Methodical spatial database design with topological polygon structures

Mazimpaka Jean Damascène
March, 2009
Methodical spatial database design with topological polygon structures

by

Mazimpaka Jean Damascène

Thesis submitted to the International Institute for Geo-information Science and Earth Observation in partial fulfilment of the requirements for the degree in Master of Science in Geoinformatics.

Degree Assessment Board

Thesis advisor       Dr. J. Morales
                      Dr. Ir. R.A. de By
Thesis examiners     Prof. Dr. M.J. Kraak
                      Dr. A. Wombacher

INTERNATIONAL INSTITUTE FOR GEO-INFORMATION SCIENCE AND EARTH OBSERVATION
ENSCHEDE, THE NETHERLANDS
Disclaimer

This document describes work undertaken as part of a programme of study at the International Institute for Geo-information Science and Earth Observation (ITC). All views and opinions expressed therein remain the sole responsibility of the author, and do not necessarily represent those of the institute.
Abstract

With the current advances in Information and Communication Technology (ICT), we see a problem shift from data availability to data maintenance. Today’s best approach to managing huge amounts of data is to use a Database Management System (DBMS). The field of Geographic Information Systems (GIS) makes no exception. However, in GIS the problem becomes much more complex due to the nature of spatial data. For spatial data, a mechanism is needed to capture the semantics of spatial objects, and an improved methodology for spatial database design is needed to simplify the design complicated by the semantics of spatial objects and an increasing number of standards and implementation platforms.

The main objective in this research is to develop primitives for modelling collections of area features that display topological dependencies, and to develop a transformational database design process for the realisation of area feature topology in spatial databases. In this respect, modelling primitives have been developed by extending the Unified Modelling Language (UML). A semi-automatic transformation process has been developed with two transformation steps. The first step uses transformation definitions that have been developed using the programming environment of Enterprise Architect. The second step uses a custom tool that has been developed using Java programming language.

The modelling primitives are used in a modelling tool such as Enterprise Architect to build a conceptual model of the database. The transformation process transforms design results between schema levels. Firstly, the conceptual schema is transformed to logical schema for an implementation of choice between spaghetti and topological models. Secondly, the logical schema is transformed to physical schema in which the final result is a text file containing SQL statements for implementing the database in PostgreSQL/PostGIS Database Management System.

Keywords

Spatial database, area features, topology, UML, modelling primitives, MDA, design, XMI
Acknowledgements

I take this opportunity to thank the Almighty God for his blessings to me on each day of my life.

I thank my first supervisor Dr. Javier Morales. Your guidance showed me the way in the world of scientific research. I extend my gratitude to my second supervisor Dr. Rolf de By. Your patience when I could not easily understand your comments encouraged me. You have inspired me to build my concrete ideas from abstract concepts.

I express my gratitude to CGIS-NUR and the Netherlands Government though NUFFIC funding for their financial support.

My thanks also go to Karangwa Charles. I can not forget your contribution in making it possible for me to pursue my studies, and the orientation you gave me in my studies.

Finally, I am so grateful to my friends at ITC, especially my GFM2 classmates for their cooperation and encouragement, and to all other people who have supported me directly or indirectly throughout my studies.

May God bless you all.
Contents

Abstract i

Acknowledgements iii

List of Figures vii

List of Tables ix

List of abbreviations xi

1 Introduction 1
  1.1 Motivation and problem statement .......................... 1
  1.2 Research identification .................................... 2
    1.2.1 Research objectives .................................. 2
    1.2.2 Research questions .................................... 3
    1.2.3 Innovation aimed at .................................. 3
    1.2.4 Related work .......................................... 3
  1.3 Method adopted ........................................... 4
  1.4 Structure of the thesis .................................... 5

2 Model Driven Architecture 7
  2.1 Introduction ................................................ 7
  2.2 The basic concepts ......................................... 8
  2.3 Standards supporting MDA ................................ 12

3 Spatial data and Feature topology 15
  3.1 Spatial data ................................................ 15
    3.1.1 Spatial data models .................................. 15
    3.1.2 Standards related to spatial data ...................... 16
  3.2 Feature topology .......................................... 21
    3.2.1 Overview of the feature topology ..................... 21
    3.2.2 Topology realisation options ........................ 23
    3.2.3 Feature topology support in Database Management Sys-
        tems ...................................................... 25
## List of Figures

1.1 The method adopted .................................. 5  
2.1 Model transformation .................................... 10  
2.2 UML extensibility mechanisms .............................. 12  
3.1 Conceptual modelling .................................... 18  
3.2 Geometry basic classes with specialization relations ...... 19  
3.3 Topology class diagram .................................... 20  
3.4 Geometry class hierarchy in the Simple Feature Access .... 20  
3.5 Topological relations between regions with connected boundary 22  
4.1 Example of area collections ................................. 30  
4.2 The transformational database design process ............... 49  
5.1 Area collection modelling primitives ........................ 53  
5.2 Enterprise Architect’s DDL standard transformation definition 55  
5.3 Hierarchy of transformation templates ........................ 56  
5.4 Code fragment that transforms the \textit{GM\_Point} data type .... 56  
5.5 Example of a \textit{ContiguousPolygonalArea} class .......... 57  
5.6 Result of transformation using \textit{PostgreSQL\_Spaghetti} ...... 57  
5.7 Result of transformation using \textit{PostgreSQL\_topological} ...... 58  
5.8 Example of an XMI file in XMI 1.1 format ................... 59  
5.9 A code fragment that invokes an XML parser ................. 61  
5.10 A code fragment that deletes nodes from a DOM tree ...... 61  
5.11 The GUI of the Logical to Physical schema transformation tool 63  
5.12 The conceptual schema of the test database ................. 65  
5.13 The logical schema of the test database in Spaghetti model ...... 65  
5.14 The logical schema of the test database in topological model ... 66  
5.15 Visualisation of the test data stored in spaghetti model ...... 67  
5.16 Visualisation of the test data stored in topological model ...... 67
List of Tables

A.1 Non spatial data types correspondence 75
A.2 Spatial data types correspondence 76
A.3 A comparison of DOM and SAX-based parsers 77
List of abbreviations

API  Application Programming Interface
CIM  Computation Independent Model
CORBA  Common Object Request Broker Architecture
DBMS  Database Management System
DDL  Data Definition Language
DML  Data Manipulation Language
DOM  Document Object Model
DTD  Document Type Definition
GEOS  Geometry Engine - Open Source
GIS  Geographic Information System
HTML  HyperText Markup Language
IEC  International Electrotechnical Commission
ISO  International Standards Organization
JDK  Java Development Kit
JTS  Java Topology Suite
MDA  Model Driven Architecture
MOF  Meta Object Facility
OCL  Object Constraint Language
OGC  Open Geospatial Consortium
OLE/COM  Object Linking and Embedding/Component Object Model
OMG  Object Management Group
PIM  Platform Independent Model
PSM  Platform Specific Model
### List of abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SAX</strong></td>
<td>Simple API for XML</td>
</tr>
<tr>
<td><strong>SFA</strong></td>
<td>Simple Feature Access</td>
</tr>
<tr>
<td><strong>SQL</strong></td>
<td>Structured Query Language</td>
</tr>
<tr>
<td><strong>UML</strong></td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td><strong>XMI</strong></td>
<td>XML Metadata Interchange</td>
</tr>
<tr>
<td><strong>XML</strong></td>
<td>eXtensible Markup Language</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation and problem statement

A database is a very large, structured and integrated collection of data. A Database Management System (DBMS) is a software system that allows creating and managing databases. A DBMS has many advantages such as providing a unified storage, management and control for large volume of data and providing a standard mechanism for accessing data.

A spatial database is a special type of database which, in addition to storing the common attributes values of the objects, stores data about their geographical location and shape and provides functions to work on these data. The representations of spatial objects in the database have specific characteristics which make their management and manipulation complex. Among such characteristics are the topological spatial relations, which are spatial relations that are not affected by transformations like rotation and scaling [ECF94].

While working with data in a spatial database to display, process or analyse spatial information, users are often interested in objects that have a certain spatial relationship between them. The spatial relations are categorized into topological relations, metric relations and relations concerning the partial and total order of spatial objects [EF91]. Furthermore, topological spatial relations can be area feature topological relations, linear feature topological relations, or point feature topological relations depending on the type of the real world object represented. Topological relations are involved in several problems ranging from simple to advanced cases. A simple case is like, instead of being interested in all hospitals of a country, people may be interested in only hospitals built in one particular province. A much more advanced case is like the requirement to enforce topological dependencies between features in a collection, such as university campuses should not overlap, or there can not be gaps between the provinces of a country.

The support for the feature topology in open source spatial database platforms is only elementary. As an example, in PostgreSQL/PostGIS, it is possible to check if the representations of two spatial objects have a certain topological spa-
1.2 Research identification

This section describes the objectives of the research, the research questions to be answered, the innovation aimed at and an overview of related work.

1.2.1 Research objectives

Overall objective

The objective of this research is to develop area feature topology modelling primitives, a number of functions associated with them and a semi-automatic design process for the realisation of area feature topology in spatial databases.

Specific objectives

The overall objective is split into the following specific objectives:

1. To identify realization options of the feature topology

2. To develop area feature topology modelling primitives by extending UML¹

3. To define a set of transformation rules between the conceptual and logical schema levels that carry topological definitions

4. To develop a transformation process between the conceptual, logical and physical schema levels that handles topological information

¹. UML: Unified Modelling Language
Chapter 1. Introduction

5. To provide two alternatives for the realization of area feature topology
6. To verify and validate the area feature topology modelling primitives, associated functions and the transformation process

1.2.2 Research questions

To achieve the objectives, the following research questions need to be answered:

1. What are the realization options of feature topology?
2. How to extend UML for developing area feature topology modelling primitives?
3. How to develop a transformational database design process for the realization of area feature topology?
4. How to verify and validate the area feature topology modelling primitives, associated functions and the transformation process developed?

1.2.3 Innovation aimed at

The innovation aimed at is a small collection of area feature topology modelling primitives, a set of maintenance functions associated with them and a semi-automatic transformation process for the realization of area feature topology.

1.2.4 Related work

This research focuses on two main related parts: methodical spatial database design and feature topology implementation. In general, the available work focuses on one of the two parts. In a work on topology implementation, Hoel et al. [HMM03] present a physical implementation of topology in which topology relationships are not explicitly stored, but rather they are derived during a topological validation done on demand. Theobald [The01] reviews spatial data structures with respect to topology implementation and presents the Cartographic Data Structure (CDS) in which topological relations are not explicitly stored but rather they are computed on-the-fly based on a consistent clockwise order of vertices in the data structure. Our alternative implementation differs from that work by providing two topology implementation options, each with topology validation at the time of operations to ensure that users can never see the topology in inconsistent state.

In a work on methodical spatial database design, Bennekom [vBM08] developed a prototype for performing a transformational design of the spatial database of the Land Administration Domain Model (LADM) Survey Package following MDA \(^2\) principles. In his work, only simple topological spatial relations are considered and they are represented as OCL \(^3\) constraints. OCL has a high expressive power, but the textual description of OCL constraints makes the model less

---

2. MDA: Model Driven Architecture
3. OCL: Object Constraint Language
understandable by the model readers, and writing OCL constraints requires the database designer to master OCL syntax. It is desirable to restrict the use of OCL constraints to the cases where they are the only possible option. Moreover, we need a general transformation process applicable to different domains. Wang et al. [WR07] present a method to extend geographic data modelling also taking into account topological spatial relations. This method uses a Constraint Decision Table (CDT) to store the topological integrity constraints. The problems of this approach are linked to the separation of the constraints from the UML conceptual schema and the mastery required for defining the constraints. Our method differs from that work by keeping topological spatial relations as an integral part of the UML conceptual schema and providing the database designer with a more user friendly way of representing these relations; which is a small collection of topological modelling constructs for which the transformation towards implementation is clear. A more general difference of this project from other related work is that it treats topological dependencies between features within a collection of features rather than topological relations between simple features treated by the related work.

1.3 Method adopted

The method that will be adopted in this project is based on the core workflows of the Unified Software Development Process which is described in [JBR99]. This method comprises five core workflows: requirements definition, analysis, design, implementation, and testing. As the project starts by exploring the available possibilities, then developing a transformation process and a tool used in that transformation and ending with testing, these activities fit well in those workflows as follows:

1. Requirements definition and analysis
   In this phase a literature review will be done to explain fundamental concepts, and to identify realisation options for the feature topology.

2. Design
   The activities in this phase are:
   - to propose a number of topological classes for modelling area features: describe each class, a primitive for modelling it and a set of maintenance functions to work on it,
   - to describe the transformational design of spatial database according to MDA principles by:
     - describing the conceptual, logical, and physical schema,
     - describing transformation rules, that apply also to the proposed primitives, for mapping between schema levels.

3. Implementation
   In this phase, programming will be done for:
Chapter 1. Introduction

- the creation of the area feature topology modelling primitives and associated maintenance functions,
- the implementation of the transformational database design process following MDA principles, and this for two of the feature topology realisation options identified in the first phase.

4. Testing
   In this phase, a test case will be prepared with real world data that can be modelled as collections of area features. The test case will be used to test the topology modelling primitives, associated functions and the transformation process.

In parallel to all these activities the documentation task will be done. The method that will be followed in the research is illustrated in Figure 1.1 below, showing the phases and deliverables from each phase which serve as input to the next phase.

![Figure 1.1: The method adopted (adapted from [AN05], p.36)](image)

1.4 Structure of the thesis

Based on the method adopted in the research, this thesis is organized in six chapters as follows:

**Chapter 1** describes the motivation to do this research, states the problem, research objectives and questions, and describes the method adopted in the research.

**Chapter 2** discusses the Model Driven Architecture (MDA), which is the technique that the transformational design process developed in the research follows. The chapter discusses MDA by presenting underlying basic concepts and some standards that support MDA.

**Chapter 3** discusses the concepts of spatial data and feature topology on which the transformational design will be applied. Some standards related to
1.4. Structure of the thesis

Spatial data are briefly presented and identified feature topology realisation options are presented.

Chapter 4 contains the design part of the project. The chapter describes two design parts: the design of primitives for modelling collections of area features and the design of a transformation process.

Chapter 5 contains the implementation and testing parts of the project. Both the creation of the modelling primitives and the implementation of the transformation process are presented. The testing that has been done to verify and validate the developed modelling primitives and transformation process is also presented.

Chapter 6 presents and discusses the results of the project execution. The chapter also presents the conclusions drawn and the recommendations made for future improvement of the modelling primitives and the transformation process.
Chapter 2

Model Driven Architecture

This research intends to provide an improved method for spatial database design. The method should be replicable to contribute to complexity management through a separation of concerns. That is, the method should separate the concerns at the conceptual level from the concerns at the implementation level, thus allowing the artefacts at the conceptual level to be reused for different targeted implementations. Among several available methods of system development, the Model Driven Architecture achieves this separation of concerns. At the conceptual level, the concepts are modelled with focus on the data content without caring about the implementation issues. After a short introduction that defines MDA, this chapter presents some concepts (Section 2.2) and standards (Section 2.3) which are at the core of MDA.

2.1 Introduction

The Model Driven Architecture (MDA) has been introduced by the Object Management Group (OMG). MDA is an approach to software development using models. In the MDA framework, a software system is produced through a series of model transformations aided by a modelling tool. As stated in [MM03], “The Model-Driven Architecture starts with the well-known and long established idea of separating the specification of the operation of a system from the details of the way that system uses the capabilities of its platform. MDA provides an approach for, and enables tools to be provided for:

- specifying a system independently of the platform that supports it,
- specifying platforms,
- choosing a particular platform for the system, and
- transforming the system specification into one for a particular platform

The three primary goals of MDA are portability, interoperability and reusability through architectural separation of concerns.”
2.2 The basic concepts

The MDA guide [MM03] defines the concepts that are fundamental to MDA. The definitions of some of those concepts are presented in this section.

System

A system is a collection of components that together perform a certain function. The system may include software, hardware and people. The discussion in this document focuses on the software component within the system.

Model

A model of a system is a representation of the system that describes it and its environment for a certain purpose. A model of a system is sometimes referred to as a view of the system. A model is often presented as a combination of drawings and text.

Metamodel

A metamodel is an abstraction that describes a model. From data to metamodel, there is an increasing abstraction; where the data are raw facts, a model is an abstraction of data, metadata being information about a model and metamodel being an abstraction of metadata [BP98].

Model-Driven

MDA is an approach to system development, which is based on models. It is model-driven because it specifies how to use models to ease the understanding, design, construction, deployment, operation and maintenance of a system.

Architecture

As stated in [MM03], “the architecture of a system is a specification of the parts and connectors of the system and the rules for the interactions of the parts using the connectors.” The Model-Driven Architecture specifies certain kinds of models to be used, how those models may be made ready for use, and the relationships of the different kinds of models.

Platform

A platform is a set of components and technologies that provide to any supported application a set of functionalities through interfaces while they hide to the application the implementation of those functionalities.
**Viewpoint**

OMG defines a viewpoint on a system as a level of suppressing selected detail when specifying that system, in order to focus on particular concerns within that system [MM03]. This suppression of detail is achieved using a set of concepts and rules. The concepts and rules may be considered to form a viewpoint language. MDA specifies three viewpoints on a system; a computation independent viewpoint, a platform independent viewpoint and a platform specific viewpoint.

**MDA Viewpoints and models**

**Computation Independent Viewpoint and Model**

The Computation Independent Viewpoint (CIV) focuses on the system requirements and its environment. In this viewpoint, the details of the structure of the system and the processing carried out by the system are hidden or not yet determined. The Computation Independent Model (CIM) of a system is its view from the computation independent viewpoint. A CIM does not show details of the structure of a system. A terminology that is familiar to the people in the domain in question is used in the specification of a CIM. For this, a CIM is sometimes called a domain model.

**Platform Independent Viewpoint and Model**

The Platform Independent Viewpoint (PIV) focuses on the operation of a system while hiding the details imposed by a particular platform. The Platform Independent Model (PIM) of a system is its view from the platform independent viewpoint.

**Platform Specific Viewpoint and Model**

The Platform Specific Viewpoint (PSV) considers the concepts available at the platform independent viewpoint but tailored with the detail imposed by a specific platform. The Platform Specific Model (PSM) of a system is the view of that system from a platform specific viewpoint.

**Model Transformation**

Model transformation is the process of converting one model to another model of the same system. In the example shown in Figure 2.1, a platform independent model and other information are combined by the transformation to produce a platform specific model.

Model transformations form a key part of MDA. The model transformation comprises two steps: mapping and the actual transformation. An MDA mapping is the process of defining or establishing relationships between concepts in the source model and concepts in the target model. After the mapping, the actual transformation is carried out to produce, from the PIM, a PSM for a particular platform. Based on [MM03], some types of mappings are explained in the following.
### 2.2. The basic concepts

**A Model Type Mapping**

The language in which the PIM is expressed is referred to as the PIM language. Similarly, the language in which the PSM is expressed is referred to as the PSM language. A model type mapping establishes a relationship between model elements based on the element types available in the PIM language and the element types available in the PSM language. This mapping may also specify transformation rules in terms of the instance values to be found in models expressed in the PIM language.

**A Model Instance Mapping**

This approach involves defining marks. A mark represents a concept in the PSM, and is applied to an element of the PIM, to indicate how that element is to be transformed. Since marks are platform specific, they are not part of the PIM. A PIM is marked for use on a particular platform, and the marked PIM will be transformed into a PSM for that platform.

**A combination of Type and Instance Mappings**

A combination of the previous two approaches allows overcoming the deficiencies of either. A model type mapping is only capable of expressing transformations in terms of rules about things of one type in the PIM resulting in the generation of some thing(s) of some (one or more) type(s) in the PSM. However information other than correspondence between types in the PIM and PSM is useful for mapping a complex model; and this information can come from marks. Similarly, because each type of model element in the PIM is only suitable for certain marks, the information on the type correspondence is useful.

**Marking Models**

Marks may also specify quality of service requirements on the implementation. In this case, instead of indicating the target of a transformation, a
mark may simply provide a requirement on the target. The transformation will then choose a target suitable for that requirement. Marks may need to be structured or constrained to be properly used. Also, instead of being supplied by a mapping, a set of marks may be specified by a mark model, which is independent of any particular mapping.

- **Templates**
  As defined in [MM03], “templates are parameterized models that specify particular kinds of transformations.” A mapping may include templates. Templates can be used in rules in model type mapping for transforming a pattern of model elements into another pattern of model elements. A set of marks can be associated with a template to indicate instances in a model which should be transformed according to that template.

- **Mapping Language**
  A mapping is specified using some language to describe a transformation of one model to another. The description may be in natural language, an algorithm or in a model mapping language (a language designed for specifying model mappings).

### Degrees and Methods of Model Transformation

The methods of model transformation vary depending on the degree of automation (manual, semi-automatic and automatic) and the approach of putting into a model the information required for the transformation. Three different transformation approaches are explained in the following

- **Manual Transformation**
  In order to make the transformation from PIM to PSM, design decisions must be made. These design decisions are made with reference to engineering requirements on the implementation. The MDA approach adds value by making the explicit distinction between a PIM and the transformed PSM, and by recording the transformation.

- **Semi-automatic Transformation**
  There is a tool to perform the transformation of the prepared model. However the user is given the possibility to interact with the transformation process and influence it for instance by selecting among the available transformation options or inputting some parameters.

- **Automatic Transformation**
  In situation where a PIM provides all the information needed for implementation and necessary architectural decisions are built in the transformation tool, there is no need to add marks or use data from additional profiles. The tool transforms the model directly to a program code. We argue that the automatic transformation is neither always feasible nor the best option because it is not easy to include into the transformation tool all the information needed for implementation. Moreover this approach
2.3 Standards supporting MDA

OMG has adopted a number of standards, that together contribute to achieving the primary goals of MDA of portability, interoperability and reusability. Some of these standards allow modelling the system conceptually without caring about the implementation platform hence achieving portability, others allow packaging the model components for use in other models and/or on different software tools hence achieving also interoperability and reusability. Among these standards there are UML, MOF and XMI.

UML

The Unified Modelling Language (UML) is a standard modelling language that uses object-oriented concepts for visualizing, specifying, and documenting components of software systems. Models used with MDA can be expressed using the UML language [MM03].

UML is extensible. UML extensibility mechanisms are stereotypes, tagged values and constraints. A stereotype is a variation of a model element with the same form as that element but with different meaning and usage. Stereotypes allow introducing new modelling elements based on existing ones. Tagged values are a mechanism that allows adding new properties to modelling elements. A constraint is an expression that specifies some condition or rule that must always hold on one or more modelling elements [JBR98]. Stereotypes, constraints and tagged values can be put together to make a reusable collection for a specific purpose. This collection is called a UML profile. An example illustrating UML extensibility mechanisms is given in Figure 2.2. In this figure, the stereotype ADT (abstract data type) has been used to introduce a new modelling element Queue based on the existing element Class. Tagged value and constraint, that can allow a specific treatment of the model element, are also shown.

![Figure 2.2: UML extensibility mechanisms](image)

UML has been increasingly adopted as the modelling language of choice for its
features of extensibility, and object-oriented concepts available in many program- 
ing languages. The extensibility of UML makes it the best choice in so- 
plicated applications such as Computer-Aided Design (CAD) and Geographic 
Information System (GIS) which deal with complex objects with complex rela- 
tionships [MVC03]. Since this research treats geographic information, it uses 
UML for the same reasons.

OCL

The Object Constraint Language (OCL) is an extension to the UML, used to 
express constraints in UML models. UML models are generally created by us-
ing UML diagrams, but there are conditions that can not be expressed in such 
models. The combination of UML and OCL allows developing complete mod- 
els [WK03]. OCL is widely used to express the constraints in aforementioned 
UML profiles.

MOF

The Meta-Object Facility (MOF) provides a specification for implementation of 
model repositories. As stated in [Obj05], “MOF specifies an abstract language 
for specifying, constructing, and managing technology neutral metamodels: A 
metamodel is in effect an abstract language for some kind of metadata.” To 
enable model transformations, MDA requires models to be expressed in a MOF-
based language. Models expressed in a MOF-based language can be stored in 
a MOF-compliant repository, parsed and transformed by MOF-compliant tools, 
and rendered into XMI for transport over a network. UML is one example of a 
MOF-based modelling language.

XMI

XML Metadata Interchange (XMI) is an OMG standard that facilitates the ex-
change of UML models in the form of XML files. It specifies how a UML model 
is converted to XML file format and the XML file back to UML for model ex-
change. Multiple tools can interchange the document containing model data 
despite the presence of tool-specific information. XMI standard uses extensions 
to hold such tool-specific information [Obj07].
2.3. Standards supporting MDA
Chapter 3

Spatial data and Feature topology

The Model Driven Architecture (MDA) approach described in chapter 2 will work on the concepts of spatial data and feature topology at both the conceptual and the implementation levels. This chapter describes these concepts for a better understanding of subsequent chapters. The concepts of spatial data are described in Section 3.1 while the concepts of feature topology are described in Section 3.2.

3.1 Spatial data

Geographic data (also called spatial data) are descriptions of geographic phenomena including non-spatial properties such as name, and spatial properties including location and shape. The consideration of spatial properties has led to the development of concepts and standards which are different from those of non-spatial data. This section reviews the concepts of spatial data, spatial data models and related standards that are relevant for this thesis.

3.1.1 Spatial data models

A data model is a collection of concepts for describing data. It describes the type of data and the operations that can be performed on them. Data models can be characterized as relational, object-oriented, or object-relational. The spatial component of geographic data may include both geometry (location and shape) and topology (a kind of spatial relationship between represented objects). The representation of the geometry and topology has led to new data models; the spatial data models. These data models are in two categories: entity-based models and field-based models [RSV02]

Entity-based models

In the entity-based models, also referred to as object or feature-based models, the interest is on discrete geographic objects, also called entities or features. Geographic objects are classified into the following types:
3.1. Spatial data

- Zero-dimensional objects or points (e.g., cities)
- One-dimensional objects or linear objects (e.g., roads)
- Two-dimensional objects or area features (e.g., countries)

The object in any type may be simple or complex and the choice of the type highly depends on the application considered. The same spatial object may have two different types in two different applications.

Field-based models

In a field-based (or space-based) model, each point in the space is associated with an attribute value, that is, the description of the geographic phenomenon is defined by a continuous function in the space. An example is the temperature on the earth’s surface.

The spatial data models that have been presented in this section define a representation of spatial objects at an abstract level. After the data modelling, a choice will be made to what mode to use for a logical and physical representation. The representation mode will be either the vector mode where the geographic object is represented using a finite set of points, or the tessellation (or raster) mode where the spatial object is represented as a set of non-overlapping cells. This research project will treat the entity-based data models and work only on the vector representation of features.

3.1.2 Standards related to spatial data

The International Organisation for Standardisation (ISO) and the Open Geospatial Consortium (OGC) are two organisations involved in defining standards for geographic information. ISO is a global organisation in charge of preparing international standards. OGC provides publicly available Geographic information system specifications that it produces in a consensus-based process among its members [Ope05]. The two organisations operate independently and produce different kinds of standards but they also collaborate on items of common interest. This collaboration is seen for instance in the adoption, by one organisation, of some standards produced by the other.

ISO has a specialised standard technical committee called ISO/TC 211 which is dedicated for Geographic Information and Geomatics [OS02]. The ISO 19100 series of standards are produced by ISO/TC211 in the aim of establishing a set of standards for information concerning objects that have a location relative to the Earth [Ope05]. These standards are mainly aimed at achieving interoperability between software components that process spatial data. This section provides an overview of some standards related to spatial data, which are relevant for this thesis. A list of standards and further discussion of the ISO 19100 series can be found in [Ope05].
OGC defines two types of specifications or standards: *Abstract specification* and *implementation specification*. Abstract Specifications provide the conceptual foundation for many OGC specification development activities. They are platform and programming language independent; they provide a reference model for the development of Implementation Specifications. Implementation Specifications are based on Abstract Specifications and are specific to a certain target computing environment. They are targeted to a technical audience, that is, to software developers [Ope05].

### ISO 190101

The ISO 19101 provides the Reference Model of the ISO 19100 series. ISO 19101 defines a framework for standardization in the field of geographic information, establishes the basic principles and describes how the contents of the different standards are related [Ope05]. The standard defines the abstraction of the real world, which means the creation of an abstract description of some part of the real world. Such abstract description, that the standard calls a conceptual model, is presented in a conceptual schema using a conceptual schema language [Ope05]. An example for a conceptual model would be the set of features that are important for a cadastral management application. UML has been approved to be a conceptual schema language to be used in the ISO 19100 series. The rules for a conceptual application schema are specified in ISO 19109 (Rules for application schema) and the rules for using the conceptual schema language are specified in the ISO 19103 (Conceptual schema language). Figure 3.1 illustrates the conceptual modelling from real world to conceptual schema.

### ISO 190107

The ISO 19107 specifies conceptual schemas for describing spatial characteristics (geometry and topology) of 2-dimensional and 3-dimensional geographic features, and a set of related operations such as *buffer*. It treats geometry and topology of vector data only [ISO03]. The conceptual schemas specified by the ISO 19107 comprise two sets of packages: the *Geometry packages* and the *Topology packages*. The Geometry packages contain various classes for representing geographic objects, whereas the Topology packages contain classes that allow associating geographic objects with the information about topological relations.

The geometry packages distinguish between *primitive* geometric objects, *aggregates* and *complexes*. The basic classes defined in the geometry packages are shown in Figure 3.2. The root class *GM_Object* is specialised into *GM_Primitive*, *GM_Complex* and *GM_Aggregate* classes. As defined by ISO in [ISO03], “a geometric primitive (*GM_Primitive*) is a geometric object that is not decomposed further into other primitives in the system. A geometric complex (*GM_Complex*) is a set of primitive geometric objects (in a common coordinate system) whose interiors are disjoint”. A composite (*GM_Composite*) is a special case of geometric complex in which the primitives are not completely disjoint but share
3.1. Spatial data

Figure 3.1: Conceptual modelling (adapted from [Ope05])

common boundaries. A composite is a complex that could be represented as one primitive. Composites are intended for use in datasets in which the underlying geometry has been decomposed, usually to show its topological nature. Objects of type GM_Primitive will be open, that is, they will not contain their boundary points whereas objects of type GM_Complex will be closed. The aggregates (GM_Aggregate) collect geometric objects. The geometries collected into a single aggregate geometry can intersect, overlap or even be equal to each other. This is the main difference between an aggregate and a complex.

The basic classes in the topology packages are shown in Figure 3.3. The root class of the diagram is TP_Object, with subclasses TP_Primitive, and TP_Complex, which are related in a way similar to the way GM_Primitive and GM_Complex are related. The major difference, as stated in [ISO03], is that “a GM_Primitive is more loosely coupled to a GM_Complex, allowing it to stand alone, whereas a TP_Primitive must be in at least one TP_Complex”. A topological complex consists of collections of topological primitives of all kinds up to the dimension of the complex; a TP_Complex is an organized structure of TP_Primitives.

The ISO 19107 standard is the OGC’s Feature Geometry Abstract Specification (FGAS) approved by ISO. OGC has developed a simple implementation specification of the FGAS, which focuses on Distributed Computing Platforms (DCP)
Chapter 3. Spatial data and Feature topology

Figure 3.2: Geometry basic classes with specialization relations. (Source: [ISO03])

and specifies a simplified data model based on points, lines and polygons in two Euclidean dimensions. This simplified specification was called the Simple Feature Implementation Specification (SFS). The SFS is available for three Distributed Computing Platforms: SQL, CORBA and OLE/COM.

OpenGIS Implementation Specification for Geographic information - Simple feature access

The Simple Feature Implementation Specification which provides a well-defined and common way for applications to store and access feature data in relational or object-relational databases comprises two parts: Part 1 - Common architecture and Part 2 - SQL option. The first part provides the common feature model, and the second part provides a standard specification for the implementation of the abstract model, defined by the first part, on the SQL Distributed Computing Platform. The diagram in Figure 3.4 shows the features hierarchy as specified by the Simple Feature Access (SFA) Part 1.

In this thesis, the Platform Independent Model (PIM), to be transformed fol-
3.1. Spatial data

Figure 3.3: Topology class diagram. (Source: [ISO03])

Figure 3.4: Geometry class hierarchy in the Simple Feature Access. (Source: [Ope06])
Following MDA technique, will be built using the spatial data concepts specified by ISO 19107 because the latter is platform independent. The target Platform Specific Model (PSM) will be built following the Simple Feature implementation Specification for SQL because the target platform (PostgreSQL/PostGIS) is an SQL database management system.

3.2 Feature topology

The feature topology describes the structure of space mainly through neighbourhood relationships between different topological elements that make the features. Throughout this thesis, the feature topology will be dealt with. To lay the foundation for a better understanding of subsequent chapters, this section provides an overview of feature topology and its current support in Database Management Systems. Also this section describes different options of realizing the feature topology.

3.2.1 Overview of the feature topology

When one is interested in single objects, he/she needs to consider the data model and the representation mode only. Unlikewise, when geographic objects are considered together, they present spatial relationships which are of interest as well. Topological relationships are the most important of these relationships. Topology studies topological relations of features. Topological relations have been a subject of intense research (Max [Ege89], Max et al. [EF91], Max et al. [ECF94], Markus et al. [SB06], etc) that has produced a formal definition of them and the models for describing and computing them. The primary models that resulted from the research are the 4-Intersection Model, the 9-Intersection Model and their dimensionally-extended versions.

The 4-Intersection Model (4-IM)

The 4-Intersection model is based on the distinction between the interior \((\text{int} A)\) and the boundary \((\partial A)\) of a region \(A\). All possible intersections of the interiors and the boundaries of two spatial objects \(A\) and \(B\) form a 4-Intersection Matrix. By considering whether the value is empty or non-empty, a number of topological relations can be distinguished. For regions with connected boundaries, eight of those relations can be realized and have led to eight topological predicates (Boolean functions that are used to test the topological relationships between two spatial objects): disjoint, meet, equal, inside, contains, covers, coveredBy and overlap. These relations are mutually exclusive.

The 9-Intersection Model (9-IM)

The 4-intersection matrix makes it possible to identify topological relations between simple spatial objects, that is, spatial objects with connected boundaries [ECF94]. To make it possible to identify relations between regions with
holes, the 9-Intersection Model has been introduced as an extension to the 4-Intersection model by considering also the exterior (A−) of region A [SB06]. The eight meaningful topological relations between two regions A and B, based on the 9-intersection model, are shown in Figure 3.5

The Dimensionally-Extended Models (DE-4IM and DE-9IM)

The 4-Intersection Model makes it possible to identify topological relations between spatial objects of the same dimension (for example surface/surface). To make it possible to also identify topological relations between spatial objects of different dimensions (e.g., line/surface), Clementini et al. [CF96] extended the model by considering the dimension of the intersection in the intersection matrix. In this extended model, the entries in the matrix do not contain empty or no-empty value, but instead the maximum dimension (-1, 0, 1, 2) of the intersection geometries, with a value of -1 corresponding to empty intersection. The extended models are called Dimensionally-Extended 4-Intersection Model and Dimensionally-Extended 9-Intersection Model for 4-IM and 9-IM respectively.
3.2.2 Topology realisation options

Based on the way of expressing topological relationships among component objects of a feature collection, there are three commonly used models of representing feature collections: spaghetti, network and topological models [RSV02].

**Spaghetti Model**

The spaghetti model is a vector representation of spatial objects where lines and regions are represented as an ordered list of points, a point being a pair of coordinates [RSV02]. In this model, no topology information is stored, and all topological relationships must be computed on demand.

With the notation: [ ] for tuple, ⟨⟩ for list, and { } for set, the structure for points, polylines, polygons, and regions can be summarised as follows:

- **point**: [x: real, y: real]
- **polyline**: ⟨point⟩
- **polygon**: ⟨point⟩
- **region**: { polygon }

**Network model**

This model was initially designed for representing networks for transportation, utilities, and so on. This model introduces two new types: node and arc (sometimes called edge). A node is a distinguished point that connects a list of arcs. An arc is a polyline starting at a node and ending at another node. A node is either an arc end point or an isolated point in the plane. A point is represented by a pair of coordinates, a node by a tuple containing the point coordinates and a list of arcs originating from it. An arc is represented as a tuple containing its starting node, its ending node and a list of intermediate vertices, and a polygon is represented by the ordered list of vertices of its polygon boundaries [RSV02]. This representation can be summarised as follows:

- **point**: [x: real, y: real]
- **node**: [point, ⟨arc⟩]
- **arc**: [node-start, node-end, ⟨point⟩]
- **polygon**: ⟨point⟩
- **region**: { polygon }

**Topological model**

The topological model is similar to the network model, except that the network is planar, which is not necessarily the case for the network model. This planarity implies a planar subdivision into adjacent polygons, some of which may not correspond to actual geographic objects [RSV02]. The representation differs from that in the network model on arc and polygon. To represent an arc, the left and right polygons to the arc are also specified. A polygon is represented by an ordered list of arcs that form its boundaries. The representation in this
3.2. Feature topology

model can be summarised as follows:

\[
\begin{align*}
\text{point} & : \{x: \text{real}, y: \text{real}\} \\
\text{node} & : \{\text{point}, \langle \text{arc} \rangle\} \\
\text{arc} & : \{\text{node-start}, \text{node-end}, \text{left-poly}, \text{right-poly}, \langle \text{point} \rangle\} \\
\text{polygon} & : \langle \text{arc} \rangle \\
\text{region} & : \{\text{polygon}\}
\end{align*}
\]

Evaluation of topology realisation options

Each of the models of realising the feature topology has its pros and cons. For each model, some advantages and disadvantages are described in the following.

The spaghetti model has the advantage of being simple to understand and use, and it is easy to add new objects to the collection because all objects are stored independently. However, the spaghetti model has also drawbacks such as representation redundancy (e.g., a shared boundary is stored twice). This redundancy leads to inefficient use of storage capacity when the data sets become large. Furthermore, it carries a risk of inconsistency as the replicated boundary of two adjacent objects may have slightly different coordinates due to different sources of information [RSV02]. Also, since no explicit storage of information on topological relationships, the execution of queries involving topological relationships may be less efficient. The spaghetti model explicitly represents all the entities being modelled. This is an advantage when it is selected as the PSM in the MDA approach where we need to describe as much as possible the entities being modelled at both the PIM and PSM, to enable a good mapping of PIM to PSM.

The network model has advantages of supporting efficient line connectivity tests and network computations, such as shortest paths. It is easy to navigate through the network, by choosing the arc to follow whenever a node is encountered [RSV02]. However, since the line connectivity is the only topological information explicitly stored; without information on topological relationships between two-dimensional objects (area features), the execution of queries involving topological relationships may be less efficient like in spaghetti model. Also the storage of line connectivity makes the model complex to understand and use compared to spaghetti model. The network model describes very well linear features but it poorly describes area features. Given that this thesis focuses on area features and that this model contains less information on area features, it is not a suitable PSM candidate for a good mapping in the MDA approach in the context of this project.

The topological model has the advantage that the execution of queries involving topological relationships is efficient. Also, consistency maintenance and updates are easier with the topological model than with the spaghetti model; the geometry of each shared edge is stored once and not twice as in spaghetti model where update propagation needs to be handled. The general disadvantage of this
model is the cost of computing and maintaining the planar graph (computing all the intersections). This makes some operations like map overlay, extremely expensive [KS00]. Similarly, the addition of a new object to the collection requires the precomputation of part of the planar graph. The complexity of the structure resulting from this model may slow down some operations [The01]. For example, displaying a part of a map requires scanning a set of line segments. Moreover, with topological model, scanning is much slower than with spaghetti model [KS00]. This model does not describe explicitly all entities being modelled but it includes a detailed description of the relationships between components of those entities which makes it a rich PSM in the MDA approach.

There are many considerations that must be taken during the selection of the model to use. Some of these considerations are the following:

- The type of data to be stored: for some data, a certain model is more suitable than others. For example, for roads connectivity data, the network model is the most suitable model.

- The operations that are most likely to be performed on the data. For example, if the expected queries will never involve topological relationships, the spaghetti model is the most suitable one because it will allow the needed fast data retrieval while avoiding the complex data structure [The01].

- The DBMS to be used: A DBMS may have explicit support for certain models and not others. For example, PostgreSQL/PostGIS has no explicit support for the network model.

### 3.2.3 Feature topology support in Database Management Systems

In general, DBMSs support spatial data through an extension to the conventional DBMS components. The simplicity of the spaghetti model has made it the default option in such extension. DBMSs then add support for topological data structure. The support for feature topology in two DBMSs (Oracle and PostgreSQL) is presented in brief in the following.

**Feature topology support in Oracle**

Oracle DBMS, starting from version 8i, handles spatial data through its extension *Oracle Spatial* [RSV02]. In addition to the spaghetti model, Oracle supports also topological and network models. With Oracle Spatial topology model you can store information about topological elements and geometry layers, perform certain spatial operations referencing the topological elements, for example, finding which chains (such as roads) have any topological spatial relationship with a specific polygon entity (such as a park). You can also export topology data from one database and import it into another database.
3.2. Feature topology

The topology data model is built on basic topological elements *node*, *edge* and *face*. A topology geometry (also referred to as a feature) is a spatial representation of a real world object. The geometry is stored in terms of topological elements (nodes, edges, and faces). A topology geometry layer is a collection of topology geometries of a specific type. For example, museums might be the topology geometry layer that includes the Science museum topology geometry which is stored as a node element. In some topologies, the topology geometry layers have one or more parent-child relationships hence making a topology hierarchy. A multilevel hierarchy is also supported. To use the Spatial topology capabilities, you must first insert data into special edge, node, and face tables, which are created by Spatial when you create a topology. Spatial maintains a relationship information table for feature tables and topologies associated to them. Spatial also has the topology data model application programming interface (API) that can be used to further manipulate the data structure and perform various operations.

The network data model in Oracle Spatial shares many concepts with the topological data model. The network model is built on the basic elements *node* and *link* (sometimes called edge or segment). The network model contains logical information such as connectivity relationships among nodes and links, directions of links, and costs of nodes and links. With logical network information, you can analyze a network and answer questions, many of them related to path computing and tracing. This logical information makes a logical network. In additional to logical network information, spatial information such as node locations and link geometries can be associated with the logical network, hence making the spatial network. A spatial network can be directed or undirected, depending on the application. The network model in Oracle Spatial also supports the network hierarchy. The network modelling capabilities of Spatial include schema objects and an application programming interface (API). The schema objects include metadata and network tables. The API includes a server-side PL/SQL API (the SDO_NET package) for creating, managing, and analyzing networks in the database, and a client-side Java API for network analysis. A detailed discussion of the topology and network data models can be found in [Mur03].

**Feature topology support in PostgreSQL**

PostgreSQL DBMS supports spatial data through its extension PostGIS. PostGIS provides geometry operations and most importantly topological functions using the GEOS module [CJE94]. GEOS (Geometry Engine - Open Source) is a C++ version of a part of the Java Topology Suite (JTS), a Java API of 2D spatial predicates and functions for fundamental geometric operations. PostgreSQL/PostGIS supports the spaghetti model and currently provides the topological model as an option to be enabled. To enable the topological model, you need a schema-aware PostgreSQL installation (version 7.3 and up), PostGIS-1.1.x, and GEOS-2.1 or up. Enabling the topology creates a separate schema that will contain the topology-enabling tables and topology-related functions.
Currently, the support for the topology data model in PostGIS is in pre-alpha stage [Ref05]. With the topology support enabled you can explicitly store features’ topology using predefined topological elements (faces, edges and nodes), define geometries composed by those elements and convert these geometries to simple Geometry objects to make it possible to use all standard functions defined on the latter. With the same support, it is also possible to validate the topology (this is done by retrieving the features that violate the topological relationships), and to perform basic topology editing operations.

Our observation on the feature topology support in the DBMSs is that a manipulation of stored spatial data, with interest on topological relationships, requires the database designer or user to perform an extensive programming using the provided API. This thesis tries to relieve the database designer/user of this task in some basic operations by taking care of it in the transformational design process. Being the most widely used open source spatial DBMS [RSV02], PostgreSQL/PostGIS has been selected to be used in the implementation phase of this thesis.
3.2. Feature topology
Chapter 4

Design of area collection modelling primitives and the transformation process

This chapter describes the design of primitives to model collections of area features. To allow their reusability in different models, these primitives are designed using the standard extensibility mechanisms available in UML (stereotypes, tagged values and constraints) which have been described in Section 2.3. Furthermore, the primitives are packaged in a UML profile to allow their portability to different projects and modelling tools. This chapter also describes the design of the transformation process in which the result at one database design level is transformed to another.

4.1 Design of a UML profile for modelling collections of area features

An area collection is a spatial object made of one or more related area objects. It can be represented by a polygon if it is made of only one element, or by a set of polygons (a multipolygon) if it is made of more than one element. The inherent characteristic of area collections is that a collection is made of areas that have mutually disjoint interiors. Considering other specific characteristics, we can classify area collections into a number of categories.

Specific characteristics of area collections are such as whether the objects making all the related area collections form a contiguous area, whether the elements in one collection are disjoint, touch in points or in line segments and whether there are area collections with a semantic spatial aggregation connection. A contiguous area is an area in which, from any point in it there exists a path, fully contained in the area, to any other point in the area. A semantic spatial aggregation connection means a relationship in which an object at one level is made of the geometric union of some objects at another level. Based on these characteristics we classify area collections into the following categories which are illustrated in Figure 4.1:
4.1 Design of a UML profile for modelling collections of area features

1. Collections representable as polygons that make up a contiguous area (Figure 4.1(a))
2. Collections representable as multipolygons that make up a contiguous area (Figure 4.1 (b))
3. Collections made of disjoint areas (Figure 4.1 (c))
4. Collections of discretely touching areas (Figure 4.1 (d))
5. Collections of continuously touching areas (Figure 4.1 (e))
6. Hierarchy of areas (Figure 4.1 (f))

This is not an exhaustive list of categories of area collections. A different categorisation can be made based on other specific characteristics for a specific application. For instance, based on whether objects in an area collection can include holes, we can have a category of area collections representable as polygons without holes and that make up a contiguous area. We can have another category of area collections representable as polygons that make up a contiguous area, but that can also contain objects with holes. Our categorisation is one
possible general example that considers mostly encountered real world objects without considering the details of a specific application. It is up to the user to decide which modelling primitive most suits the application at hand.

The set of new modelling primitives will be made available as a UML profile containing six stereotypes, one for each of the aforementioned categories. For each of the six stereotypes, the following can be or are specified: the name of the stereotype that appears in the model, the base class which is the UML meta-model element that serves as the base for the stereotype, a short description of the stereotype, an example of a real world object that can be modelled using this stereotype, an icon providing a distinctive visual clue, constraints that restrict the real world objects that can be modelled using the stereotype, and a list of tagged values which specify some specific properties that the stereotype carries. Each stereotype is associated with a set of maintenance functions to be used for storing and manipulating the objects of the type the stereotype represents. A design of these functions is presented after the stereotypes.

**Stereotypes**

1. For collections representable as polygons that make up a contiguous area

   **Name:** ContiguousPolygonalArea  
   **Base class:** Class  
   **Description:** A *ContiguousPolygonalArea* class is a class of which the instances represent objects representable as polygons that make up a contiguous area.  
   **Example:** Parcel; parcels form a contiguous area which is the area they occupy and each parcel can be represented by a polygon.  
   **Icon:** None  
   **Constraints:**  
   - Each object can be represented by a polygon  
   - The geometric union of all areas forms a contiguous area  
   - Areas do not overlap  
   **Tagged values:** None

2. For collections representable as multipolygons that make up a contiguous area

   **Name:** ContiguousMultipolygonalArea  
   **Base class:** Class  
   **Description:** A *ContiguousMultipolygonalArea* class is a class of which the instances represent objects representable as multipolygons that make up a contiguous area.  
   **Example:** Land use type; each land use type can be found in one or more separ
4.1. Design of a UML profile for modelling collections of area features

Rate areas and can be represented by a set of polygons. The geometric union of those land use types form a contiguous area which is the entire study area.

**Icon:** None

**Constraints:**
- Each object can be represented by a set of one or more polygons
- The geometric union of the representations of all objects forms a contiguous area
- Areas do not overlap

**Tagged values:** None

3. For a collection made of disjoint areas

**Name:** DisjointArea

**Base class:** Class

**Description:**
A DisjointArea class is a class of which the instance represents an object made of one or more areas which are disjoint.

**Example:**
Island country; each island country consists of one or more islands and any two islands are disjoint.

**Icon:** None

**Constraints:**
- The areas in the collection are mutually disjoint

**Tagged values:** None

4. For a collection of discretely touching areas

**Name:** PointTouchArea

**Base class:** Class

**Description:**
A PointTouchArea class is a class of which the instance represents an object made of one or more areas which are disjoint or touch at a finite number of points. A DisjointArea object is a special type of PointTouchArea object in that its elements are mutually disjoint. Real world objects that share a finite number of points are not common. In general, this class contains erroneous representations of real world objects, where the errors are due to data acquisition techniques such as digitisation.

**Example:**
University (where each university can have many campuses each being considered as one element of the university object); if the campuses of any university can not share a boundary, the University is a PointTouchArea class.

**Icon:** None

**Constraints:**
- Any two areas in the collection are disjoint or they touch along zero-dimensional intersections (points).
Tagged values: None

5. For a collection of continuously touching areas

Name: LineTouchArea  
Base class: Class  
Description:  
A LineTouchArea class is a class of which the instance represents an object made of one or more areas which are disjoint or touch along either zero-dimensional (points) or one-dimensional (line segments) intersections. A DisjointArea object is a special type of LineTouchArea object in that its member areas can only be disjoint. Likewise, a PointTouchArea object is a special type of LineTouchArea object in that its member areas can only be disjoint or share a finite set of points but can not share line segments.

Example:  
Boarding school; if each boarding school is made of one or two adjacent compounds (one for male students and the other for female students), the boarding school is a LineTouchArea class.

Icon: None  
Constraints:  
• Any two areas in the collection do not overlap

Tagged values: None

6. For a hierarchy of areas

Name: AreaHierarchyAssociation  
Base class: Association  
Description:  
An AreaHierarchyAssociation association establishes a semantic spatial aggregation connection between area collections such that a collection at an upper level equals the geometric union of some collections at a lower level.

Example:  
An AreaHierarchyAssociation association exists between a State and its Provinces, and also between a province and its Districts.

Icon: None  
Constraints:  
• Objects at two associated levels (upper and lower levels) are of the same area collection type  
• The geometry of any object at the upper level is equal to the geometric union of some objects at the lower level  
• Each lower level object can be involved in the geometric union to make only one immediate upper level object.

Tagged values: None
Design of maintenance functions

Database Management Systems (DBMS) that store spatial data have a set of functions used to store, manipulate and retrieve stored data. However these functions do not perform or perform very limited special treatment of maintaining topological dependencies between and within area collections. To enhance this special treatment of area collections, we design a set of functions that will be carried by the new modelling primitives.

For ContiguousPolygonalArea and ContiguousMultipolygonalArea objects we propose the functions to carry out the following operations:

- to add a new ContiguousPolygonalArea object to the set of existing ContiguousPolygonalArea objects, or to add a new ContiguousMultipolygonalArea object to the set of existing ContiguousMultipolygonalArea objects
- to delete a ContiguousPolygonalArea object or a ContiguousMultipolygonalArea object
- to modify a part of the boundary of a ContiguousPolygonalArea or a ContiguousMultipolygonalArea object (shrinking or enlarging the object)
- to merge two ContiguousPolygonalArea objects or two ContiguousMultipolygonalArea objects

Additionally, for ContiguousMultipolygonalArea objects, we propose the functions to carry out the following operations:

- to modify an existing ContiguousMultipolygonalArea object adding to it a new component area
- to modify an existing ContiguousMultipolygonalArea object deleting one component area
- to change the membership of a component area from one ContiguousMultipolygonalArea object to another

AddToContiguousArea( )

Description:
AddToContiguousArea adds a new ContiguousPolygonalArea object to the set of existing ContiguousPolygonalArea objects, or a new ContiguousMultipolygonalArea object to the set of existing ContiguousMultipolygonalArea objects, preserving topological dependencies between the objects.

Input:
the name of the table that will contain the objects, an array of names of non geometry columns into which values are to be inserted, an array of values to insert into non geometry columns and the geometry value of the object to be added.
**Chapter 4. Design of area collection modelling primitives and the transformation process**

**Output:** The text *Success* if the operation was successful or the cause of failure otherwise.

**Algorithm:**

AddToContiguousArea (T: text, u: text[], v: text[], w: geometry): text

**Step 1:** Check if the new object has an intersection relationship which is not a touching relationship with any existing object. If it does, set the error message, then go to step 4.

**Step 2:** Check if the geometric union of existing and new objects would have gaps or holes. If it would, set the error message, then go to step 4.

**Step 3:** Insert the new object. Set the message to Success.

**Step 4:** Return the message.

The pseudocode of this function is shown in Algorithm 1.

```plaintext
begin
//count the collections overlapping with the new collection
n = [[s ∈ T | Intersect(w, s.geom) ∧ ¬(Touch(w, s.geom)) • (Count(*))]]
if n ≠ 0 then
    return “insertion failed due to overlapping with other collection”
end if
//compute the geometric union of the existing objects and the new one
f = [[s ∈ T | • (GeomUnion(s.geom))]]
q = GeomUnion(f, w)
if IsContiguous(q) and NumHoles(q) = 0 then
    T = T ∪ { ⟨v, w⟩ } // insert the new area collection
    return “Success”
else
    return “Failed. The new object would introduce holes or gaps”
end if
end
```

**DeleteFromContiguousArea ( )**

**Description:**

DeleteFromContiguousArea deletes a ContiguousPolygonalArea or ContiguousMultipolygonalArea object from the set of existing objects, preserving the topological dependencies between the objects.

**Input:** the name of the table containing the objects, the identifier of the object to be deleted.

**Output:** the text *Success* if the operation was successful or the cause of failure otherwise.

**Algorithm:**

DeleteFromContiguousArea (T: text, i: integer): text
4.1. Design of a UML profile for modelling collections of area features

**Step 1:** Compute the geometric union of existing objects except the one to be deleted.

**Step 2:** Check if the computed geometry is contiguous and contains no holes. If it is not so, set the error message and go to step 4.

**Step 3:** Delete the object and then set the message to Success

**Step 4:** Return the message

The pseudocode of this function is shown in Algorithm 2

```plaintext
begin
//compute the geometric union of all collections except the one to be deleted
let g = \{s ∈ T | ¬(Id(s) = i) • (GeomUnion(s.geom))\}
if IsContiguous(g) and NumHoles(g) = 0 then
    T = T \ {s ∈ T | Id(s) = i} // delete the area collection
    return "Success"
else
    return "Failure: the deletion would introduce holes or gaps"
end if
end

Algorithm 2: DeleteFromContiguousArea (T: text, i: integer): text
```

**MergePolygonalAreas ( )**

**Description:**

*MergePolygonalAreas* merges two *ContiguousPolygonalArea* objects, making one *ContiguousPolygonalArea* object.

**Input:**
the first argument is the name of the table containing the objects, the second is the identifier of the first object, the third is the identifier of the second object, and the fourth argument is an integer (1, 2 or 0) specifying how to handle non-spatial attribute values; 1 means that non-spatial attribute values of the first object will be copied for the new object, 2 means that non-spatial attribute values of the second object will be kept, and 0 means that non-spatial attributes of the new object will be set to NULL.

**Output:** the text *Success* if the operation was successful or the cause of failure otherwise

**Algorithm:**

```
MergePolygonalAreas (T: text, i: integer, j: integer, k: integer): text

Step 1: Check if the input value in the last argument is valid (if it is 0 or 1 or 2). If it is not, set error message then go to step 5.

Step 2: Check if the two objects touch in one-dimensional intersections (line segments). If they do not, set error message then go to step 5.

Step 3: Compute a geometry which is the union of the geometries of the two objects.
```
**Step 4:** Check the value of the last argument. If it is 1, update the geometry of the first object by setting it to the geometry computed in step 2, and then delete the second object. If it is 2, update the geometry of the second object by setting it to the geometry computed in step 2, and then delete the first object. If it is 0, insert a new object setting the value of its geometry attribute to the geometry computed in step 2 and thematic attributes to NULL value, and then delete the two input objects. Set the message to Success

**Step 5:** Return the message

The pseudocode of this function is shown in Algorithm 3

**MergeMultipolygonalAreas ( )**

**Description:**

*MergeMultipolygonalAreas* takes two *ContiguousMultipolygonalArea* objects and merges them making one *ContiguousMultipolygonalArea* object.

**Input:**

the first argument is the name of the table containing the objects, the second is the identifier of the first object, the third is the identifier of the second object, and the fourth argument is an integer (1, 2 or 0) specifying how to handle non-spatial attribute values; 1 means that non-spatial attribute values of the first object will be copied for the new object, 2 means that non-spatial attribute values of the second object will be kept, and 0 means that non-spatial attributes of the new object will be set to NULL.

**Output:** the text *Success* if the operation was successful or the cause of failure otherwise

**Algorithm:**

*MergeMultipolygonalAreas (T: text, i: integer, j: integer, k: integer): text*

**Step 1:** Check if the input value in the last argument is valid (if it is 0 or 1 or 2). If it is not, set the error message and go to step 4.

**Step 2:** Compute a geometry which is the union of the geometries of the two objects.

**Step 3:** Check the value of the last argument. If it is 1, update the geometry of the first object by setting it to the geometry computed in step 2, and then delete the second object. If it is 2, update the geometry of the second object by setting it to the geometry computed in step 2, and then delete the first object. If it is 0, insert a new object setting the value of its geometry attribute to the geometry computed in step 2 and thematic attributes to NULL value, and then delete the two input objects. Set the message to Success

**Step 4:** Return the message

The pseudocode of this function is shown in Algorithm 4
begin
  gm = \emptyset
  //variable initialized to empty geometry
  result = “Operation failed; check your inputs”
  if k \notin \{0, 1, 2\} then
    result = “Failure; the value in the last argument should be 0 or 1 or 2”
  else
    p = \{ s \in T \mid -Id(s) = i \bullet \{s.geom\} \}
    q = \{ s \in T \mid -Id(s) = j \bullet \{s.geom\} \}
    if Dimension(Intersection(p, q)) \neq 1 then
      result = “Failure; the collections can not be represented as one polygon”
    else
      //compute the geometric union of the two collections
      gm = GeomUnion(p, q)
      if k = 1 and notIsEmpty(gm) then
        for each r in T do
          if Id(r) = i then
            Set r.geom = gm //update the geometry of the 1st collection
            exit for
          end if
        end for
        T = T \{ s \in T | Id(s) = j \} // delete the second collection
        result = “success”
      else if k = 2 and notIsEmpty(gm) then
        for each r in T do
          if Id(r) = j then
            Set r.geom = gm //update the geometry of the 2nd collection
            exit for
          end if
        end for
        T = T \{ s \in T | Id(s) = i \} // delete the first collection
        result = “success”
      else if k = 0 and notIsEmpty(gm) then
        // insert a new collection with null values for non-spatial attributes
        // and the computed geometric union for spatial attributes
        T = T \cup \{ \langle NULL, gm \rangle \}
        T = T \{ s \in T | Id(s) = i \} // delete the first collection
        T = T \{ s \in T | Id(s) = j \} // delete the second collection
        result = “success”
      end if
    end if
  end if
  return result
end

Algorithm 3: MergePolygonalAreas(T:text, i:integer, j:integer, k:integer): text
Chapter 4. Design of area collection modelling primitives and the transformation process

begin

\(gm = \emptyset\)

//variable initialized to empty geometry
result = “Operation failed; check your inputs”
if \(k \notin \{0, 1, 2\}\) then
result = “Failure; the value in the last argument should be 0 or 1 or 2”
else
\(p = \{ s \in T \mid \neg Id(s) = i \cdot \langle s, geom \rangle \}\)
\(q = \{ s \in T \mid \neg Id(s) = j \cdot \langle s, geom \rangle \}\)
//compute the geometric union of the two collections
\(gm = \text{GeomUnion}(p, q)\)
if \(k = 1\) and not IsEmpty\(\text{gm}\) then
for each \(r\) in \(T\) do
if \(Id(r) = i\) then
Set \(r.geom = gm\) //update the geometry of the 1st collection
exit for
end for
end if
end for
\(T = T \{ s \in T \mid Id(s) = j \}\) // delete the second collection
result = “success”
else if \(k = 2\) and not IsEmpty\(\text{gm}\) then
for each \(r\) in \(T\) do
if \(Id(r) = j\) then
Set \(r.geom = gm\) //update the geometry of the 2nd collection
exit for
end if
end for
\(T = T \{ s \in T \mid Id(s) = i \}\) // delete the first collection
result = “success”
else if \(k = 0\) and not IsEmpty\(\text{gm}\) then
// insert a new collection with null values for non-spatial attributes
// and the computed geometric union for spatial attributes
\(T = T \cup \{ \langle \text{NULL}, gm \rangle \}\)
\(T = T \{ s \in T \mid Id(s) = i \}\) // delete the first collection
\(T = T \{ s \in T \mid Id(s) = j \}\) // delete the second collection
result = “success”
end if
end if
return result
end

Algorithm 4: MergeMultiPolygonalAreas (T: text, i: integer, j: integer, k: integer): text
AddPolygonToMultiArea( )

Description: AddPolygonToMultiArea modifies an existing ContiguousMultipolygonalArea object by adding one polygon to its member polygons, preserving the topological dependencies within that object and between existing ContiguousMultipolygonalArea objects.

Input:
the name of the table containing the object, an integer value which is the identifier of the object to which the polygon is to be added and the geometry value of the polygon to be added.

Output:
the text Success if the operation was successful or the cause of failure otherwise

Algorithm:
AddPolygonToMultiArea (T: text, i: integer, g: geometry): text

Step 1: Check if the new polygon has an intersection relationship which is not a touching relationship with any existing objects. If it does, set the error message, then go to step 4

Step 2: Check if the geometric union of the existing objects and the new polygon would have gaps or holes. If it would, set error message, then go to step 4.

Step 3: Add the new polygon to the object. It is done by updating the geometry of the object of which the id was input, setting it to the union of this geometry and the new polygon. Set the message to Success.

Step 4: Return the message

The pseudocode of this function is shown in Algorithm 5

DeletePolygonFromMultiArea( )

Description:
DeletePolygonFromMultiArea deletes a polygon from an existing ContiguousMultipolygonalArea object, preserving the topological dependencies within that object and between existing ContiguousMultipolygonalArea objects.

Input:
the name of the table containing the object, an integer value which is the identifier of the object containing the polygon to be deleted, the polygon number of the polygon to be deleted

Output:
the text Success if the operation was successful or the cause of failure otherwise.
Algorithm:
DeletePolygonFromMultiArea (T: text, i: integer, num: integer): text

Step 1: Compute the geometry difference between the geometric union of all existing objects and the polygon to be deleted (polygon of which the polygon number was input).

Step 2: Check if the computed geometry is a contiguous area without holes. If it is not, set the error message, then go to step 4.

Step 3: Delete the polygon of which the polygon number was input. This is done by updating the geometry of the object, setting it to the difference between this geometry and the polygon to be deleted. Set the message to Success.

Step 4: Return the message

The pseudocode of this function is shown in Algorithm 6

ReassignCollection( )

Description:
ReassignCollection changes the membership of a polygon from one ContiguousMultipolygonalArea object to another, preserving topological dependencies between existing ContiguousMultipolygonalArea objects.

Input:
four arguments; the name of the table containing the two objects, an integer which is the identifier of the source object, an integer which is the identifier of the destination object, and the polygon number of the polygon for which membership is to be changed.

Output:
the text Success if the operation was successful or the cause of failure otherwise.

Algorithm:
ReassignCollection( T: text, src: integer, dest: integer, num: integer ): text

Step 1: Update the geometry of the source object by setting it to the difference between the current geometry value and the polygon concerned.

Step 2: Update the geometry of the destination object by setting it to the geometric union of the current geometry value and the polygon for which membership is to be changed. Set the message to Success.

Step 3: Return the message

The pseudocode of this function is shown in Algorithm 7
begin

result = “Operation failed; check your inputs”

//check if the new area overlaps with any existing collection by
//counting the collections overlapping with the area to be inserted

\[ n = \left| \left| \left| s \in T \mid \text{Intersect}(g, s.\text{geom}) \land \neg \text{Touch}(g, s.\text{geom}) \right| \right| \right| \]

if \( n \neq 0 \) then

result = “insertion failed due to overlapping with existing objects”

else

//compute the geometric union of the existing objects and the new polygon

\[ h = \left| \left| s \in T \cdot \langle \text{GeomUnion}(s.\text{geom}) \rangle \right| \right| \]

\[ h = \text{GeomUnion}(h, g) \]

if IsContiguous(h) and NumHoles(h) = 0 then

for each \( r \) in \( T \) do

if Id(r) = i then

//update the geometry of the collection

Set r.\text{geom} = \text{GeomUnion}(r.\text{geom}, g)

result = “Success”

end if

end for

else

result = “Operation failed because it would introduce holes or gaps”

end if

end if

end

end

Algorithm 5: AddPolygonToMultiArea (T: text, i: integer, g: geometry): text

begin

result = “Operation failed; check your inputs”

\[ g = \text{GeomDifference}(\text{Union}(\left| \left| s \in T \cdot \langle s.\text{geom} \rangle \right| \right|), (\left| \left| s \in T \mid s.gid = i \cdot \langle s.\text{geom}(\text{num}) \rangle \right| \right|)) \]

if IsContiguous(h) and NumHoles(h) = 0 then

for each \( r \) in \( T \) do

if Id(r) = i then

Set r.\text{geom} = \text{GeomDifference}(r.\text{geom}, (r.\text{geom})(\text{num}))

return “success”

end if

end for

else

result = “operation failed; it would violate topological dependencies”

end if

end

end

Algorithm 6: DeletePolygonFromMultiArea (T: text, i: integer, num: integer): text
Chapter 4. Design of area collection modelling primitives and the transformation process

begin
result = “Operation failed; check your inputs”
g:geometry
k = 0 //number of successful steps
for each r in T do
if Id(r) = src then
  g = (r.geom)(num)
  Set r.geom = GeomDifference(r.geom, g)
  k = k + 1
  exit for
end if
end for
for each r in T do
if Id(r) = dest then
  g = (r.geom)(num)
  Set r.geom = GeomUnion(r.geom, g)
  k = k + 1
  exit for
end if
end for
if k = 2 then
  result = “success”
end if
return result
end

Algorithm 7: ReassignCollection(T: text, src: integer, dest: integer, num: integer): text

4.2 Design of the transformation process

The Model Driven Architecture (MDA) approach simplifies the database design task through a separation of concerns. This is done by performing the design process at a number of levels of abstraction and mapping the results from level to level. Those levels of design are the conceptual design, logical design and the physical design and the abstractions resulting from them are conceptual schema, logical schema and physical schema respectively.

Conceptual schema

The conceptual schema is a representation of the conceptual model produced during the conceptual design. A conceptual schema uses concepts that are easily understood to represent real world objects relevant in the application domain. This has been illustrated in Figure 3.1. The conceptual schema specifies the object types that represent the things and concepts, their properties and their relationships, that are relevant in the problem space [Amb04]. The conceptual design uses a modelling language which is very often graphical-based [PM07]. The advantages of UML (see Section 2.3) have made UML our choice for a modelling language. In particular, the UML extensibility mechanisms allow us to introduce new modelling primitives for area collections. In UML, the concept of class is used to specify the object type, attributes and operations are used to represent the properties, and relationships (association, ag-
4.2. Design of the transformation process

Aggregation, composition and inheritance) are used to represent the relationships between objects. A detailed discussion of these object-oriented concepts can be found in [Amb04]. The conceptual schema will be built using the modelling primitives provided by the modelling language and the additional modelling primitives developed using UML extensibility mechanisms.

**Logical schema**

While the conceptual design concentrates on specifying the data contents of a database, the logical design specifies the logical structure of the data in the database. That is, it specifies the structure of the data that a specific DBMS will present to the user. A logical schema represents real world objects, their properties and relationships using the concepts of tables, attributes (or columns), primary keys, foreign keys, etc., as they are available in the DBMS being used. These concepts form the relational data model, named so because the basic unit in this data model is the 'table' (also referred to as a relation). A detailed discussion of the relational data model concepts can be found in [EN07].

**Physical schema**

The physical design specifies the schema, obtained from logical design, in the SQL of the DBMS chosen for implementation. In addition to this, the physical design fine-tunes the logical schema to make efficient use of the physical storage space and enhance the performance of the database according to the requirements of the application [MVC03]. The concepts of indexes are used at this level of design. In this research, the physical design activities will be limited to expressing the database schema in the SQL of the target DBMS and a performance consideration limited to SQL statements to create indexes.

**Considerations related to spatial data**

The description of the schema levels given above is general for the design of any database. Since this research deals with spatial databases, it further considers some spatial data peculiarities described in chapter 3. These are for instance spatial data types and spatial topological relationships. During the conceptual design, *area collections* will be modelled using the modelling primitives that have been presented in Section 4.1. Spatial indexes will be considered to fine-tune the logical schema for better database performance.

**The transformation process**

The major advantage of MDA is the ability to transform one type of model into another type, and this being done with a certain level of automation. In the context of MDA, the conceptual schema is a Platform Independent Model (PIM) whereas the logical and physical schemas are Platform Specific Models (PSM). Therefore the MDA approach allows the transformation from conceptual schema to logical schema and from logical schema to physical schema without excluding the transformation at the same schema level. However the transformation from conceptual schema to logical schema is the most challenging due
Chapter 4. Design of area collection modelling primitives and the transformation process

to the “impedance mismatch” between the two schema levels [Amb04]. The two schemas are based on different paradigms (the object-oriented paradigm and the relational paradigm respectively) leading to mismatch such as concept in one schema missing a corresponding concept in the other schema. For example, the inheritance relationship available in conceptual schemas has no direct corresponding concept in logical schemas for most of DMBSs. The diagram in Figure 4.2 illustrates the transformational database design process of which the transformation process is a part.

The transformation from conceptual schema to logical schema and the transformation from logical schema to physical schema are described in the following.

**From conceptual schema to logical schema**

The transformation from conceptual schema to logical schema takes as input a UML class diagram, which is the conceptual model of the database, and produces as output another UML class diagram that represents the relational model of the database. The transformation is done following a set of MDA transformation rules. Since the logical schema is a PSM, the MDA transformation rules are influenced by the implementation choice. General MDA transformation rules are listed below. Customisations of those rules for handling spatial data concepts are mentioned also. A detailed description of these customisations will be presented in a section on the implementation of the transformation process where implementation choices are considered.

1. Transform each class to one table: in the simplest case each class is transformed to one table. However some situations require further decision; if classes are linked by an inheritance relationship, you need to select an inheritance strategy. There are three basic ways to translate classes and their inheritance relationship:

   - For each class hierarchy, create a single corresponding table that contains all the attributes in all classes in the hierarchy. This table is therefore the union of every class in the hierarchy. For example, a class hierarchy made of Member, Teacher, Student and Secretary can be transformed in this way to produce only one table.

   - For each class hierarchy branch, create only one table that will include as columns all the attributes within that branch (including inherited attributes). In this case the parent class will have no corresponding table. For example, if Teacher inherits Member only, then the table will contain attributes of Member and those of Teacher only.

   - For each class in the hierarchy, create one corresponding table containing only the attributes of that class.

An area collection class is transformed into one table in the spaghetti model, but it is transformed into four tables and one type class in the topological model. This transformation will be described in a section on the implementation of the transformation process.
4.2. Design of the transformation process

2. For each table, generate a primary key: a column is created and marked as primary key. In this research, the primary key column is named as the name of the class appended with ID (for identifier). Also additional information may be included (possibly in the form of tagged value) for the creation of a sequence that populates the primary key.

3. Transform attribute to column: at the end of the transformation process, each class attribute will have been transformed to zero or more columns.
   - **Simple class attributes**: each simple class attribute is transformed to one table column. Select the appropriate data type for each column based on the data type of the class attribute and the data types available in the implementation DBMS. In this research, the data types of geometric attributes are taken from the ISO 19107 (Spatial Schema) whereas the data types of the table columns generated from geometric attributes are taken from the Simple Feature Access (SFA) - Part 1 (see Section 3.1.2). The tagged value *srid* that was created on a geometry attribute is transformed to a tagged value on the geometry column produced, to hold the spatial reference system identifier.
   - For each *multivalued attribute* (a collection of values), create a new table and include in it, as foreign key, the primary key column of the table resulting from the class that contains this attribute. Add also a column corresponding to the multivalued attribute. Set a tag value to indicate that the propagate (CASCADE) option will be specified on the foreign keys in the new table, during the transformation from logical schema to physical schema.
   - For *complex attributes* (attributes which represent other classes), the following rules for handling associations and aggregations are used.

4. Transform associations, aggregations and compositions to relationships or tables depending on the multiplicity of the relationship. In the relational model, a relationship is represented by a primary-foreign key pair.
   - **One-to-one relationship**
     To transform a one-to-one relationship there are two strategies. The first strategy is to create a relationship with a foreign key in the table produced from the target class and a primary key in the table produced from the source class. The second strategy is to simply put the columns corresponding to attributes of both classes in one table, hence making only one table out of the two classes.
   - **One-to-many relationship**
     Create a relationship with a foreign key in the table produced from the class on the *many* side of the relationship and a primary key in the table produced from the class on the *one* side of the relationship.
   - **Many-to-many relationship**
     Create a separate table with a primary key made of the primary keys of the two related tables. The new table is linked to the other two
tables by relationships with a multiplicity of one on the other tables’ side and a multiplicity of many on the new table’s side. Set the value for a tag to indicate that the propagate (CASCADE) option will be specified on the foreign keys in the new table, during the transformation from logical schema to physical schema.

- **Association class**
  
  An association class is transformed like a many-to-many relationship, but also the attributes of the class part of the association are transformed to columns that are then added into the table produced from the association part.

A reflexive relationship (one in which the same class is involved on both ends) is transformed in the same way as other relationships; based on the multiplicity. After the transformation of each association relationship type, role information may be added at each end of the relationship. Typically, this is done by including primary key constraint name and foreign key constraint name.

Additional information can be included in the logical schema to specify the indexes, triggers, stored procedures, uniqueness constraints and validity check constraints, that will be created to model the database behaviour. In this research, this will be done to specify spatial constraints, indexes and the maintenance functions designed in this chapter.

The transformation from the input conceptual schema to the final logical schema can be done in several steps depending on the complexity of the conceptual schema. The transformation from conceptual schema to logical schema is done using the functionalities offered by the modelling tool. These functionalities include the standard transformation definitions built in the tool and the transformation definitions that can be developed using the programming environment of the modelling tool.

**From logical schema to physical schema**

The final logical schema is transformed into a set of statements written in the SQL of the DBMS chosen for implementation. This is a PSM to PSM transformation. These SQL statements include Data Definition Language (DDL) statements used to create the database structure, and Data Manipulation Language (DML) statements used to work on the data in the database. Some modelling tools provide the functionality to generate DDL and DML statements from the logical schema. However, this default functionality is only suitable in a few simple cases. For instance, these modelling tools do not come with support for modelling spatial database and therefore their DDL statement generation functionality does not handle spatial data peculiarities in DBMS such as specific treatment for geometry columns. Moreover, we want to provide a flexibility by allowing the choice between two different implementation options. For these reasons, this research develops a custom tool for performing the logical schema to physical schema transformation.
The custom tool generates SQL statements which are executed in the following sequence to implement the database:

1. Delete objects if they exist: before each object (sequence, table, geometry column, constraint, index and function) is created, it is first deleted if it already exists.

2. Create sequences: sequences that generate values for primary key columns are created before the creation of tables and primary key columns that refer to them.

3. Create tables: tables are created. Columns, except geometry columns, are created and their data types are specified along with “NOT NULL” properties, default values or reference to the next value of a sequence where required and ON UPDATE CASCADE and ON DELETE CASCADE options.

4. Create geometry columns: spatial DBMSs require that geometry columns be created with separate statements after the creation of tables to enable proper maintenance of metadata tables.

5. Create primary key constraints: Primary key constraints are created before referential integrity constraints that use them are created.

6. Create other constraints: other base table constraints (Unique, Check and Foreign key constraints) are created.

7. Create indexes: In addition to the automatically created indexes upon the creation of Primary key and Unique constraints, spatial indexes are created.

8. Create functions and triggers: maintenance functions designed in this chapter are created. If there are any other functions and any triggers that need to be created, they are also created.

There are two options for implementing the designed custom tool to generate SQL statements. The first option is to implement it as an add-in in the modelling tool such that after selecting the add-in from the menu the currently selected logical schema is automatically transformed to physical schema. The second option is to implement the tool to run separately from the modelling tool. In the latter option, the logical schema is exported to XMI file which will then be loaded as input to the transformation tool. In both options the output from the tool is a text file containing SQL statements which can be executed to implement the designed database. This research takes the second option to achieve the earlier stated MDA goals of interoperability and reusability; the tool can be used to transform schemas developed on different modelling tools, the only requirement being the possibility to export the schema to XMI file.
Chapter 4. Design of area collection modelling primitives and the transformation process

Figure 4.2: The transformational database design process

[Diagram of the transformational database design process]
4.2. Design of the transformation process
Chapter 5

Implementation of modelling primitives and the transformation process

This chapter presents the implementation of area collections modelling primitives and the transformation process. Section 5.1 presents the implementation tools, Section 5.2 presents the implementation of the primitives while Section 5.3 presents the implementation of the transformation process. Section 5.4 summarises the steps taken to perform the whole transformational design. Finally, Section 5.5 presents the testing that has been done to validate the modelling primitives and the transformation process developed.

5.1 Implementation tools

The implementation of the primitives for modelling area collections requires a UML-based implementation tool with Model Driven Architecture (MDA) support. Among the widely used such tools there are: Enterprise Architect developed by Sparx Systems [Spa], and Rational Rose developed by IBM [IBMa]. Enterprise Architect being one of the widely used tools with support for XML Metadata Interchange (XMI) that is needed in the implementation of the transformation process, it has been chosen for creating the new modelling primitives.

Our choice for the implementation of the transformation process involves using the capabilities provided by the modelling tool and the development of a custom tool to carry out some steps of the transformation. Software resources used in this implementation are:

1. Enterprise Architect 7.1
   Enterprise Architect is used as the modelling tool.

2. Java Software Development Kit 1.6 (JDK 1.6)
   Java is used to develop our custom transformation tool. It has been selected for its wide support for XML parsing that is involved in the trans-
formations, and for our convenience. However a different programming language with XML parsing support can be used.

3. PostgreSQL/PostGIS (PostgreSQL 8.3 and PostGIS 1.3.3)
PostgreSQL/PostGIS is our target Database Management System (DBMS) for the implementation of a designed database. It has been selected because it is the most widely used Open Source Spatial DBMS and because of its topological model that we want to test and improve.

4. SQuirreL SQL Client
This tool is used to interact with PostgreSQL/PostGIS throughout the implementation and testing phases.

5. Quantum GIS (QGIS 0.11.0 Metis)
QGIS is used for spatial data visualization that is needed during the implementation and testing phases.

5.2 Implementation of primitives for modelling area collections

The area collections modelling primitives that have been presented in Section 4.1 are created in Enterprise Architect. As shown in Figure 5.1, five modelling primitives; ContiguousPolygonalArea, ContiguousMultiPolygonalArea, DisjointArea, PointTouchArea and LineTouchArea extend the standard Class modelling primitive, while the AreaHierarchyAssociation extends the standard Association modelling primitive. These modelling primitives are packaged in a UML profile that is then made available as an XML file that a user can import into his/her Enterprise Architect project.

The functions associated with these modelling primitives are implemented using the procedural language of PostgreSQL (PL/pgSQL). A description of the features of PL/pgSQL can be found in [Pos08]. Among the functions designed and presented in Section 4.1, current implementation includes only functions associated with the ContiguousPolygonalArea modelling primitive. Functions are implemented to insert ContiguousPolygonalArea objects, to merge two objects and to delete objects. The function AddToContiguousArea is implemented to insert only one object. With this function, the order of insertion of objects matters; the next object to be inserted must be sharing a boundary with at least one existing object. A modified version of this function, AddToContiguousAreaLoader, is implemented for inserting ContiguousPolygonalArea objects without caring about the insertion order. This function requires that objects be first inserted into a temporary table. The function takes as input the name of the temporary table, the name of the destination table or topology for the objects, and an integer specifying the user’s choice of whether to empty the temporary table (value 0) or to delete it (any other value).

Topological model support in PostgreSQL/PostGIS is in pre-alpha stage as mentioned in section 3.2.3. Because of this, before implementing our functions in
that model, we had to make improvement of some basic functions defined in the SQL/MM specification [ISO07] on which the implementation of that model is based. The improvement that we have done includes the modification of some parts of functions and implementing some unimplemented functions as follows:

- The function ST_AddIsoNode has been modified to cast a value of geometry type to text type before it is passed to the function quote_literal.
- The trigger function relationtrigger fails due to comparing a character value with an integer value without type casting. The function has been modified on lines 73 and 74 to cast the character type to integer.
- The function ST_ModEdgesSplit has been modified on line 52 to cast a value of geometry type to text type before it is passed to the function quote_literal. The function was modified on line 130 to include updating absolute value which was missing. The function was modified also to change the assignment to next_right_edge from the value -anedge to oldedge.next_right_edge; the assignment was not correct on line 117.
- The function ST_GetFaceEdges is needed for our implementation but it was not implemented. Now, the function has been partly implemented; it returns the edges but further work is needed on it to return correct sequence numbers of those edges.
- The function ST_ModEdgeHeal is needed for our implementation but it was not implemented. Now, the function has been implemented.
5.3 Implementation of the transformation process

In the transformational design, a conceptual schema developed using modelling primitives undergoes a series of transformations leading to the physical implementation of the designed database. The transformation process is semi-automatic; the user can interact with it setting some parameters. The transformation process produces a text file containing statements to use to create the database tables and other related database objects. For the Contiguous-PolygonalArea primitive, the transformation produces also a text file containing statements to use to create maintenance functions. The user can model contiguous area collections representable as multipolygons using the ContiguousPolygonalArea primitive. In this case, the user represents individual component polygons and adds more descriptive information in the model to convey the semantics of ContiguousMultiPolygonalArea primitive. As a result, the transformation process will create also the file of functions.

5.3.1 Transformation from conceptual schema to logical schema

The transformation from conceptual to logical schema is implemented using the transformation capabilities available in Enterprise Architect. Enterprise Architect provides two options for performing model transformation: using transformation definitions and using add-ins [Spa07].

- The Enterprise Architect Add-In Model allows to add functionality to Enterprise Architect using components developed in an application development tool and then registered in Enterprise Architect. Application development tools that can be used to develop such components are Borland Delphi, Microsoft Visual Basic and Microsoft Visual Studio .Net for instance.

- Enterprise Architect transformation definitions enable you to transform a model using transformation templates that specify how specific model elements are transformed.

Transformation definitions provide limited capabilities for the transformation and because of this limitation, add-ins are very often used in them for advanced transformation tasks such as deleting model element. In our transformation implementation we use transformation definitions without add-ins, and advanced transformation tasks are performed using a separate transformation tool.

Enterprise Architect comes with a number of standard transformation definitions. An example of such transformation definitions is the DDL (Data Definition Language), intended for use in the transformation of database design. This transformation definition is shown in Figure 5.2.

A transformation definition consists of a number of transformation templates. The code of a selected transformation template is shown in the transformation...
Chapter 5. Implementation of modelling primitives and the transformation process

Figure 5.2: Enterprise Architect’s DDL standard transformation definition

templates editor (see figure 5.2 where the Class transformation template is selected). The transformation templates are executed in a fixed sequence (shown in Figure 5.3) during the transformation. The structure presented in Figure 5.3 shows that the transformation starts with the File transformation template that invokes the Namespace to list all model packages and invoke the Class transformation template. The invocation of templates goes on till the transformation is finished.

In our transformation implementation we have customized the DDL transformation definition. The customisation we have done include modifying the code in the transformation templates and adding new custom templates to perform the transformation according to our design and to handle our new modelling primitives.

While customising the DDL transformation definition, the Class and Attribute transformation templates have been rewritten to handle specific needs in our design. These specific needs include the transformation of data types specified by the Spatial schema (ISO 19107) and imported into Enterprise Architect project. A correspondence between the data types used in conceptual model and the data types used in implementation is given in Appendix Section A.1. Among other specific needs there is a handling of Spatial Reference System information. While building a conceptual schema, the user specifies the Spatial Reference System of spatial objects by creating a tag on the attribute having a geometry data type. The tag is given the name srid and a value of the identifier of the Spatial Reference System of the object. The code in the Attribute
5.3. Implementation of the transformation process

Figure 5.3: Hierarchy of transformation templates: (a) in PostgreSQL_Spaghetti, (b) in PostgreSQL_Topological

A transformation template retrieves this tagged value if it has been created or takes the default value of -1 otherwise. The code fragment shown in Figure 5.4 has been taken from the Attribute transformation template and shows how this transformation is done on GM_Point data type.

```
$COMMENT="Transformation of geometric data types"
$COMMENT="If a srid tag and value have been specified, copy them"
$COMMENT="Otherwise add tag srid and set it to default value of -1"
%else $typeof==="GM_POINT"
  type="POINT"
%if attTag":"srid"=true%
  Tag
  {name="srid"
    value="%gt%attTag:'srid'"%gt%}
%else%
  Tag
  {name="srid"
    value="-1"}
%endif%
```

Figure 5.4: Code fragment that transforms the GM_Point data type

Two transformation definitions have been written: PostgreSQL_Spaghetti and PostgreSQL_Topological. The transformation templates that make each of these transformation definitions and their hierarchy are illustrated in Figure 5.3. PostgreSQL_Spaghetti transforms a conceptual schema to a logical schema in which area collections are intended to be implemented in the Spaghetti model. PostgreSQL_Topological transforms a conceptual schema to a logical schema in which area collections are intended to be implemented in the topological model. These two models have been described in Section 3.2.3.
Chapter 5. Implementation of modelling primitives and the transformation process

The transformation definition *PostgreSQL* Topological* includes two custom transformation templates; Class_AreaCollection and Attribute_AreaCollection (see Figure 5.3), that handle transformation requirements specific to the topological model. One of such requirements is the creation of a new type and transformation of geometry data types to this new type.

The two transformation definitions differ in the way they transform classes representing area collections: PostgreSQL_Spaghetti transforms one area collection class into one table whereas PostgreSQL_Topological transforms the same class into four tables and one type class. The four tables are: one table to store node elements, one to store edge elements, one to store face elements, and one table to store the actual objects using references to their basic elements (nodes, edges and faces). The type class defines a new data type that is used by the objects table to store objects in terms of their basic elements.

The difference in transformation result is illustrated using an example in Figures 5.5, 5.6 and 5.7. Figure 5.5 shows a conceptual model of a spatial object Parcel using a ContiguousPolygonalArea class while Figure 5.6 shows a logical model which is the result of transformation of that class using the transformation definition PostgreSQL_Spaghetti. Figure 5.7 shows a logical model which is the result of transformation of the same class using the transformation definition PostgreSQL_Topological. The two transformation definitions (PostgreSQL_Spaghetti and PostgreSQL_Topological) have been exported to XML files. Users can import these XML files into their Enterprise Architect projects to get the transformation definitions to use.

![Figure 5.5: Example of a ContiguousPolygonalArea class](image)

![Figure 5.6: Result of transformation using PostgreSQL_Spaghetti](image)
5.3. Implementation of the transformation process

Figure 5.7: Result of transformation using PostgreSQL topological

58
5.3.2 Transformation from logical schema to physical schema

The result of the transformation from conceptual to logical schema is a model of the database representing its structure in the form of a class diagram. This class diagram is exported to XMI file, an XML-based file, that is input to the next step of transformation. Exporting a class diagram to XMI file is an option available in the modelling tool (Enterprise Architect). Enterprise Architect can export model elements to many XMI versions (1.0, 1.1, 1.2 and 2.1). However, when performing Enterprise Architect-to-Enterprise Architect transfers, the user has to ensure that the XMI version selected is 1.1 or 2.1. A brief comparison of XMI 1.1 and XMI 2.1 is given in Appendix section A.2. An example of an XMI file in XMI 1.1 format is shown in Figure 5.8.

![Figure 5.8: Example of an XMI file in XMI 1.1 format](image)

We have developed a custom tool to perform the transformation from logical to physical schema. The tool takes as input the XMI file. The current implementation accepts only XMI 1.1 format. The transformation tool performs two operations:

- Parsing the XMI file to build a data structure in which information on individual model elements can be extracted in a standard way.
- Generating SQL statements to create database objects based on the information extracted from the data structure that has been built during XMI file parsing.
5.3. Implementation of the transformation process

Parsing the XMI file

An XML parser is a computer program that reads an XML file and analyses its structure. There are Document Object Model (DOM)-based parsers and Simple API for XML (SAX)-based parsers. A comparison of these two types of parsers is given in Appendix in Table A.3. DOM is a standard recommendation of the World Wide Web Consortium (W3C) for building a tree structure in memory for an HTML or XML document [Wor98]. A DOM-based parser allows data in an XML document to be accessed and modified by manipulating the nodes in a DOM tree.

A DOM-based parser is the most suitable parser when you need to know a lot about the structure of a document, to move parts of the document around, and to use the information in the document more than once. These features being the requirements in our XMI parsing, we have selected to use a DOM-based parser in our transformation tool (MLoPhyST). The DOM-based parser that is used by our tool is the Xerces parser which was developed by Apache. This parser is also bundled with Java in its versions JDK 1.5 and above. Since we are using JDK 1.6 to develop our transformation tool we do not need a separate installation of the parser.

To prepare the document for use to generate SQL statements, following steps are taken in the tool:

- Get a document builder using document builder factory
- Parse the XMI file to create a DOM object
- Delete irrelevant nodes from the DOM tree

An XMI file contains a lot of data that are irrelevant for the transformation. These are, for instance, the data providing information on where to position specific model elements in a class diagram when the XMI file is imported into Enterprise Architect project. The nodes corresponding to these irrelevant data are deleted to free memory space for use in later processing.

A code fragment that uses the parser's library to perform the first two steps is shown in Figure 5.9. The parser can throw an exception under a number of conditions such as in case an invalid file path is passed, in case the XML file refers to a Document Type Definition (DTD) that can not be found, or in case the document is not a valid XML document. To handle such situations gracefully, we have included in our code (see Figure 5.9) a code fragment that catches those exceptions. A code fragment that deletes nodes from a DOM tree is shown in Figure 5.10. The function in this code fragment takes the parent node and the name of the node to be deleted. It then finds in the tree all the nodes for which the name is the input node name but deletes only those that have the input node as their parent.
Chapter 5. Implementation of modelling primitives and the transformation process

```java
//get the factory
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
try
{
    //Using factory get an instance of document builder
    DocumentBuilder db = dbf.newDocumentBuilder();
    statusDisplay.append("Now parsing XML file");
    //pass the document to the parser to produce the DOM tree
doc = db.parse(inputPath);
    statusDisplay.append("Parsing completed");
}
catch(ParserConfigurationException pce) {
    statusDisplay.append("String pce+toString() +
    "Transformation cancelled");
} catch(SAXException se) {
    statusDisplay.append("String se+toString() +
    "Transformation cancelled");
} catch(IOException ioe) {
    statusDisplay.append("String ioe+toString() +
    "Transformation cancelled");
}
```

Figure 5.9: A code fragment that invokes an XML parser

```java
/* a function that deletes a node from the DOM tree */
public void deleteNode(Node nd, String nodeName)
{
    try
    {
        //count the elements that have this node name
        int m = doc.getElementsByTagName(nodeName).getLength();
        //use the loop to delete all these elements if they are children of
        // the input node*/
        int p = 0; //index of the item to be deleted
        for (int k = 0; k < m; k++)
        {
            Element element = (Element) doc.getElementsByTagName(nodeName).item(p);
            // check if the parent of the current element is the input node*/
            if (element != null && nd == element.getParentNode())
                { //delete the node
                    element.getParentNode().removeChild(element);
                }
        }
    }
    catch (Exception e) {
        System.err.println(e);
        System.exit(0);
    }
}
```

Figure 5.10: A code fragment that deletes nodes from a DOM tree
5.3. Implementation of the transformation process

Generating SQL statements for PostgreSQL/PostGIS

After the DOM tree has been built and irrelevant nodes have been deleted, the transformation tool traverses the tree extracting information on model elements to use for generating SQL statements to create database objects needed to implement the model. The steps taken are:

1. Search the tree for the class nodes

2. For each class, identify the class type (table or type) based on the child node containing the stereotype of the class. Classes representing tables and types intended to be implemented in a topological model are identified by their names which include a schema name with suffix _data that must become the topology name. This is done to make sure that proper SQL statements are generated to create database objects in the topological model.

3. Extract table columns and type attributes from the class'child node that contains attributes and operations. For each attribute retrieve its properties from the attribute's child node that contains tag values. The properties meant here are the column/attribute name, column’s default value if it was specified, whether the column is part of the primary key and if so the parameters to create the sequence if any are specified. The properties also include whether the column value can be NULL, whether the column is part of a Unique key and the spatial reference system (srid) if the column is a geometry column.

4. Generate SQL statements to create sequence and table, type, and topology, depending on the type of the class and whether the class is to be implemented in the topological model. Generate statements to add a geometry column to the table and to create a spatial index if the column is a geometry column. In these statements, the information retrieved on step 3 is used to specify the properties of columns/attributes including table and object level constraints such as NOT NULL, UNIQUE and PRIMARY KEY.

5. Extract information contained in association nodes and use this information to generate statements to create Foreign Key constraints

6. If the model contains area collection classes, generate statements to create maintenance functions for area collection objects

The transformation tool, MDA Logical to Physical Schema Transformer (MLo-PhyST), has a Graphical User Interface(GUI) that allows the user to select input file (the XMI file) and to specify the location and name for the output file. It also has a text area in which it reports to the user about the transformation activity. A screenshot of this GUI is shown in Figure 5.11.
5.4 Summary of the transformational design steps

This section lists the steps taken to perform a transformational design. The description of each step is given in the transformational design user guide in Appendix B.1.

Preparatory steps

- Install the Logical to Physical schema transformation tool (MLoPhyST) if it has not yet been installed.
- Create a project in Enterprise Architect
- Import into the project spatial data types defined in ISO 19107
- Import into the project the area collection modelling primitives
- Import into the project the transformation definitions

Transformational design steps

1. Build a conceptual schema of the database
2. Transform the conceptual schema to logical schema
5.5 Verification and validation of the transformational design

The transformational design that was developed has been tested using real world data to validate it. This section describes the testing that has been done. The test example involves designing the spatial database for the provinces of Rwanda. This test example has been selected for the properties of theses provinces that they can be modelled as ContiguousPolygonalArea objects. The objective of this testing is to see how well the designed modelling primitives can be used to model real-world objects that meet their specification, how the transformation process that has been developed can help to simplify the task of spatial database design, and how the functions associated with the modelling primitives can help to maintain the database consistency.

- Conceptual design
  A conceptual schema of the provinces database was built in Enterprise Architect using the standard and the new modelling primitives. The conceptual schema is shown in Figure 5.12.

- Logical design
  The transformation definitions that were written have been used to transform the conceptual schema to logical schema. The logical schema produced by PostgreSQL_Spaghetti is shown in Figure 5.13 while the logical schema produced by PostgreSQL_Topological is shown in Figure 5.14.

- Physical design
  The custom transformation tool has been used to transform the logical schema to generate SQL statements to implement the provinces database in PostgreSQL/PostGIS. SQL statements for implementing the database in Spaghetti model are shown in Appendix section A.4.1 while SQL statements for implementation in topological model are shown in Appendix section A.4.2. A second text file containing statements to create maintenance functions was generated but the statements are not shown in this thesis.

- Populating the database and testing the manipulation functions
  Generated SQL statements have been used to create the database. The created database has been populated using the designed functions and manipulation functions have been tested. QGIS has been used to visualise the data stored in both spaghetti model and topological model as shown in Figures 5.15 and 5.16 respectively. Figure 5.16 shows that though in topological model objects are stored in terms of their topological elements (nodes, edges and faces), a database view of these objects and the individual objects stored in the spaghetti model are visually same.
Chapter 5. Implementation of modelling primitives and the transformation process

Figure 5.12: The conceptual schema of the test database

Figure 5.13: The logical schema of the test database in Spaghetti model
5.5. Verification and validation of the transformational design

Figure 5.14: The logical schema of the test database in topological model
Chapter 5. Implementation of modelling primitives and the transformation process

Figure 5.15: Visualisation of the test data stored in spaghetti model

Figure 5.16: Visualisation of the test data stored in topological model
5.5. Verification and validation of the transformational design
Chapter 6

Discussion, Conclusions and Recommendations

This chapter presents a summary of the results of this research project, discusses those results, and presents the conclusions drawn from them and the recommendations made for future improvement. Section 6.1 contains the results and their discussion. Section 6.2 contains the conclusions and finally Section 6.3 presents the recommendations.

6.1 Results and Discussion

The results are presented and discussed with respect to individual research questions specified in Section 1.2.2

Identification of feature topology realization options

The main alternatives of realizing the feature topology have been identified and presented in Section 3.2.2. These are the Spaghetti model, the Topological model and the Network model. Some advantages and disadvantages of each of these models have been described and their current implementation in two Database Management Systems (Oracle and PostgreSQL) has been presented in brief.

Our observation shows that the Spaghetti model is the mostly adopted alternative mainly due to its simplicity. This observation agrees with the literature ([The01], [KS00]). However, we argue that to select the appropriate alternative, users should not look at the simplicity only but also at other considerations such as those mentioned in Section 3.2.2.

Extending UML for modelling area feature topology

In Section 4.1, a number of topological classes have been proposed for modelling collections of area features that display topological dependencies. A modelling primitive for each topological class has been designed (see Section 4.1) using
6.1. Results and Discussion

UML extensibility mechanisms and implemented (see Section 5.2). Each modelling primitive is associated with a number of maintenance functions that are used to perform insertion, deletion and modification operations that guarantee topological robustness. The new modelling primitives have been packaged in a UML profile for use in a modelling tool such as Enterprise Architect.

The proposed topological classes do not make an exhaustive list. They have been determined based on some specific characteristics of area collections and a different classification is possible by considering other specific characteristics.

Maintenance functions have been developed for area collections representable as polygons that make up a contiguous area without holes. Maintenance functions of same purpose as those already developed are proposed for other topological classes. These counterpart functions are easier to implement based on those already developed, because they need to enforce only some of the constraints enforced by the already developed functions. For instance, a function that inserts a new DisjointArea object will be developed by modifying the function that inserts a new ContiguousPolygonalArea object. This new function enforces the non-overlapping of objects as done by the existing function but does not enforce any other constraint enforced by it. What is specific to the new function is only to enforce disjointness of the elements of the new object. Furthermore, the development of these functions in the topological model will be easier because some basic functions needed have been improved or implemented during the functional improvement of the topological model that we have made. The functional improvement that we have made in the topological model of PostgreSQL/PostGIS is described in Section 5.2.

The class of hierarchies of area collections (AreaHierarchyAssociation) needs an extended study in itself. Some of the issues to be addressed are whether we need to store the geometries of both the upper level object and the lower level object, the types of area collections valid at each level of hierarchy, and how the AreaHierarchyAssociation associates objects at two different levels of the hierarchy. Once decisions are taken on these issues, developing maintenance functions will not be difficult because this class will associate objects of other topological classes on which maintenance functions will have been developed.

Developing a transformational database design process for the realization of area feature topology

A transformational database design process has been developed for the realization of area feature topology. It consists of a conceptual design followed by a semi-automatic transformation process. Three schema levels (conceptual schema, logical schema and physical schema) have been described in Section 4.2. The three schemas result from three design levels; conceptual design, logical design and physical design respectively. Splitting the design task into three levels and performing a transformation of result from one design level to the next conforms to the principles of the Model Driven Architecture (MDA) that this transformational design follows.
The design and implementation of the transformation process have been described in Section 4.2 and 5.3 respectively. A set of rules for transforming a conceptual schema to a logical schema have been described. Two transformation definitions have been developed to perform the conceptual to logical schema transformation for two alternatives of realizing the feature topology. A tool has been developed to transform a logical schema to a physical schema.

The transformation process is semi-automatic which means that the user can interact with it. From the very beginning, the user makes a choice between the spaghetti and topological models. After the first transformation step, the user can set various parameters and hence influence the second step of transformation. The user can set or change the values of various column/attribute properties such as UNIQUE and NOT NULL constraints, the parameters of creation of sequences for primary key columns and the spatial reference system identifiers (srid) for geometry columns.

While flexibility has been one of the key considerations throughout the design and implementation parts of the research, it has not been provided during the transformation of inheritance relationship. We have described three approaches of transforming inheritance hierarchy (one table for the whole inheritance hierarchy, one table per inheritance hierarchy branch, and one table per each class in the inheritance hierarchy). However, the database designer has not been given the option to select an approach; only the transformation of each class in the hierarchy into a separate table has been provided. Aggregation, composition and association classes have not been handled in the transformation process. This implies that the current transformation implementation will transform a class which is part of an association class as a standalone class. Similarly, classes in an aggregation or a composition hierarchy will be transformed as standalone classes. If such classes contain geometry attributes, this transformation result will be an indication of wrong choice of modelling primitive: the appropriate primitive is the AreaHierarchyAssociation.

In our transformational design, a logical schema has to be exported to XMI file before it is transformed to a physical schema. Enterprise Architect can both generate and read XMI file in many versions (1.0, 1.1, 1.2 and 2.1). However, our logical to physical schema transformation tool currently accepts only XMI 1.1 format. The decision was due to compatibility issue. Unlike the latest XMI format (2.1), XMI 1.1 files exported by Enterprise Architect 7.0 (or later) are guaranteed to be correctly imported into earlier versions of Enterprise Architect. It implies that users who move their models between projects are most likely to export them to XMI 1.1. Assuming a large number of exported models to be in XMI 1.1 format, we want our tool to work on that format at least. Some implications of using XMI 1.1 instead of XMI 2.1 are described in a brief comparison of these two versions given in Appendix section A.2, but these cause no serious issues. Moreover, the modularity of the code will make it easy to provide support for other XMI formats.
6.2 Conclusions

Verification and validation of area feature topology modelling primitives and the transformation process

To verify and validate the modelling primitives, maintenance functions associated with them, and the transformation process developed, a testing has been performed with real world data. The testing has been carried out on the class of area collections representable as polygons that make up a contiguous area without holes. The testing revealed some problems in the transformation process which were then fixed. This testing has been described in Section 5.5.

Throughout the design and implementation parts of the research, flexibility and interoperability have been key considerations. For instance, flexibility is seen in providing transformations for two alternatives of realizing area feature topology, allowing the database designer to choose among the two. Also during the transformation process, the user can set some parameters and hence influence the next transformation step. The interoperability has been taken care of by making the modelling primitives available in the standard XML file format that can be imported into different projects and different modelling tools. The interoperability is also supported by the logical to physical schema transformation tool which accepts the logical schema in XMI file. The limitation is that the transformation definitions used for the first transformation step are written for Enterprise Architect. If a modelling tool other than Enterprise Architect had a way of doing similar first transformation step and could export the result to XMI file, our tool would transform the model.

6.2 Conclusions

With reference to the objectives that have driven this research project, the following conclusions are drawn:

- The main feature topology realisation options have been identified and analysed for their advantages and disadvantages. They are the Spaghetti model, the Topological model and the Network model.

- A set of primitives have been developed for modelling area feature topology. This is done by proposing a topological classification for collections of area features, then using UML extensibility mechanisms to develop a modelling primitive for each class and associating each primitive with a set of maintenance functions that guarantee topological robustness.

- A transformational database design process has been developed for the realization of area feature topology. To do this, the design task is split into three levels (conceptual, logical and physical). Next, a set of transformation rules between the results of the design levels are defined, and finally a semi-automatic transformation process that handles topological definitions carried by the transformation rules is developed.

- The transformational design process that has been developed provides two alternatives of realizing area feature topology.
The verification and validation of modelling primitive, associated maintenance functions and the transformational design process as a whole has been done. It is done by applying the software testing technique. A test case is prepared with real world data, modelling primitives are used in the transformational design and the functions are tested with the test data.

6.3 Recommendations

Based on the results of this research project, the following recommendations are made for future improvements:

- Extend the work on transformation and maintenance functions to other topological classes proposed in this research
- Develop more maintenance functions
- Extend the transformation process to handle aggregation, composition and association classes, and to give the user different inheritance transformation strategies to choose from
- Improve the logical to physical schema transformation tool to take input logical schema not only in XMI 1.1 format but also in XMI 2.1 format
- Perform also a transformation of functions
- Test maintenance functions on large data sets to evaluate their performance

Extend the work to other topological classes proposed

Most of the functions proposed for the topological class of area collections representable as polygons that make up a contiguous area without holes can have their counterparts in other topological classes. Since the class worked out is the most restrictive one, counterpart functions in other classes can be developed by starting with currently implemented functions, deciding the constraints that should not be enforced and working out the implication on topology restructuring in the topological model. The support for objects with holes can be provided in the topological model by making some modifications to the functions to make them work also on multiple faces per one object, and then providing a function to split input objects with holes into faces.

Develop more maintenance functions

This research is the first step towards a library of functions that can enhance consistency maintenance in current feature topology realization techniques while providing simple interfaces to a range of operations. In particular, such functions will shield database designers and users from the complex data structure of the topological model and hence increase its adoption in situations where it is appropriate.
6.3. Recommendations

**Extend the transformation process**

To extend the transformation process to transform association classes, the following approach may be used. In the logical schema, create a tag on the table produced from the transformation of the association part with a value of the tag `associationclass`. Create the same tag on the table produced from the class part with the value of the class id. In the logical to physical schema transformation tool, use these tag values, which have same name and value, to identify the two tables to generate statements to create only one table out of them but with the columns of both. Different inheritance transformation strategies can be implemented and the database designer can be asked to set a specific tagged value to indicate the choice on inheritance transformation strategy.

**Improve the logical to physical schema transformation tool**

The support for XMI 2.1 in the logical to physical schema transformation tool can be provided by adding a code fragment that extracts the XMI version information from the root node of the DOM tree and then writing a function to be used to perform structure-dependent operations for XMI 2.1 following the same reasoning as in structure-dependent operations for XMI 1.1.

**Perform also a transformation of functions**

It is desirable to integrate the generation of function code into the transformation process such that a modification in the conceptual function is automatically reflected in the function code generated.
# Appendix A

## A.1 Correspondence of data types between the conceptual model and the logical model

Table A.1: Non spatial data types correspondence

<table>
<thead>
<tr>
<th>Non spatial data types</th>
<th>Enterprise Architect</th>
<th>PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean types</td>
<td></td>
<td></td>
</tr>
<tr>
<td>byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Short, short</td>
<td>smallint</td>
<td></td>
</tr>
<tr>
<td>int, integer, Integer</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>Long, long</td>
<td>bigint</td>
<td></td>
</tr>
<tr>
<td>serial</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bigserial</td>
<td></td>
<td></td>
</tr>
<tr>
<td>float</td>
<td>real</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td>double precision</td>
<td></td>
</tr>
<tr>
<td>decimal, Decimal</td>
<td>decimal</td>
<td></td>
</tr>
<tr>
<td>Character types</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Char, char</td>
<td>character varying (n), varchar(n)</td>
<td></td>
</tr>
<tr>
<td>String, string</td>
<td>character(n), char(n)</td>
<td></td>
</tr>
<tr>
<td>Variant</td>
<td>text</td>
<td></td>
</tr>
<tr>
<td>Date</td>
<td>date</td>
<td></td>
</tr>
<tr>
<td>Datetime</td>
<td>timestamp[(p)] [with time zone], timestamp [(p)][without time zone]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>time[(p)] [with time zone], time [(p)] [without time zone], time</td>
<td></td>
</tr>
<tr>
<td></td>
<td>interval [(p)]</td>
<td></td>
</tr>
</tbody>
</table>

The boundary of correspondence between some data types is fuzzy. For instance, for character types the main factor to determine data types correspondence is whether the number of characters can vary or is fixed. Hence, if it can vary, the data type can be transformed into text type irrespective of which character
A.2 Comparison of XMI 1.1 and XMI 2.1

This brief comparison is based on the XMI 1.1 specification [Obj00], XMI 2.1 specification [Obj07] and the Enterprise Architect user guide [Spa07].

- XMI 1.1 has support for UML 1.3, whereas XMI 2.1 has support for UML 2.0 and UML 2.1.

Comparing currently available UML versions, the most significant changes from one version to another are in the area of behavioral diagrams; specifically, activity diagram and sequence diagrams [IBMb]. This implies that using XMI 1.1 instead of XMI 2.1 for the interchange of class diagram does not cause loss of information. The major difference between these two versions is the way the information is structured. XMI 1.1 uses a deep nesting with a lot of information contained in tagged values. The structure in XMI 2.1 is relatively flat with a lot of links between elements using elements’ identifiers. As an example of difference, XMI 1.1 has a header section while XMI 2.1 has no header section but includes information that would be in the header into the main section of the document. Another point of difference is that XMI 2.1 uses many namespaces compared to XMI 1.1.

- When exporting an Enterprise Architect package to XMI 1.1, the user can select to apply a Data Type Definition (DTD), which will write the
Appendix A.

UML_EA.DTD file to the output directory into which the XMI file is written. The XML parser uses this document to validate the correctness of the model and to check the syntax. XMI file parsing cannot be done if the DTD was applied but the DTD file is not found in the same directory as the XMI file. When exporting to XMI 2.1, no DTD file is produced.

### A.3 DOM-based parsers and SAX-based parsers

Table A.3: A comparison of DOM and SAX-based parsers

<table>
<thead>
<tr>
<th>Parser</th>
<th>Working principle</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOM parser</td>
<td>produces in memory a tree structure of the document. Functions are used to examine the contents of the tree's nodes</td>
<td>best choice when you need to know a lot about the document, to move parts of the document around or to use the information more than once</td>
<td>more CPU and memory intensive</td>
</tr>
<tr>
<td>SAX parser</td>
<td>generates events at various points in the document. A code needs to be written to handle those events</td>
<td>more CPU and memory efficient</td>
<td>suitable if the information in the document is to be used only once</td>
</tr>
</tbody>
</table>

### A.4 SQL statements generated for the test database

#### A.4.1 SQL statements for the spaghetti model

BEGIN;

-- ----Creation of sequence Number 1---------------------
DROP SEQUENCE IF EXISTS governor_governorID_seq;
CREATE SEQUENCE governor_governorID_seq
START WITH 1 INCREMENT BY 1 NO CYCLE;
-- ------Creation of sequence Number 2---------------------
DROP SEQUENCE IF EXISTS province_provinceID_seq;
CREATE SEQUENCE province_provinceID_seq
START WITH 1 INCREMENT BY 1 NO CYCLE;
-- ------Creation of sequence Number 3---------------------
DROP SEQUENCE IF EXISTS economic_activity_economic_activityID_seq;
CREATE SEQUENCE economic_activity_economic_activityID_seq
START WITH 1 INCREMENT BY 1 NO CYCLE;
-- ------Creation of table Number 1---------------------
DROP TABLE IF EXISTS governor;
CREATE TABLE governor (governorID integer NOT NULL DEFAULT nextval('governor_governorID_seq'),
    name varchar(30),
    date_of_birth date,
    qualification varchar(50),
    salary double precision,
    start_date date,
    PRIMARY KEY(governorID));
-- ----Creation of table Number 2---------------------
DROP TABLE IF EXISTS province;
CREATE TABLE province
    (provinceID integer NOT NULL DEFAULT nextval('province_provinceID_seq'),
    province_f varchar(20),
    intara varchar(20),
    province_e varchar(20),
    pop_02 integer,
    governorID integer,
    PRIMARY KEY(provinceID));

SELECT AddGeometryColumn
    ('province','geom',4326,'POLYGON',2);

CREATE INDEX province_gist ON province USING GIST(geom);
-- ----------Creation of table Number 3----------------
DROP TABLE IF EXISTS economic_activity;
CREATE TABLE economic_activity
    (economic_activityID integer NOT NULL DEFAULT nextval('economic_activity_economic_activityID_seq'),
    name varchar(50),
    population_fraction real,
    PRIMARY KEY(economic_activityID));
-- ------Creation of table Number 4-------------------
DROP TABLE IF EXISTS has;
CREATE TABLE has
    (economic_activityID integer NOT NULL,
    provinceID integer NOT NULL,
    PRIMARY KEY(economic_activityID,provinceID));
-- ------Creation of FK constraint Number 1-----------
ALTER TABLE province ADD FOREIGN KEY (governorID )
    REFERENCES governor ( governorID);
-- ------Creation of FK constraint Number 2-----------
ALTER TABLE has ADD FOREIGN KEY (provinceID )
    REFERENCES province ( provinceID);
-- ------Creation of FK constraint Number 3-----------
ALTER TABLE has ADD FOREIGN KEY (economic_activityID )
    REFERENCES economic_activity ( economic_activityID);
### A.4.2 SQL statements for the topological model

BEGIN;
-- ------Creation of topology Number 1-----------------
SELECT topology.CreateTopology ('Province_data',4326);
-- ------Creation of sequence Number 1-----------------
DROP SEQUENCE IF EXISTS
economic_activity_economic_activityID_seq;
CREATE SEQUENCE economic_activity_economic_activityID_seq
START WITH 1 INCREMENT BY 1 NO CYCLE;
-- ------Creation of sequence Number 2-----------------
DROP SEQUENCE IF EXISTS governor_governorID_seq;
CREATE SEQUENCE governor_governorID_seq
START WITH 1 INCREMENT BY 1 NO CYCLE;
-- ------Creation of sequence Number 3-----------------
DROP SEQUENCE IF EXISTS Province_provinceID_seq;
CREATE SEQUENCE Province_provinceID_seq
START WITH 1 INCREMENT BY 1 NO CYCLE;
-- ---------------Creation of table Number 1------------
DROP TABLE IF EXISTS economic_activity;
CREATE TABLE economic_activity
(economic_activityID integer NOT NULL DEFAULT
nextval('economic_activity_economic_activityID_seq'),
name varchar(50),
population_fraction real,
PRIMARY KEY(economic_activityID));
-- ---------------Creation of table Number 2-----------
DROP TABLE IF EXISTS governor;
CREATE TABLE governor
(governorID integer NOT NULL DEFAULT
nextval('governor_governorID_seq'),
name varchar(30),
date_of_birth date,
qualification varchar(50),
salary double precision,
start_date date,
PRIMARY KEY(governorID));
-- ---------------Creation of table Number 3------------
DROP TABLE IF EXISTS has;
CREATE TABLE has
(economic_activityID integer NOT NULL,
provinceID integer NOT NULL,
PRIMARY KEY(economic_activityID,provinceID));
-- ---------------Creation of table Number 4-----------
DROP TABLE IF EXISTS Province;
CREATE TABLE Province
(provinceID integer NOT NULL DEFAULT
nextval('Province_provinceID_seq'),
province_f varchar(20),
intara varchar(20),
province_e varchar(20),
pop_02 integer,
governorID integer,
A.4. SQL statements generated for the test database

PRIMARY KEY(provinceID));

SELECT topology.AddTopoGeometryColumn('Province_data',
CAST(current_schema() AS text),'Province','geom','POLYGON');

-- -------Creation of FK constraint Number 1-------------
ALTER TABLE has ADD FOREIGN KEY (economic_activityID )
REFERENCES economic_activity ( economic_activityID);

-- -------Creation of FK constraint Number 2-------------
ALTER TABLE Province ADD FOREIGN KEY (governorID )
REFERENCES governor ( governorID);

-- -------Creation of FK constraint Number 3-------------
ALTER TABLE has ADD FOREIGN KEY (provinceID )
REFERENCES Province ( provinceID);

END;
Appendix B

B.1 Transformational design user guide

This sections describes the steps taken to perform a transformational design of spatial database using the transformation process and the modelling primitives developed in this research.

Preparatory steps

- Install the Logical to Physical schema transformation tool (MLoPhyST)
  If you have not yet installed the Logical to Physical schema transformation tool, install it now. The tool is installed once and you can use it until you uninstall it.

- Create a project in Enterprise Architect

- Import into the project Spatial data types defined in ISO 19107
  Select the root node in the Project browser then go to Project menu: Project > Import/Export > Import Package from XMI. Select the file ISO19107Package.xml and import it.

- Import into the project the area collection modelling primitives
  Switch from the Project browser to Resources, right-click on UML Profiles and select Import Profile. Select the file areaCollections.xml and import it.

- Import into the project the transformation definitions
  Go to Tools menu. Tools > Import Reference Data, then select the file PostgreSQL_Spaghetti.xml and import it. Repeat this step to import the file PostgreSQL_Topological.xml.

Transformational design steps

1. Build a conceptual schema of the database
   Open the project you created and build your conceptual schema. Area collections modelling primitives are available in the toolbox by selecting More tools > AreaCollections. To specify the Spatial Reference System on a class, select the attribute with a geometry data type and create a tagged value with the name srid and the Spatial Reference System as the
value. To set a UNIQUE constraint on a future column, select the static property on the corresponding attribute from the attribute’s properties.

2. Transform the conceptual schema to logical schema
   In the Project browser, select the package containing the class diagram to transform. Go to Project menu: Project > Transformations > Transform Current Package. In the list of transformations, select the transformation definition to use and the package to contain the transformation output. If you want to transform the schema to logical schema in which area collections are to be implemented in Spaghetti model, select the PostgreSQL_Spaghetti transformation definition. For implementation in topological model, select the PostgreSQL_Topological transformation definition. Select click Do Transform to run the transformation. After the transformation, you can open the output logical schema and modify some parameters if you need to. To see the parameters that can be modified, select the table column of interest and check the available tagged values. For instance, you can disable the creation of a sequence number for a primary key column by selecting the primary key column and modifying the tagged value setting AutoNum=1 to AutoNum=0.

3. Export the logical schema to XMI file
   From the Project browser, select the package containing the logical schema. Make sure that if you have nested packages, you select the inner package. Go to Project menu: Project > Import/Export > Export Package to XMI. Select the location and name for the output file. In the area For Export to Other Tools, select UML 1.3 (XMI 1.1) for the field XMI Type. Click Export to export the package.

4. Transform the logical schema to physical schema
   Launch the Logical-to-Physical schema transformation tool. For the field Input file, select the XML file produced in step 3. Select the location and name for the output file. Click Transform to run the transformation.

Post design steps

1. Database creation
   The result of the transformational design is a text file (.sql) containing the statements for creating the database objects in PostgreSQL/PostGIS. If the model contains area collections, another text file (.sql) will be created. The statements in this second file will be executed to create maintenance functions. At this step, if the design contains area collections intended to be implemented in Spaghetti model or contains no area collections, you can execute the statements in the output file to create database objects. If the design contains area collections intended to be implemented in topological model, you need first to make sure that the topological model has been enabled in your PostgreSQL/PostGIS installation.

2. Database population
   After the database objects have been created, the database can be popu-
lated. Standard methods (insertion of objects one by one or bulk loading) can be used, however to preserve topological relationships of *area collection* objects you use the functions created as part of this implementation. Functions to be used in the topological model are located in the *Topology* schema whereas functions to be used in Spaghetti model are located in the *Public* schema.

3. Data manipulation

Likewise, you use the functions created as part of this implementation to manipulate *area collections* objects without violating topological relationships. Currently, manipulation functions available are those to delete objects from a contiguous area and those to merge objects in a contiguous area.
B.1. Transformational design user guide
Bibliography


Bibliography


