the creation and the execution of the templates.

Similar like other template approaches, XPand is very well suited to generate output with mainly static text and compact control statements. The direct access to model elements or values and the easy handling of collection types keeps the code clear and structured. For very complex and huge mappings, the code is likely to get too mixed and difficult to maintain, which is a problem of all template-based approaches.

The design of mapping templates is is supported by OpenArchitectureWare editors with code highlighting, auto completion and the possibility to use code debuggers on execution.

\section*{4.5 Summary}

Following the conceptual analysis of chapter 3, technical implementations are described in this chapter.

For the implementation a combination of different frameworks and tools from the Eclipse Modelling Platform, the leading open source implementation for model-driven technologies, and the OpenArchitectureWare components for model transformation are used. Both frameworks are developed and maintained by an active and huge developer community\footnote{The platforms are currently evolving very fast; e.g. OpenArchitectureWare is now integrated into the Eclipse Modelling Platform, an alternative graphical tooling is established and it can be expected that upcoming QVT implementations replace some model-to-model languages.}, so that bugs and implementation problems are likely to be solved in upcoming releases.

In the first part (chapter 4.2), the abstract syntax of the domain-specific language HML is used to create supportive tools for conceptual geospatial
modelling. The Ecore model of the Humboldt Modelling Language is combined with models for graphical representation and models for basic behaviour. The HUMBOLDT GeoModel Editor is directly and automated generated from those combined models. This graphical editor is comparable in layout and functionality to standard UML modelling tools but is reflecting exactly the underlying HML approach. The user can rely on a simple, yet sufficient environment.

To link domain specific modelling approaches to other, established languages, horizontal transformations are needed. In chapter 4.3 the pseudo-code mappings for a HML model to UML model transformation (chapter 3.4) are translated to an executable transformation process with the OpenArchitectureWare XTend language. The creation of the mapping code is mostly simple, because structure and syntax of XTend is straightforward and the creation of mappings is supported by editors from the framework. Some mapping efforts arise from the huge and complex UML meta model and the cumbersome handling of UML profiles.

Also vertical transformations from HML models to XML Schemas are implemented as OpenArchitectureWare transformations in chapter 4.4. Similarly as for the horizontal transformations, the pseudo-code mapping rules from chapter 3.5 could be translated to executable XTend mapping code quite directly. But as already evident in the pseudo-code, the effort for this implementation is rather high, because simple language elements from the HML input model must be mapped to large XML Schema sequences.

In addition to the technical mappings, also a model-to-text transformation is carried out, which directly translates the conceptual HML model to a human-readable HTML documentation. For this process, the OpenArchitectureWare template language XPand is used.
Chapter 5

Summary of the Results

The working assumptions about the usability of meta modelling techniques from the field of model-driven software development for conceptual geospatial modelling have been proven by conceptual developments and practical implementations.

Standard meta modelling approaches can be used to define conceptual schema languages for the geospatial domain

The core aspects of the geospatial modelling language INTERLIS 2 have been remodelled with the standard eMOF/ECore meta modelling language\(^1\). This was easily possible, especially as INTERLIS 2 is very clearly specified and documented.

Additionally, a new geospatial modelling language was developed from a set of user requirements; a new ‘domain-specific modelling language’ was set up. For this language (called the ‘Humboldt Modelling Language’ HML), standard patterns from UML and INTERLIS 2 were considered and additional information elements were introduced to exactly fit the language to the specific needs of the Humboldt project. The resulting language is simple to understand and use, but completely sufficient for all requested modelling tasks.

Tools of model-driven software development can be used to generate advanced graphical editors

Functionally rich editors can be defined from a combination of language meta model, graphical and behavioural models in a pure declarative way. Standardised model-driven development processes and frameworks like the

\(^1\)The abstract syntax of MADS core parts was also described with Ecore and is listed in annex C. The abstract syntax of the commonly used UML, the base language for the ISO/TC-211 conceptual modelling approach, is defined in eMOF and Ecore implementations are available.
Graphical Modelling Framework GMF lead to highly sophisticated and user-friendly modelling tools. Such a graphical model editor was created for the developed domain specific language ‘HML’. Directly from the defined meta model and the additional graphical and behavioural models the ‘HUMBOLDT GeoModel Editor’ was generated in an automated process. The editor offers similar functionalities as standard graphical UML editors, but is highly adjusted to the requirements of the HUMBOLDT project to simplify the conceptual modelling process.

**Model Quality can be improved with model validation mechanism**

For modelling languages based on formal meta models, validating parsers and structural checkers can easily be derived in model-driven procedures. Also additional validation rules on naming conventions, minimum requirements and style guidance can be included in a declarative way. Such validation components were included in HUMBOLDT GeoModel Editor. The graphical model editor supports context sensitive menus and on-request validation with graphical feedback. This ensures model quality and simplifies modelling processes by direct feedback, esp. compared to procedures with subsequent validation or manual consistency rules.

**Transformations between modelling languages can be created with standard transformation tools**

Transformations between different conceptual geospatial modelling languages are handled in literature and current implementations rather rarely. Up to now, every modelling approach was designed as singular environment without bridging to others. By that, it is currently impossible to transform between languages like INTERLIS 2, GeoUML, MADS or UML with ISO/TC-211 rules.

In this work a transformation between the domain-specific HUMBOLDT modelling language and profiled UML according to ISO/TC-211 was set up. Meta model aware tools for the definition of the mappings and the execution of the transformation were used. This was possible, because both languages are based on Ecore (eMOF). The transformations were highly efficient and especially simple were direct mappings between language concepts were found.

**Standard vertical transformation tools support the creation of typical targets like GML schemas, documentations or database schemas**

In this work, transformations were set up for the creation of GML schema and a basic html documentation from conceptual models designed in HML. For the transformations, two approaches were chosen; model-to-model trans-
formations for the creation of GML-Schema and template-based transformations for a html model documentation.

The model-to-model transformation turned out to be quite difficult. On one hand, the used framework could not completely handle the complexity of GML schemas. On the other hand the conceptual differences between the very limited, geospatially-oriented HUMBOLDT modelling language and the general format description XML schema are huge and only solvable by complex mappings. The complexity of GML schema is problematic for all transformation processes and often other, more suitable data formats should be prefered.

The template-based approach to generate documentations was easy to realise. The syntax is simple and the strong tool support allowed to define a model-to-text engine with very little effort.
Chapter 6

Discussion

In this work several process steps in the context of geospatial conceptual modelling were carried out by using a meta modelling approach. Modelling tools were derived from language definitions with eMOF/eCore modelling tools and different language-to-language as well as language-to-text transformations were executed (see figure 6.1).

![Diagram](image)

**Figure 6.1:** Language-oriented development based on meta modelling can be used at different stages of the geospatial modelling procedures. The usage of useful modelling languages leverages the initial efforts for the domain experts; according tools support the creation and ensure model quality; language transformations allow flexibility in the choices of modelling languages as well as offer support for vertical transformations to logical or physical models; model-to-text transformations give help in the creation of the final target products like documentation, feature catalogues or transfer formats.

There are implementation alternatives to the presented approaches, like the tools and transformers reported in chapter 2 ‘State of the Art’ with their specific advantages and disadvantages. In the presented implementation sev-
eral problems occurred, which were especially caused by the rather young open source frameworks. The developer communities of the used open source approaches are very active, either in giving support via the internet fora or in fixing bugs, but still not all implementation issues could be solved within the time frame of this thesis. The most prominent issue is related to the complex transformation towards GML Schema, where a postprocessing as stylesheet transformation must be executed after the main transformation.

Another serious problem for productive use is the ongoing fast development that can generate maintenance problems. The used OpenArchitectureWare project is now integrated into the main Eclipse modelling trunk and the transformation projects must be adapted to be runnable within the new releases. The same can be expected for the transformation code, where there is actual strong development towards complete QVT implementations to replace the other mapping languages.

Despite to those technical issues the working thesis ‘Meta Modelling Techniques will Improve Usability and Interoperability for Conceptual Geospatial Modelling’ can be stated true. All sub statements are valid and meta modelling can act as a connecting technology. Especially, the topics of ‘domain-specific modelling languages for geospatial modelling’, the ‘specialised modelling tools’ and the ‘language-to-language transformations’ deserve special attention, as they open new possibilities for the geospatial community.

6.1 Definition of Domain Specific Languages

The definition of small, well-adjusted domain-specific languages is one key concept of modern model-driven software development and can be transferred to conceptual geospatial modelling. The new languages can vary from simple limitation or extensions of existing languages over language mixtures to complete new developments. Usually it is preferable to base on already established approaches [Budinsky et al., 2010] to leverage efforts in the language design and for the users to learn it.

The language designed in this work adopts key concepts of UML and INTERLIS 2. Compared to UML it is limited, but natively offers basic geometry types, ways of handling measurements and differentiates between real world objects and data structures. Compared to INTERLIS 2, several features like multilingualism or the complex associations are omitted and the terminology is adjusted. The formalism of the new language HML follows the standardised approach of model driven software development - eMOF/Ecore. With this approach, the abstract syntax is described straight forward and sufficiently for technical implementations. But for end users the language must be supplemented with a concrete syntax as described in chapter 4.2.3 and the semantic explanations of chapter 3.3.2.
6.2. TOOLS

The design of well-adjusted languages can be seen as alternative to other strategies of current practise, such as:

- **UML Profiles:** UML profiles (see chapter 2.2.1) are used as lightweight mechanism to adopt the meta model of UML within the model. The possibilities are very limited and focus on adding properties and stereotypes to existing elements. Limitations or structural changes in the language are hardly possible\(^1\). The ISO/TC-211 profile of UML is focussed on the design of GML and overloads the concepts of UML with the General Feature Model ISO/TC-211 [2005b]\(^2\).

- **Textual Rules:** Textual explanations on the use of modelling languages are essential to describe semantics or rules of usage and give modelling examples. But sometimes those rules are expanded to overload the language itself (e.g. INSPIRE Drafting Team "Data Specifications" [2007]). Such rules are e.g. special naming conventions that change the interpretation of a Class or an Association and can hardly be validated by automated, tool-based checking means.

The alternatives should be judged individually. The usage of UML profiles and textual rules shifts considerable efforts to the modelling experts as end users. They need profound knowledge not only of their special domain, but also on the Unified Modelling Language, the basics of the applied profile and the additional rules.

The creation of a new language puts more effort in the preparatory phase. Even for simple or reused approaches, the new language must be developed, formalised and described. But as presented in this work, modelling experts can rely on simple, well-adjusted languages and supporting tools guarantee a minimum model quality.

6.2 Tools

Tools are of major importance for conceptual geospatial modelling. Suitable editors support the creation process and can even compensate complexity in modelling approaches. In conceptual geospatial modelling there is a considerable lack of supportive editors. Either tools tend to be outdated with a lack of supportive user communities or offer only general modelling support for UML, which is not tailored to data modelling but supports software development in general.

\(^1\)Restrictions are only possible by adding cumbersome OCL constraints.

\(^2\)A detailed evaluation on the limitations of representing the General Feature Model with UML can be found at Einspanier [2005]
CHAPTER 6. DISCUSSION

Graphical Model Editors  The design, implementation and test of interactive model editors is a resource demanding task. Most new products therefore avoid complete new developments and rely on existing frameworks or product platforms that offer all basic functionality on file-handling, help systems and graphical user interfaces. One of the most popular ones is the Eclipse platform, which is used for numerous tools.

With the Eclipse Graphical Modelling Framework (GMF), the production of adapted graphical editors for that platform can be simplified even more. Compared to traditional programming with re-used libraries, the meta-model-aware approach offers fast development circles and high flexibility, especially for cross-cutting changes that affect internal structures, access and visualisation\(^3\). This technique is current standard and used for several commercial and open source editors, like Papyrus UML\(^4\), IBM Rational Modeller\(^5\) or MagicDraw\(^6\) and the language supports for SysML and BPMN.

The realised HUMBOLDT GeoModel Editor for the designed domain specific language HML is also build on Eclipse GMF. The development is straightforward and main components are derived automatically from the eCore language definition, so that implementation effort is strongly reduced. The resulting tool offers graphical modelling in a UML-like notation with colour support, context-sensitive menus, build-in validation and layout facilities.

The combination of fast and flexible development on latest technologies and excellent modelling support is without true alternatives. The standard UML editors only give good support for basic UML modelling and the editors for the specific geospatial languages like INTERLIS 2 or MADS lack active development communities to keep them up-to-date.

Model Validators  Although of major importance, model validation is often neglected in geospatial modelling. It ensures a minimum quality for further processing and should give direct feedback to the domain expert.

In this thesis, the basic validator is implicitly derived from the meta model and supplemented with declarative rules for name spelling and uniqueness of names. These rules are integrated in the transformation engines and the graphical editor, where it gives direct feedback to the user.

A similar tight integration can be found for INTERLIS 2, where a validating parser can be called directly from the graphical modelling tool. But similar to the graphical editor the tools are less flexible, so that additional rules in INTERLIS 2 must be coded via the programming interfaces.

\(^3\)The complexity to modify traditional editors with conceptual changes is e.g. documented in Najar [2006].

\(^4\)http://www.papyrusuml.org

\(^5\)http://www-01.ibm.com/software/de/rational/

\(^6\)http://www.magicdraw.com/
For both groups of tools, the used meta modelling approach with rich frameworks for code production are currently without true alternatives. The functional richness, the adaptability and the speed of development pose the technique as preferable tool. Even if no changes in geospatial modelling languages or transformations are planned, the creation of new model editors and validators based on the meta model approach should be considered.

6.3 Horizontal Language-to-Language Transformations

In the implementation a transformer from HML models to ISO/TC-211 compliant UML models was realised. Both modelling languages share the same meta modelling approach eMOF/Ecore and partly overlap in concepts. The mapping rules between the languages were easy to define, especially where language concepts were similar; e.g. for classes and complex data types, but more difficult for the spatial data types.

The language-to-language transformations are not common in the geospatial world. But they will allow the flexible usage of different modelling languages varying from general purpose languages like UML over geospatial languages like INTERLIS 2 up to small languages like HML. The best available tools could be used for each specific purpose and interoperability would be strengthened.

6.4 Outlook

The presented meta modelling approach can act as a kind of base technology for conceptual geospatial modelling, similar to its usage for model driven software development. Based on the definition of modelling languages multiple tools and transformers can be created to support modelling experts as well as GIS and software developers.

Actual modelling initiatives like European INSPIRE data specifications (Joint Research Centre [2011]) or the Swiss federal act on geoinformation (Bundesversammlung der Schweizerischen Eidgenossenschaft [2007]) illustrate on national and international level the importance of conceptual modelling and the involvement of domain experts. It is the primary task of the conceptual models to act as discussion base and communication platform between technical and domain experts. One potential of the meta model approach is in the optimal adjustment of this communication functionality; the languages are focussed to optimally gain the knowledge of the domain experts and to be easily processable. Derived graphical modelling tools and validation mechanisms support these functionalities.

The transformation capabilities between modelling languages based on standardised meta model approaches help to overcome the singularities of
different modelling approaches. It is possible to offer domain experts easy languages with good tool support and transfer those models to a more sophisticated modelling approach and derive the end products. An 'ecosystem' of models and tools could be established to support developers as well as modellers (see figure 6.2).

Assuming ISO/TC-211 UML modelling (as international accepted standard) to be a kind of foundation system, different language variations for optimised processing could be defined in future:

- A simplified modelling language, based on UML, but reflecting the concepts of the General Feature Model directly in the language (see Einspanier [2005]) without profiles and the cluttered concepts of UML. This language can directly be used to generate a user friendly modelling editor as a standard tool.

- A product-oriented modelling language with adequate transformer to GML Schema. The standard vertical transformation process to GML Schema can also be supported by simplified languages that focus on the production of GML Schema. Those modelling languages are reduced in size and the models form a kind of bridge between conceptual and logical models.\(^7\)

- Arbitrary domain-specific languages. If the modelling domain is well defined and a sufficient user need exist, the definition of additional, more specialised languages can be useful. A compelling example would be the definition of an INSPIRE modelling language build on the requirements formulated in the Generic Conceptual Model INSPIRE Drafting Team "Data Specifications" [2007].

- For the creation of highly specific target products, also adjusted models might be useful to simplify the processing steps. Those intermediate models might already be enriched by target specific information and can therefore already be regarded as logical models.

Of course several variations of figure 6.2 are possible. Vertical transformations to end products might be applied directly on the editor models, and different already existing languages like MADS, INTERLIS 2 or GeoUML might be adopted to fit to the meta model approaches.

With regard to other research activities, especially in the field of semantic transformations (e.g. [Staub, 2009] or [Orchestra Open Architecture and Spatial Data Infrastructure for Risk Management, 2007]), language-based tools might likely also push technical developments for data transformations. Transformation languages on the conceptual model level (either textual or

\(^7\)Such intermediate models are typical for model-driven software development, with multiple small transformation steps.
Figure 6.2: Technical capabilities of tools and transformation can allow more variability in modelling approaches. On one hand a better support for domain experts with adapted languages for graphical modelling and on the other hand easy processable models for transformations to end products could. The ‘general purpose geospatial modelling approach’ of ISO/TC-211 might act as foundation or reference system.
graphical), can easily be described with meta model approaches. Generated editors might support the mapping definitions and transformation code could be derived with means of model-driven software development.

Technically further development in the field of the model-driven software development can be expected. In early stages and not considered in this work are especially the textual frameworks like XText\(^8\), that support the creation of arbitrary textual modelling languages and according tools. Also tools for model comparison\(^9\), model repositories\(^10\), model querying\(^11\) etc. are under development. Such developments must be observed in future as those projects address urgent topics in the maintenance of conceptual geospatial modelling.

\(^8\)http://www.eclipse.org/Xtext/

\(^9\)EMF Compare: http://www.eclipse.org/emf/compare/

\(^10\)EMF CDO http://www.eclipse.org/cdo/

\(^11\)EMF Query http://www.eclipse.org/modeling/emf/?project=query
Appendix A

INTERLIS 2 - Meta Model

A.1 INTERLIS 2 - Meta Model Overview

Figure A.1: Complete abstract syntax, developed for the selected INTERLIS 2 language subset.
A.2 INTERLIS 2 - Meta Model

```java
@namespace(uri="http://www.ethz.ch/hml/interlis", prefix="interlis")
package Interlis;

class Model {
    attr String[1] name;
    attr String[1] owner;
    attr String[1] version;
    ref Model translation;
    val Topic[*] topic;
}
class Topic {
    attr String[1] name;
    attr boolean[1] "abstract = false;
    attr boolean[1] final = false;
    ref Topic "extends;
    val Class[*] "class;
    val AttributeType[*] domain;
    val Unit[*] unit;
}
class Structure extends AttributeType, ClassOrStructure {
    attr String[1] name;
    attr boolean[1] final = false;
    attr boolean[1] "abstract = false;
    ref Structure "extends;
    val Attribute[*] attribute;
}
class Class extends ClassOrStructure {
    attr String[1] name;
    attr boolean[1] final = false;
    attr boolean[1] "abstract = false;
    val Attribute[*] attribute;
    ref ClassOrStructure "extends;
}
class Attribute {
    attr String[1] name;
    attr boolean[1] extended = false;
    attr boolean[1] final = false;
    ref AttributeType[1] type;
    attr Container container;
    val Cardinality cardinality;
}
abstract class AttributeType {
}
class TextType extends BaseType {
    attr int length;
    ref TextType "extends;
}
class EnumerationType extends BaseType {
    ref EnumerationType "extends;
    val EnumerationType[*] contains;
```
A.2. INTERLIS 2 - META MODEL

```java
class NumericType extends BaseType {
  attr double min;
  attr double max;
  attr int integerPlaces;
  attr int decimalPlaces;
  attr boolean[1] cyclic = false;
  ref Unit unit;
  ref NumericType ~extends;
  ref Refs system refs system;
  ref Axis axis;
}
class BaseType extends AttributeType {
  attr String[1] name;
  attr boolean[1] final = false;
  attr boolean[1] ~abstract = false;
}
class CoordinateType extends BaseType {
  val NumericType[1..3] coord;
}
class Cardinality {
  attr int[1] min = 0;
  attr int[1] max = 1;
}
enum Container {
  BAG = 0;
  LIST = 1;
}
class Unit {
  attr String[1] name;
  attr String shortName;
  attr boolean[1] ~abstract = false;
  ref Unit ~extends;
}
class Coords system extends Refs system {
  val Axis[1..3] axis;
}
class Refs system {
  attr String[1] name;
  attr boolean[1] ~abstract = false;
}
class Axis {
  ref Unit unit;
}
class Association extends RoleEnd {
  ref Association ~extends;
  attr String name;
  attr boolean[1] ~abstract = false;
  attr boolean[1] final = false;
  attr boolean[1] extended = false;
  val Attribute[∗] attribute;
  val Role[2..∗] role;
}
class Role {
```
Listing A.1: Abstract syntax for the selected subset of INTERLIS 2 as Ecore Model, listed in EMFATIC notation (see Eclipse Foundation [2010])
Appendix B

HUMBOLDT Modelling
Language - Meta Model

B.1 HML - Meta Model Overview

Figure B.1: Complete abstract syntax of the Humboldt Modelling Language (HML).
B.2 HML - Meta Model

```java
@namespace(uri="http://www.ethz.ch/hml", prefix="hml")
package hml;

class Model extends ModelElement {
   val Package[1] modelpackage;
   attr String[1] creator;
   attr String[1] version;
   attr double[1] timestamp;
}

class Unit extends DefinitionElement {
   ref Unit superunit;
   attr String definition;
   attr String shortUnit;
}

abstract class Type extends DefinitionElement {
}

abstract class ModelElement {
   attr String[1] name = "";
}

abstract class DefinitionElement extends ModelElement {
   attr boolean[1] base = false;
   attr boolean[1] "abstract = false;
}

class Package extends ModelElement {
   val Class[*] classes;
   val Package[*] subpackages;
   val Type[*] types;
   val Unit[*] units;
   val Relation[*] relationships;
   val Coordinatesystem[*] coordinateSystems;
   val Coordinate[*] coordinates;
}

class Class extends ModelElement {
   attr boolean "abstract;
   val Attribute[*] attributes;
   ref Class superclass;
}

class Attribute extends ModelElement {
   ref Type[1] type;
   attr int[1] minCardinality = 1;
   attr int[1] maxCardinality = 1;
   attr Boolean[1] "ordered = "false";
```
abstract class Relation extends ModelElement {
    ref Class [1] fromClass;
    attr int [1] fromCardinalityMin = 1;
    attr int [1] fromCardinalityMax = 1;
    attr String fromRole;
    ref Class [1] toClass;
    attr int [1] toCardinalityMin = 1;
    attr int [1] toCardinalityMax = 1;
    attr String toRole;
}

class Aggregation extends Relation {
}

class Association extends Relation {
}

class Composition extends Relation {
}

class StringType extends SimpleType {
    attr int length;
    ref StringType supertype;
}

class NumericType extends SimpleType {
    attr int length;
    attr int precision;
    attr float min;
    attr float max;
    attr boolean cyclic;
    ref Unit unit;
    ref NumericType supertype;
}

class Enumeration extends SimpleType {
    val EnumItem [*] item;
    ref Enumeration supertype;
    ref Enumeration[*] merge;
}

class EnumItem {
    attr String [1] text;
}

class SimpleType extends Type {
    attr String definition = "";
}

class ComplexType extends Type {
    val Attribute[*] attributes;
    ref ComplexType supertype;
}
class Coordinatesystem extends DefinitionElement {
  attr String definition;
  val Axis[0..3] axis;
}
class Axis extends ModelElement {
  attr String definition;
  ref Unit[*] allowedUnits;
}
class Coordinate extends DefinitionElement {
  val CoordinateNumericType[0..3] elements;
  ref Coordinate supercoordinate;
}
class CoordinateNumericType extends NumericType {
  ref Axis axis;
}
class UnionType extends Type {
  !ordered ref Type[*] unionTypes;
  ref UnionType supertype;
}
class PointType extends SimpleGeometryType {
  ref Coordinate[1] position;
  ref PointType supertype;
}
class LineType extends SimpleGeometryType {
  ref PointType[1] nodes;
  ref LineType supertype;
}
class PolygonType extends SimpleGeometryType {
  ref PolygonType supertype;
  ref LineType[1] rings;
}
class GeometryType extends Type {
}
class MultiGeometryType extends GeometryType {
  ref SimpleGeometryType[*] primitives;
}
abstract class SimpleGeometryType extends GeometryType {
}

Listing B.1: Abstract syntax for the developed Humboldt Modelling Language (HML) as Ecore Model, listed in EMFATIC notation (see Eclipse Foundation [2010])
Appendix C

MADS - Meta Model

C.1 MADS - Meta Model Overview

Figure C.1: Complete abstract syntax, developed for the selected MADS language subset.
C.2 MADS - Meta Model

```java
@namespace(uri="http://www.ethz.ch/hml/mads", prefix="mads")
package mads;
class ObjectType extends InformationElement {
  val Attribute[*] attribute;
  val Key key;
}
abstract class Attribute {
  attr EString[1] name;
  attr EBoolean[1] mandatory = false;
  val Cardinality[1] cardinality;
}
class SimpleAttribute extends Attribute {
  ref Type[1] type;
}
class ComplexAttribute extends Attribute {
  val Attribute[+] attribute;
}
abstract class Type {
}
abstract class Relationship extends AbstractRelationship {
  val Role[2..*] role;
  val Relationship superRelation;
}
class Role extends AbstractRole {
  val Cardinality[1] cardinality;
}
class Cardinality {
  attr int[1] cardinalityMin = 0;
  attr int[1] cardinalityMax = 1;
  attr Container container = "true";
}
enum Container {
  set = 0;
  bag = 1;
  list = 2;
}
class Key {
  ref Attribute[*] keyAttribute;
  ref AbstractRole[*] keyRole;
}
class MultiAssociationRelationship extends AbstractRelationship {
}{
  val MultiAssociationRole[2..*] role;
  val MultiAssociationRelationship superRelation;
}
class MultiAssociationRole extends AbstractRole {
  val Cardinality[1] cardinalityObject;
  val Cardinality[1] cardinalityAsso;
}
abstract class InformationElement {
  attr EString[1] name;
}
C.2. MADS - META MODEL

```java
val Attribute[*] infoAttribute;
}
class AbstractRelationship extends InformationElement {
  val Key key;
}
abstract class AbstractRole {
  ref ObjectType[1] objecttype;
  attr EString name;
}
class Inheritance {
  attr boolean overlapping = false;
  ref ObjectType[1] supertype;
  ref ObjectType[+] subtype;
}
class Integer extends Type {
}
class Real extends Type {
}
class String extends Type {
}
class Boolean extends Type {
}
class Geo extends Type {
}
class ComplexGeo extends Geo {
}
class SimpleGeo extends Geo {
}
class Point extends SimpleGeo {
}
class Line extends SimpleGeo {
}
class Surface extends SimpleGeo {
}
class PointBag extends ComplexGeo {
}
class LineBag extends ComplexGeo {
}
class SurfaceBag extends ComplexGeo {
}
class OrientedLine extends Line {
}
class SimpleSurface extends Surface {
}
class OrientedLineBag extends LineBag {
}
class SimpleSurfaceBag extends SurfaceBag {
}
abstract class Constraint {
  attr EString name;
}
abstract class TopoConstraint extends BinConstraint {
}
abstract class BinConstraint extends Constraint {
```
APPENDIX C. MADS - META MODEL

Listing C.1: Abstract syntax for the selected subset of MADS as Ecore Model, listed in EMFATIC notation (see Eclipse Foundation [2010])
Appendix D

CD Content

The CD contains:

- Runnable version of the HUMBOLDT GeoModel Editor for Microsoft Windows
- Sample modelling project on hydrological topic
- Sample modelling project on street data
- Sample modelling project for INSPIRE theme on ‘Geographical Names’
- Complete distribution of Eclipse modelling environment for Microsoft Windows
- Complete Eclipse projects for the generation of the HUMBOLDT GeoModel Editor
- Complete Eclipse projects for the described horizontal transformation
- Complete Eclipse projects for the described vertical transformations
- Eclipse project for MADS meta model
- Eclipse project for INTERLIS 2 meta model
## Appendix E

### Glossary

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Abstract Syntax</strong></td>
<td>Description of the elements forming a modelling language and their relations.</td>
</tr>
<tr>
<td><strong>BPMN</strong></td>
<td><strong>Business Process Model and Notation.</strong> A modelling language defined by the Object Management Group (OMG) to specify workflows and processes for businesses or companies.</td>
</tr>
<tr>
<td><strong>CIM</strong></td>
<td><strong>Computational Independent Model.</strong> The computational independent model in model driven software development describes the target system or data structures independent of any implementation issues. It is solely focussing on the user perspective towards the system or data structure.</td>
</tr>
<tr>
<td><strong>Conceptual Model</strong></td>
<td>Formal description of a software system or data format from the end user’s perspective. A conceptual model should be easy to understand for non-software experts and exclude any computational aspects.</td>
</tr>
<tr>
<td><strong>Concrete Syntax</strong></td>
<td>The notation of a modelling language. The abstract and the concrete syntax together define a modelling language.</td>
</tr>
<tr>
<td><strong>DSL</strong></td>
<td><strong>Domain Specific Language.</strong> Modelling Language with a high specialisation for a specific field of use or a certain user group.</td>
</tr>
<tr>
<td><strong>Eclipse</strong></td>
<td>Popular open source software development tool and framework.</td>
</tr>
<tr>
<td><strong>Ecore</strong></td>
<td>Metamodel of the Eclipse Modeling Framework (EMF).</td>
</tr>
<tr>
<td><strong>EMP</strong></td>
<td><strong>Eclipse Modeling Project.</strong> A main project of Eclipse to support techniques of model driven software development. EMP is the coordinating project for the subprojects EMF, GMF and several others.</td>
</tr>
</tbody>
</table>
EMF  **Eclipse Modeling Framework.** A modelling framework for the creation of modelling languages based on Ecore, model transformations and supportive tools.

EMOF  **Essential Meta-Object Facility.** Subset of MOF.

ERM  **Entity Relationship Model.** A conceptual modelling approach describing reality with similar ‘entities’ and their relationships. The ERM is closely linked to the idea of relational databases.

ETL  **Extract, Transform, Load.** Processes and supporting tools for the integration of heterogeneous datasources into one target system; especially used in the field of data warehouses.

Encoding  The translation of information into computer readable formats. In the context of this work, encoding describes the translation of conceptual models into transfer formats.

GCM  **Generic Conceptual Model.** Model packages used by several INSPIRE themes. Additionally to those packages general rules for conceptual modelling are defined.

GEF  **Graphical Editing Framework.** An Eclipse based toolkit for the creation of interactive drawing components, like diagram editors or visualisations.

GFM  **General Feature Model.** Part of ISO 19109 ‘Rules for Application Schema’. The GFM introduces the general modelling elements according to ISO.

GMES  **Global Monitoring for Environment and Security.** European initiative supporting the use of earth observation and in-situ data.

GML  **Geography Markup Language.** An OGC standard to define geographic data formats on base of XML. The concrete data format must be specified by an according GML Schema.

GML Schema  An XML Schema to define a specific GML data format

GPL  **General Purpose Language.** Modelling language with a very wide scope.

HML  **Humboldt Modelling Language.** A domain specific modelling language defined by the author to support conceptual geospatial modelling in the Humboldt project.
IDL  **Interface Definition Language.** A language defined by OMG for the specification of interfaces in distributed software systems.

INSPIRE  **Infrastructure for Spatial Information in Europe.** The INSPIRE Initiative is a legal act of the European Parliament to facilitate the exchange of spatial information by building up a common spatial data infrastructure (SDI).

INTERLIS  Language and Toolset to describe spatial information. INTERLIS offers a conceptual modelling language, especially for the definition of spatial datasets, a toolkit for validation and processing as well as specific data transfer formats.

ISO/TC 211  Standardisation working group on topics related to geoinformation.

Jet  **Java Emitter Template.** A template based mechanism for M2T transformations.

Logical Model  Formal description of a software system including general design issues of the underlying computer system.

M2M  **Model to Model Transformation.** A transformation process, generating one or more target models from one or more input model. It can be differentiated between content preserving transformation, where only the formalism of the model is changed and content changing transformation, where information of the source is omitted or additional information is added during the transformation process.

MADS  **Modeling of Application Data with Spatio-temporal features.** A conceptual geospatial modelling approach especially suited for the designing applications with complex multidimensional features and spatial constraints.

MDA  **Model Driven Architecture.** Specialisation of MDSD focussing on OMG technologies.

MD(S)D  **Model Driven (Software) Development.** Techniques to generate runnable software or software fragments (like data formats, database schemas, user interfaces etc.) from models by applying transformations. In Model Driven Development, the models are used as primary element for a repeatable development and not only as sketch or documentation.

MD(S)E  **Model Driven Software Engineering.** Synonym for MDD.
<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meta modelling</td>
<td>Formal definition of modelling languages by a standard modelling approach like MOF.</td>
</tr>
<tr>
<td>Model Validation</td>
<td>Process of proving formal correctness of models.</td>
</tr>
<tr>
<td>Model Transformation</td>
<td>Using one or more input model to generate according outputs. It can be differentiated between 'Model to Model' (M2M) or 'Model to Text' (M2T) transformations.</td>
</tr>
<tr>
<td>MOF</td>
<td><strong>Meta-Object Facility.</strong> The OMG approach to meta modelling. MOF offers a self-defined language for specifying arbitrary modelling languages like UML or SysML. Two profiles are defined for MOF: CMOF (Complete MOF) and the mostly sufficient subset EMOF (Essential MOF).</td>
</tr>
<tr>
<td>OAW</td>
<td><strong>OpenArchitectureWare.</strong> Platform for model transformations based on the Eclipse Modeling Framework. It consists of the languages 'oaw Check' for model validation, 'oaw Xtend' for M2M transformation, 'oaw Xpand' for M2T transformation as well as an execution environment for chaining arbitrary transformation steps. Since Eclipse release 'Galileo' OAW is completely integrated in the Eclipse Modeling Project.</td>
</tr>
<tr>
<td>OCL</td>
<td><strong>Object Constraint Language.</strong> Language defined by OMG for specifying queries and constraints on models based on MOF defined languages.</td>
</tr>
<tr>
<td>OMG</td>
<td><strong>Object Management Group.</strong> Open industry consortium for harmonisation and standardisations in the field of software engineering, esp. in model driven software development.</td>
</tr>
<tr>
<td>OO</td>
<td><strong>Object Orientation.</strong> Technique in software development, supporting the simplification and reusability of software components. Object oriented modelling approaches are current standard in conceptual modeling.</td>
</tr>
<tr>
<td>ORM</td>
<td><strong>Object Role Modelling.</strong> A technique for conceptual data modelling, focussing on the relations between entities and their properties.</td>
</tr>
<tr>
<td>Physical Model</td>
<td>The concrete implementation model for a software system or data format.</td>
</tr>
<tr>
<td>PDM</td>
<td><strong>Platform Description Model.</strong> The platform description model in model driven software development describes the targeted implementation platform</td>
</tr>
</tbody>
</table>
PIM  **Platform Independent Model.** The platform independent model in model driven software development describes the target system or data structure considering general implementation strategies.

PSM  **Platform Specific Model.** The platform specific model in model driven software development describes the target system or data structure considering all relevant aspects of the targeted implementation platform.

SDI  **Spatial Data Infrastructure.** A network to facilitate the exchange of spatial data. The infrastructures are built as distributes systems according to the general ideas of Service Oriented Architectures.

SQL  **Structures Query Language.** Language to query, define and manipulate data in relational database systems.

SysML  **System Modelling Language.** A modelling language defined by the Object Management Group (OMG) to model complex software systems.

UML  **Unified Modelling Language.** A modelling language for conceptual and logical modelling. UML is the de-facto standard for modelling in the field of software engineering and maintained by the Object Management Group (OMG).

UoD  **Universe of Discourse.** Term describing the thematic field of an application or a data structure. The UoD can be described as the subject of a conceptual model.

UUID  **Universally Unique Identifier.** Open standard for the creation of worldwide unique identifiers.

QVT  **Query View Transform.** A M2M transformation language for MOF based languages defined by OMG.

XMI  **XML Metadata Interface.** XML representation technique for models defined in any MOF based language.

XML  **Extensible Markup Language.** A standard of the World Wide Web Consortium (W3C) for text based, hierarchical data formats. Data formats must be individually defined by according XML Schema.

XML Schema  Schema Language to define specific XML data formats. XML Schema is defined as XML itself.
XSLT  
**Extensible Stylesheet Language Transformation.** A XML based language for the transformation of XML documents into text or other XML documents.

Xtext  
An open source framework for the definition of concrete textual syntaxes for Ecore based abstract syntaxes. Initially Xtext was part of oaw, but is now integrated in the Eclipse Textual Modeling Framework (TMF).
# List of Figures

2.1 Differentiation of models into physical model, logical model and conceptual model ........................................... 21

2.2 Example of an UML Class Diagram (taken from the UML Infrastructure [OMG - Object Management Group, 2007b]) ........................................ 28

2.3 Example of a simple Profile, containing one Stereotype 'device’ for Classes with the 3 Tagged Values ‘author’, ‘color’ and ‘volume’ (taken from [OMG - Object Management Group, 2007b]) ............................................. 29

2.4 Extract from the General Feature Model (taken from [ISO/TC-211, 2005b]) ......................................................... 31

2.5 Basic geometric types of ISO-19107 and their inheritances (taken from [ISO/TC-211, 2001]) ........................................... 33

2.6 The INTERLIS approach for data exchange (adapted from [KOGIS, 2006b]) ............................................................... 35

2.7 Example of conventional mapping of related groups. Building representations for specific scales (1:15000 and 1:25000) are grouped together and mapped by the 'Corresponds' relationship. (taken from [Parent et al., 2006a]) ............................ 41

2.8 The same association between grouped objects for building with the simplifying multi-association relationship type (taken from [Parent et al., 2006a]) ...................................................... 42

2.9 MADS Is-A relationship with two inheritance lines; private and public house are mutually exclusive, while private and living house may be overlapping (taken from [Parent et al., 2006a]) ............................................. 43

2.10 MADS hierarchy of spatial data types, including the according symbol (taken from [Parent et al., 2006a]) ................................................. 44

2.11 MADS spatial object and intersection (taken from [Parent et al., 2006a]) ................................................................. 44
2.12 Transformations related to conceptual data models: (1) illustrates a horizontal transformation between datasets of different structures. The definition of the data transformation is done on the conceptual (or logical) level; (2) illustrates a vertical transformation to generate a logical or physical data schema from a conceptual model; (3) shows a horizontal transformation between two conceptual models of different modelling languages. Target of this transformation to switch between different approaches, while keeping the model content as far as possible. .................................................. 46

2.13 Computational-independent model (CIM), platform-independent model (PIM), platform description model (PDM) and platform-specific model (PSM); illustration of their dependencies and actors. .................................................. 52

2.14 Vertical transformation of a platform-independent model to a platform-specific model (taken from OMG - Object Management Group [2009]) .................................................. 53

2.15 OMG illustration of a model transformation using meta model mappings (taken from OMG - Object Management Group [2009]) .................................................. 54

2.16 MDA model levels according to OMG, describing a three level approach with a non-formally described modelling language (like INTERLIS or MADS) .................................................. 57

2.17 MDA model levels according to OMG, describing a four level approach with a formally described language (like UML) .................................................. 58

2.18 Simplified eMOF model depicted as UML Class Diagram (taken from OMG - Object Management Group [2006b]) .................................................. 59

2.19 ‘myClass’ of a fictive modelling language modelled with Ecore .................................................. 60

2.20 ‘myClass’ of a fictive modelling language with attributes and associations modelled with Ecore .................................................. 61

3.1 Adoption of figure 2.1 to depict the process steps between the universe of discourse and a logical or physical model. Step (1) illustrates the manual process of modelling with a helping model editor to create a conceptual model and step (3) shows the vertical transformation as a process with an according transformation tool. Process step (2) describes a horizontal transformation between different conceptual modelling approaches. .................................................. 64
3.2 Detailation of figure 2.12 to illustrate the work assumptions of this thesis: (1) Different spatial conceptual modelling languages can be described with a common meta modelling approach. (2) Based on the defined meta models, transformations between different conceptual schema languages are possible. (3) The language based tools can also be used in vertical transformations between conceptual and logical/physical models.  

3.3 INTERLIS 2 ‘Model’ Element; abstract syntax defined with Ecore  

3.4 INTERLIS 2 ‘Model’ and ‘Topic’ Elements, modelled with Ecore  

3.5 INTERLIS 2 basic structures of ‘Class’, ‘Structure’ and ‘Attribute’, modelled with Ecore  

3.6 INTERLIS 2 ‘Associations’ and ‘ReferenceAttributes’, modelled with Ecore  

3.7 INTERLIS 2 base ‘Domain’ elements, modelled with Ecore  

3.8 INTERLIS 2 ‘Units’, modelled with Ecore  

3.9 INTERLIS 2 ‘geospatial domains’, modelled with Ecore  

3.10 Humboldt Modelling Language elements ‘Model’ and ‘Package’ elements, modelled in Ecore  

3.11 ‘Classes’, ‘ComplexTypes’ and ‘Attributes’ of the HML meta model, expressed in Ecore  

3.12 HML meta model; different relationships modelled with Ecore  

3.13 Simple Types for the HML meta model; modelled with Ecore  

3.14 Union Type for the HML meta model; modelled with Ecore (to simplify the superclass ‘DefinitionElement’ of ‘Type’ is omitted)  

3.15 Spatial Types for HML; modelled with Ecore (to simplify, the following inheritances are omitted: From ‘Type’ to ‘GeometryType’; from ‘SimpleType’ to ‘NumericType’ and from the abstract ‘DefinitionElement’ to ‘Unit’, ‘Coordinate’, ‘CoordinateSystem’ and ‘Axis’)  

3.16 Standard predefined types for HML (Graphic created with the HUMBOLDT GeoModel Editor as described in chapter 4.2.5)  

3.17 Base Units defined in HML (Graphic created with the Humboldt GeoModel Editor as described in chapter 4.2.5)  

4.1 Simplified roles and products in an EMF software generation  

4.2 GMF components for the creation of a graphical domain-specific editor (white elements are project-specific components; grey components are standard framework components from EMF and GMF)  

4.3 Graphical representation of the model element ‘Package’  

4.4 Graphical representation of the model element ‘Class’
4.5 Graphical representation of the model element ‘ComplexType’. 118
4.6 Graphical representation of the model element ‘Enumeration’. 118
4.7 Graphical representation of simple types; ‘UnionType’, ‘StringType’, ‘NumericType’ and spatial types ‘PointType’, ‘LineType’, ‘PolygonType’ and ‘SurfaceType’. 119
4.8 Graphical representation of the model element ‘Unit’. 119
4.9 Graphical representation of the model element ‘Coordinate System’. 120
4.10 Graphical representation of the model element ‘Coordinate’. 120
4.11 Graphical representation of the model elements ‘Inheritance’, ‘Association’ and ‘Aggregation’. 121
4.12 Screen shot of the Eclipse Modelling Environment with the definition of the HML concrete syntax. 122
4.13 Screen shot of the ‘HUMBOLDT GeoModel Editor’ as standalone application. 124
4.14 Screen shot of a ‘Context Menu’ in the HUMBOLDT GeoModel Editor. 126
4.15 Screen shot of a ‘Shortcut’ element in the HUMBOLDT GeoModel Editor. 126
4.16 Screen shot of validation markers in HUMBOLDT GeoModel Editor. 127
4.17 Set-up for the transformation process. The Graphical Model Editor is used to define a conceptual model. The OpenArchitectureWare installation acts as mapping environment and transformation engine. The resulting UML model can be visualised with standard tools. 129

6.1 Language-oriented development based on meta modelling can be used at different stages of the geospatial modelling procedures. The usage of useful modelling languages leverages the initial efforts for the domain experts; according tools support the creation and ensure model quality; language transformations allow flexibility in the choices of modelling languages as well as offer support for vertical transformations to logical or physical models; model-to-text transformations give help in the creation of the final target products like documentation, feature catalogues or transfer formats. 156

6.2 Technical capabilities of tools and transformation can allow more variability in modelling approaches. On one hand a better support for domain experts with adapted languages for graphical modelling and on the other hand easy processable models for transformations to end products could. The ‘general purpose geospatial modelling approach’ of ISO/TC-211 might act as foundation or reference system. 162
A.1 Complete abstract syntax, developed for the selected INTERLIS 2 language subset. 164

B.1 Complete abstract syntax of the Humboldt Modelling Language (HML). 170

C.1 Complete abstract syntax, developed for the selected MADS language subset. 176
Listings

2.1 INTERLIS 2 definition of a simple class with an inlined domain definition ........................................... 37
2.2 INTERLIS 2 definition of a textual type (taken from [KOGIS, 2006b]) ......................................................... 37
2.3 INTERLIS 2 definition of three numeric types (taken from [KOGIS, 2006b]) ..................................................... 37
2.4 INTERLIS 2 definition of an enumeration with nesting and cyclic behaviour (taken from [KOGIS, 2006b]) ........... 38
2.5 INTERLIS 2 definition of a WGS 84 coordinate type (taken from [KOGIS, 2006b]) ........................................ 38
2.6 INTERLIS 2 definition of a line string type using the WGS84 Coordinate type (taken from [KOGIS, 2006b]) ........... 38
2.7 INTERLIS 2 definition of a polygon type with extra line attributes (taken from [KOGIS, 2006b]) ......................... 39
3.1 Pseudocode for the ‘Null’-transformation of HML Model ................................................................. 94
3.2 Pseudocode for the transformation of HML Packages to UML Model and UML Packages ................................ 94
3.3 Pseudocode for the transformation of HML Class to UML Class with «FeatureType» stereotype ....................... 95
3.4 Pseudocode for the transformation of HML ComplexType to UML Class with «DataType» stereotype .................. 96
3.5 Pseudocode for the transformation of HML Association, Aggregation and Composition to UML Associations with different association ends ................................................................. 96
3.6 Pseudocode for the transformation of HML StringTypes to UML String primitives ........................................... 97
3.7 Pseudocode for the transformation of HML numeric types to UML primitives or ISO/TC-211 MeasureType ................ 97
3.8 Pseudocode for the transformation of HML Enumerations to UML Classes with stereotype «Enumerations» ............ 99
3.9 Pseudocode for the transformation of HML UnionTypes to UML Classes stereotyped with «Union» ..................... 99
3.10 Pseudocode for the transformation of the HML spatial types ............................................................ 100
3.11 Pseudocode for the transformation of HML Model ................................................................. 101
LISTINGS

3.12 Pseudo code for the transformation of HML Packages to GML Schemas .................................................. 101
3.13 Pseudo code for the transformation of HML String to XML Simple Type .................................................. 102
3.14 Fragment of pseudo code for the transformation of HML Numeric Types to XML Simple Types or GML Measure Types . 102
3.15 Pseudo code for the transformation of HML Enumerations to XML Simple Types ........................................ 103
3.16 Pseudo code for the transformation of HML Union to XML Complex Types ............................................... 103
3.17 Pseudo code for the transformation of HML Class to XML Complex Type .................................................... 104
3.18 Pseudo code to resolve the type of HML Attributes .......................................................... 104
3.19 Pseudo code to resolve HML Relations .......................................................................................... 105
4.1 Two constraints defined with oaw ‘Check’ language ............................................................................ 122
4.2 Opening a model file of an Ecore based language (snippet of the oaw workflow file). The generic configuration structure consists of the reader class ‘oaw.emf.XmiReader’ with properties for filename and slot-name ........................................... 130
4.3 Snippet of oaw workflow file to validate the HML input model against the metamodel and an additional check file .............. 130
4.4 Snippet of oaw workflow file to execute the step-by-step transformation from a HML model to the UML model ............ 131
4.5 Snippet of the Xtend mapping for the creation of the UML model element and the packages ................................ 132
4.6 Snippet of the Xtend mapping for the creation of the UML classes from HML Class, HML ComplexType, HML UnionType and HML Enumeration ............................................................ 133
4.7 Snippet of the Xtend helper to retrieve a reference to an existing UML class for a given source element .................. 134
4.8 Snippet of the Xtend function to map attribute types to different UML types .............................................. 135
4.9 Snippet of the Xtend function to map attributes with geometry types .................................................... 136
4.10 Snippet of the Xtend code for applying stereotypes and tagged values from the ISO-19113 profile on packages and enumerations 136
4.11 Snippet of the oaw workflow to store the generated UML model in a file ..................................................... 137
4.12 Fragment of the Xtend mapping file for HML Packages .......... 139
4.13 Fragment of the Xtend mapping file for HML String Types to XSD Simple Types .................................... 140
4.14 Fragment of the Xtend mapping file for HML Numeric Types to XSD Complex Types .................................. 140
4.15 Fragment of the Xtend mapping file for HML Enumeration Types to XSD Simple Types. The Code covers the resolving of merged Enumeration Items and in the function ‘createEnumFacet’ the restriction of an XSD Simple Type. 141
4.16 Fragment of the Xtend mapping file for HML Classes. 142
4.17 Fragment of the Xtend mapping file for HML Attributes of HML Classes and HML Complex Types. 142
4.18 Fragment of the Xtend mapping file for HML Relations. 143
4.19 Snippet of the oaw workflow file for a HML to documentation transformation. 145
4.20 Snippet of a XPand template, which creates an empty html file for a HML model. 145
4.21 Snippet of a XPand template. The template functions create the documentation for a HML package with all HML classes as tables. 146

A.1 Abstract syntax for the selected subset of INTERLIS 2 as Ecore Model, listed in EMFATIC notation (see Eclipse Foundation [2010]). 165

B.1 Abstract syntax for the developed Humboldt Modelling Language (HML) as Ecore Model, listed in EMFATIC notation (see Eclipse Foundation [2010]). 171

C.1 Abstract syntax for the selected subset of MADS as Ecore Model, listed in EMFATIC notation (see Eclipse Foundation [2010]). 177
Bibliography


H. A. Proper. Data schema design as a schema evolution process. Data \\


Curriculum Vitae

Jan Schulze Althoff
Diplom Forstwirt

ETH Zurich
Institute of Cartography
Wolfgang-Pauli-Strasse 15
8093 Zurich
Switzerland

Contact   schulzealtho@yahoo.de
Date of Birth   25.05.1971
Place of Birth   Neuss, Germany

Education
03/2007–08/2011   Doctoral student at Institute for Cartography at ETH Zurich
07/2009–06/2010   Grantholder at Spatial Data Infrastructure Unit, EC Joint Research Center, Ispra (Italy)
10/1992–02/1998   Studies of Forestry at the University of Freiburg, Germany

Professional Experience
06/2005–03/2007   Software Developer and Consultant at Geotask AG, Basel
04/2004–06/2005   Independent Consultant and Developer for Geospatial Application with projects for municipality of Vienna and University of Natural Resources and Life Sciences, Vienna
12/2000–09/2002 Software Developer and Consultant at Trivadis AG, Zurich, Freiburg

02/1998–11/2000 Researcher in the project 'AGRO' and 'Tree Resources Outside the Forests' at Department of Remote Sensing and Landscape Information Systems 'FeLis', University of Freiburg, Germany
Acknowledgements

This work would not have been possible without the support of many people I met and worked together during the last years. My work at the European HUMBOLDT project (EC contract SIP5-CT-2006-030962) and the grant from the Joint Research Centre gave me the opportunity to concentrate on issues related to spatial data infrastructures and conceptual modelling. Among all those persons, I would especially thank:

Dr. Christine Giger for that fantastic working opportunity with her strong support and practical guidance;
Prof. Dr. Lorenz Hurni and Prof. Dr. Barbara Koch for the examination work, lots of useful hints and fruitful colloquia;
Hans-Ruedi Gnägi and Marian DeFries for several nice discussions and many interesting thoughts;
And finally a warm thank you to the friendly colleagues of the Institute of Cartography, especially Corina Pachlatko for guiding me through the pitfalls of administrations.