UNIVERSITY OF OSLO
Department of informatics

GeoXSLT
GML processing with XSLT and spatial extensions

Master thesis
60 credits

Carl Fredrik Melsom Klausen

08. 11 2006

Abstract

This thesis claim that XSL Transformations combined with extensions can be used to process geodata encoded as GML. The assertion is backed up by the following deliverables:

• A working proof-of-concept for an XSLT based transformation of spatial data.

• Tests providing measurements of functionality and performance.

• Argumentation that shows how and why this is a viable approach by discussion and practical examples.

The paper concludes with a confirmation on the feasibility of the approach inline with the research objectives and findings provided by the deliverables.

Acknowledgements

This thesis is submitted to the Department of Informatics at the University of Oslo as part of the Master Degree.

There is a long list of people that I owe a lot of thanks for their support.

First, I wish to thank my thesis supervisor David Skogan at SINTEF Information and Communication Technology, Oslo. He has shown a patience and interest in my studies far beyond what could be expected. His guidance and inspiration has helped me time and again.

I am grateful to Anne Lund and Stine Holm at SINTEF for their help with practical matters during the course of the thesis. Especially the support they showed when my laptop got stolen 5 days before the deadline meant a lot to me.

I would also like to thank Knut Staring at the Department of Informatics, University of Oslo, for introducing me to GIS systems in the first place.

A thank you also goes to my employer Tinde AS which gave me the necessary time off and flexibility to work with my thesis while being in a fulltime job.

Finally I would like to thank my wonderful live-in girlfriend, Karine for her everlasting patience and support during the late nights and weekends working with the thesis.

Contents

List of figures

List of tables

1 Introduction

In this thesis I claim that XSL Transformations combined with extensions can be used to process geodata encoded as GML. The assertion is backed up by the following deliverables:

• A working proof-of-concept for an XSLT based transformation of spatial data.

• Tests providing measurements of functionality and performance.

• Argumentation that shows how and why this is a viable approach by discussion and practical examples.

The paper concludes with a confirmation on the feasibility of the approach inline with the research objectives and findings provided by the deliverables.

1.1 Research objectives

Through research and prototyping, the following research questions are under focus.

1. Can the XSLT language in combination with extensions be used for processing geodata?
2. What limitations on factors such as performance, flexibility, and scalability will this approach imply?
3. How will the use of this approach have an effect on code readability, ease of use, and the development of geospatial applications?

1.2 Motivation

We see an increasing trend of using web services and XML throughout the Internet. The standardization of workflows and data formats as facilitated by the use of this approach provides possibilities for the development and integration of different systems through open standards for communication and functionality. The GIS community embraces the use of web services in the transport and manipulation of geodata with the Web Feature Server (WFS) and Web Map Server (WMS) specifications. Data encoded as Geography Markup Language (GML) can be downloaded on demand from various sources. However, it may be necessary to do relatively complicated work to integrate the data or to perform generic spatial operations. Unfortunately, there is no guarantee that the necessary spatial operations are available on all platforms. Often there is a need to use external GIS tools to do the spatial operations manually. In other, non-spatial settings, the transformation and formatting of XML based data is made easier by the use of XSL Transformations (XSLT). XSLT is a declarative XML based template language dedicated to the integration and manipulation of XML based data. The motivation for this thesis is the simplification and standardization gained by applying the XSLT model of processing to geodata by integrating spatial capabilities into the language. The main advantages of this approach are:

- The relative simplicity of XML processing with XSLT compared to generic functional and object-oriented languages.
- Platform independence
- Changes to the templates can be performed without recompiling or altering the whole application.
- Standardized interface to spatial operations

By applying XSLT to the GML transformation process, an approach already used in many other systems based on web services can be used to simplify the work with spatial data. It is believed that easy access to spatial functionality directly available from within XSL templates will lower the threshold for using geodata in existing and new applications and has a potential for incurring a more widespread use.

1.3 Method

In the article "Software Engineering Research versus Software Development" (Marcos 2005) Esperanza Marcos points out how engineering research differs from other research in that its aims are to find out how to do things or to create new objects. This thesis overlaps with both Marcos' point of view and the more traditional research in that it proposes to create new objects and methods to implement an existing model of technology on a different field from what it is commonly used for today. Hence, the method chosen for this approach both has to provide a framework for the creation of new concepts, objects and methods along with the evaluation of existing practice. In regard of the traditional research methods, this thesis will use qualitative methods of research to evaluate current programming practices and experiences of GIS systems by the analysis of previous studies, code and architectures. Qualitative methods are used to evaluate certain parameters of success, such as the simplification of code, usability of framework, flexibility, and the degree of separation of concerns achieved. Quantitative empirical methods are used for the evaluation of other parameters of success, such as the evaluation of performance, scalability and standards conformance. Marcos introduces the term creative research methods as methods for those sciences that "...require a high level of creativity as opposed to observation or experimentation" and "These methods are based on such characteristics as imagination, premonition, visualization and the like...". By acknowledge of the creative component in the architectural design process for the experimental prototype used in this thesis, Marco's parallels between software development and the scientific method used in Software Engineering (SE) design are made available. The SE method is based on Bunge's general model as presented in Scientific Research (Bunge 1967) Basing the prototype development on a generally sound scientific model is done to bring more quality to the process and to assure results available for evaluation. Based on this reasoning, the following steps in the approach have been identified:

- Identification of research objectives.
- Definition of criteria for success
- Prototyping/implementation
- Evaluation of implementation
- Discussion of findings in view of previous studies
- Conclusion

Each step is intended to be executed in iteration with the steps before and after to include the knowledge accumulated through each step. A continuous study of related studies and material is performed along each step.

1.4 Expected research contributions

The planned results of this thesis are:

- A proof of concept for an XSLT based transformation of spatial data in Geographic Markup Language (GML)
- A set of requirements for the development of spatial extensions to

(E)XSLT

• Measurements and evaluations of performance

• A conclusion on the viability of the idea and ideas for future work.

1.5 Limitations

This thesis does not aim to produce a fully working processing environment for GML, but to evaluate the feasibility of the approach through a combination of theoretical reasoning and partial practical experimentation. Hence, only a subset of the proposed features will be implemented. Due to the readily available resources for the processing of geodata on the Java platform in Java Topology Suite (JTS) and Geotools, the implementation will be done in Java.

1.6 Structure of thesis

Theoretical foundations

This thesis' theoretical foundations are organized as three sections. Section XML Schema presents technology related to the implementation of the GeoXSLT framework and examples. Section 2.2.1 gives a walk-through of a geospatial workflow and shows how spatial XSLT can be of use. Section 2.3 outlines a series of previous studies of relevance to the research done.

Supported functionality

Chapter 3 presents the functionality to be supported and explains the motivation and relation to community standards defined by the Open Geospatial Consortium.

Implementation of core functionality

Chapter 4 gives a detailed explanation of the motivations and design of the implemented architecture in the GeoXSLT framework.

Testing

Chapter 5 introduces the tests designed to validate correct functioning and technical and practical performance.

Findings

Chapter 6 presents the findings from execution of the tests defined in Chapter 5.

Discussion

Chapter 7 discusses the findings and experience from the work with the GeoXSLT framework in view of the research questions and previous research.

Conclusion

Chapter 8 concludes with the viability of spatial XSLT, lists the major contributions made, and outlines future work.

2 Theoretical foundations

This chapter will start off by giving a walkthrough of related technologies in 2.1. In section 2.2 a description of a scenario where geospatial data is processed is used to show the motivation for why spatial extensions for XSLT are needed. The chapter is rounded off with a rundown of previous research in 2.3and a chapter summary in 2.4.

2.1 Related technologies

This section presents the technologies and concepts involved in XSL based template processing of GML

2.1.1 XML Schema

The XML format can be adapted to many different uses by defining rules for its composition and data allowed. XML Schema is a language for creating such rules (Biron, Permanente et al. 2004; Fallside and Walmsley 2004; Thompson, Beech et al. 2004). The Open Geospatial Consortium (OGC) has used the Schema language to define the format of the Geography Markup Language (GML), a format tailored for encoding geographic data (OpenGeoSpatial 2002). A complete primer on XML Schema is provided in (Fallside and Walmsley 2004).

2.1.2 Geography Markup Language

This section will provide a quick background on what GML 2.1.2 is and explain the core features. Finally, a short explanation on the future of GML as defined in GML 3.0 is given.

The Geography Markup Language (GML) is an XML format for representing entities in the real world, such as trees, buildings, and roads. Entities are represented as *features* that can describe both geometric and non-geometric properties. As an example, a building can have features representing the location (geometric) and the building-type (non-geometric). GML is designed to support the encoding of both types of features, where non-geometric features can be associated through integration with other XML schemas. Table 2.1 below lists the GML representation of a building. The location is represented as a geometric *point* feature, while type, status, number and other properties are represented as non-geometric features.

```
<gml:featureMember>
 <topp:bulroad fid="bulroad.2545">
  <topp:the_geom>
   <gml:Point srsName="http://www.opengis.net/gml/srs/epsg.xml#32633">
    <gml:coordinates xmlns:gml="http://www.opengis.net/gml" decimal="." cs="," ts=" ">
      357080,7766653
    </gml:coordinates>
   </gml:Point>
   </topp:the_geom>
   <topp:type>Outhouse</topp:type>
   <topp:status>2</topp:status>
   <topp:number>192574250</topp:number>
   <topp:started>10101</topp:started>
   <topp:updated>19940210</topp:updated>
  </topp:bulroad>
</gml:featureMember>
```

Table 2.1 Example of an entity represented by both geometric and non-geometric features

Structure

The structure of a GML document is very flexible. Generally, it consists of a series of *Features* representing the real-world entities. The features are children of a *FeatureCollection* which hence works as a container. One of the things that make the GML format so flexible is that each Feature also is a FeatureCollection. In this way, an entity can be represented by aggregations of other features. As an example, one can think of a park with trees, green areas, water, and roads. While each of these is an independent entity represented as a feature, the park can be defined as Feature/FeatureCollection consisting of all the trees, roads etc. GML also has support for defining other relations between different features through the use of XLinks.

Geometric features supported

The GML 2.1.2 schemas provide a method of encoding what the Open Geospatial Consortium (OGC) defines as *simple features* (Ryden 2005). With simple, OGC means "…features whose geometric properties are restricted to 'simple' geometries for which coordinates are defined in two dimensions and the delineation of a curve is subject to linear interpolation" (Cox, Cuthbert et al. 2002). In short, this means that GML 2.1.2 mainly focuses on representing geometric features in two dimensions. The following is a list of the OGC simple geometry classes:

- Point

- LineString
- LinearRing
- Polygon
- MultiPoint
- MultiLineString
- MultiPolygon
- MultiGeometry

Based on sampled data, we provide examples of the first four geometries below while the complete schema definitions for all the geometries can be found in (OpenGeoSpatial 2002). Multi geometries are simply feature collections consisting of one to many basic geometric features.

```
<gml:Point srsName="http://www.opengis.net/gml/srs/epsg.xml#32633">
  <gml:coordinates xmlns:gml="http://www.opengis.net/gml" decimal="." cs="," ts=" ">
    357080,7766653
  </gml:coordinates>
</gml:Point>
```

Table 2.2 Example of Point geometry encoded as GML

```
<gml:LineString>
   <gml:coordinates xmlns:gml="http://www.opengis.net/gml" decimal="." cs="," ts=" ">
     357015,7766698    357127,7766654    357205,7766613    357286,7766585    357364,7766577
357389,7766583 357406,7766595 357488,7766710 357498,7766735 357502,7766771
   </gml:coordinates>
</gml:LineString>
```

Table 2.3 Example of LineString geometry encoded as GML

```
<gml:Polygon>
          <gml:outerBoundaryIs>
                    <gml:LinearRing>
                              <gml:coordinates decimal="." cs="," ts=" ">
                                 80,340 160,340 160,280 80,280 80,340
                              </gml:coordinates>
                    </gml:LinearRing>
          </gml:outerBoundaryIs>
          <gml:innerBoundaryIs>
                    <gml:LinearRing>
                              <gml:coordinates decimal="." cs="," ts=" ">
                                 100,330 130,330 130,290 100,330 90,290 130,290 130,290 100,330
                             </gml:coordinates>
                    </gml:LinearRing>
                    <gml:LinearRing>
                              <gml:coordinates decimal="." cs="," ts=" ">
                                 150,335 150,320 140,335 150,335
                              </gml:coordinates>
                    </gml:LinearRing>
          </gml:innerBoundaryIs>
</gml:Polygon>
```

Table 2.4 Example of Polygon consisting of LinearRing(s) encoded as GML

GML 3.0

According to (Lake 2004) GML 3.0 is almost entirely backwards compatible with GML 2.1.2. The main difference with the two versions is that GML 3.0 has a larger feature set with support for more geometries, dynamic features, three-dimensional objects, and default styling to mention some. This thesis has focused on GML 2.1.2 due to its simpler nature and more widespread use. The backward compatibility of GML 3.0 indicates that the prototype developed at least should work on a subset of the geometric features defined without any changes.

2.1.3 Web Feature Service

This section describes Web Feature Services in terms of definitions and interaction workflow.

Web services are a collection of standardized protocols and methods for communication between applications across the Internet over HTTP. (Cabrera, Kurt et al. 2004). With the Web Feature Service (WFS) specification (Vretanos 2005) OGC has defined web service interfaces for access and manipulation of geographic data. The operations available are (Vretanos 2005):

- Create feature
- Delete feature
- Update feature
- Lock feature
- Get and query features based on geometric and non-geometric constraints.

GML is used as the language for encoding queries and results of the transactions performed. The use of GML for “transport” enables WFS servers to provide an abstraction level independent of the internal data sources. This makes data stored in formats such as Shape, SOSI or other proprietary or country specific standards available as GML over the WFS interfaces. This allows users to “..combine, use and manage geodata – the feature information behind a map image – from different sources..” (Vretanos 2005).

WFS Interaction workflow

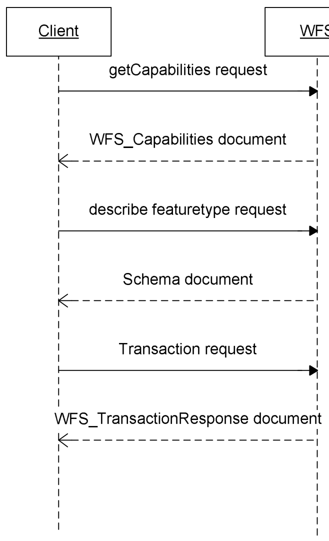

Client
WFS
getCapabilities request
WFS_Capabilities document
describe featuretype request
Schema document
Transaction request
WFS_TransactionResponse document

Figure 2.1 Workflow between client and WFS server (Vretanos 2005)

Figure 2.1 describes the flow in a WFS interaction. First, the client issues a “getCapabilities” request, which result in a WFS_Capabilities document containing the features available on the server. Based on this, the client can issue a request for the description of a given feature type. To this the WFS returns an XML schema containing descriptions of the feature. The client can now issue operations on the feature(s) according to operations available on the WFS. Section 2.2.2 goes into more detail around WFS interaction with a walkthrough of a WFS filter query.

The above has described WFS as a set of interfaces for querying and manipulating geospatial data. It should also be mentioned that OGC has defined a corresponding Web Map Server (WMS) specification, which deals with the visualization and presentation of the geospatial data (Beaujardiere 2006).

2.1.4 XPath

XPath is a language for identifying parts of XML documents, designed to be used in XSLT and XPointer (Clark and DeRose 1999; DeRose, Jr. et al. 2002). In XSLT the language is used to match and select particular parts of the source tree for copying into the result document or further processing by the template rules (Harold and Means 2002).

XPath expressions can be grouped into location paths, general expressions, and functions.

Location paths

Location paths are powerful tools to identify a set of nodes in a document and consist of a series of *location steps* where each step is separated by a "/" char. There are two distinct types of location paths, relative and absolute. The two types are distinguished on how the series of location steps is specified.

Location steps

A location step is always a selection of nodes relative to the *context node (Clark and DeRose 1999).* For absolute location paths the first location step uses the source tree's root node as context node (Bray, Paoli et al. 2006). Relative paths consist of location steps that start with the node currently being processed as *context node.*

For each location step the identified set of nodes are used as the context node for the next step. Absolute location paths are also called *root location paths* (Harold and Means 2002).

```
/rootelement/feature
```

Table 2.5 Example of absolute location path

```
Feature
```

Table 2.6 Example of relative location path

The examples in Table 2.5 and Table 2.6 are location paths that apply to the XML document listed in Table 2.9. The relative path in Table 2.6 would only return nodes if called from within a "rootelement" context.

In addition to the node selection, each step contains an axis and an optional predicate test expressed with *general expressions*(Harold and Means 2002).

The axis specifies the direction of which to perform the node selection and can be along one of a comprehensive set of axes as defined in (Clark and DeRose 1999). Only a small subset is presented below:

- *Ancestor* All nodes that are parents of the context node.
- *Preceding-sibling* All nodes that precede the context node and share the same parent in reverse document order (Harold and Means 2002)
- *Descendant* All descendants of the context node, but not the context node itself (Harold and Means 2002)

Examples of various different location paths can be found throughout the thesis.

XPath functions

XPath provides several functions that can be used separately or as part of general/predicate expressions in location paths. The functions can be grouped by the data types they operate on:

Boolean	Number	Node-set	String
Boolean()	Ceiling()	Count()	Concat()
False()	Floor()	Id()	Contains()
Lang()	Number()	Last()	Normalize-space()
Not()	Round()	Local-name()	Starts-with()
True()	Sum()	Name()	String()
		Namespace-uri()	String-length()
		Position()	Substring()
			Substring-after()
			Substring-before()
			Translage()

Table 2.7 XPath functions grouped by data types (Clark 1999; Holzner 2001)

The functionality provided by the XPath functions makes working with XPath more efficient. As an example, the “count()” function makes it possible to bring the size of a given node-set into a predicate expression. Detailed explanations and examples of all standard functions can be found in (Holzner 2001; Harold and Means 2002).

2.1.5 Extensible Stylesheet Language Transformations (XSLT)

This section gives a presentation of the XML based XSLT language with a focus on the areas most relevant to this thesis.

XSLT is a functional programming language for the specification of how one XML document should be converted into another document. Though common, the output document does not necessarily need to be an XML document.

Basic workflow

The transformation rules specified for how a document should be converted are enclosed within *template rules* (Clark 1999) and saved in a *stylesheet*. Each template has a “match” attribute. This attribute contains a pattern that identifies the source node or nodes to which the rule applies (Clark 1999). The pattern is defined with a language called XPath which is described further down. With a set of template rules saved in a stylesheet and an XML source to apply the rules on, an XSLT Processor is needed to perform the actual transformation.

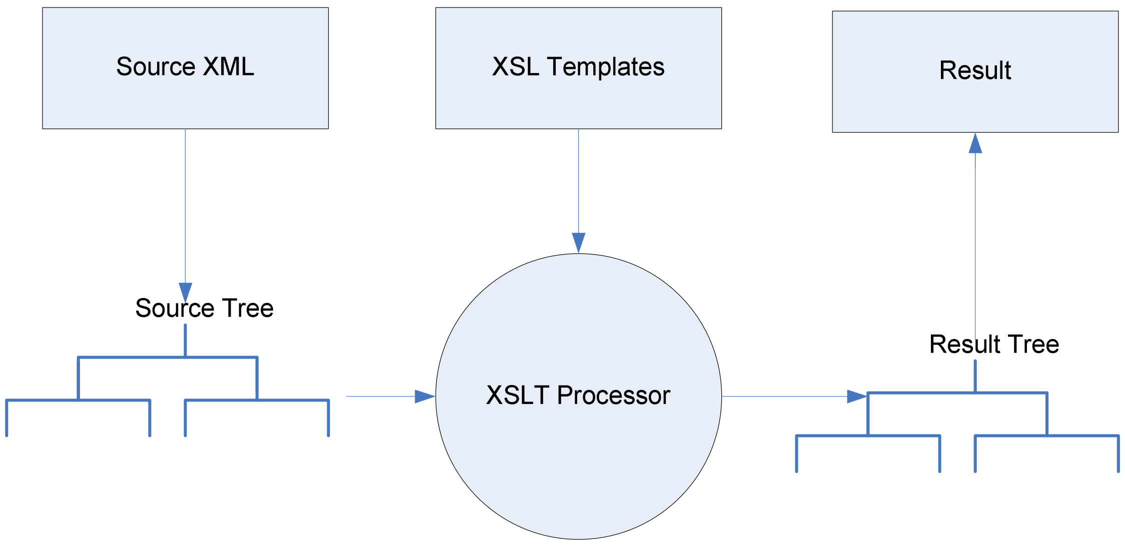


Figure 2.2 The conceptual workflow of an XSLT transformation.

The XSLT processor works by parsing both the source and stylesheet into separate tree structures. The source tree is then searched for nodes that fit some template's match pattern. Templates matched are then executed. Table 2.8 below lists an example of a very simple XSLT stylesheet used to process the XML in Table 2.9. The transformation result is available in Table 2.10.

```
<?xml version="1.0" encoding="UTF-8"?>
<!—ROOT element, defining language version, namespaces and prefixes →
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

        <!—TOPLEVEL element defining output type and encoding of genereated result-->
         <xsl:output method="xml" version="1.0" encoding="UTF-8" indent="yes"/>

         <!—Template rule with an intuitive match attribute -->
         <xsl:template match="/rootelement/feature">
                  <anotherRootElement><!— Element created by this template -->
                    <anotherfeature><!— Element created by this template -->

                    <!— Instructing the processor to output the text-value of the current node.
                          As this template rule only matches the "feature" node of the "rootelement",
                          this value will be the text-value of the feature node -->
                          <xsl:value-of select="."/>

                     </anotherfeature>
                  </anotherRootElement>
         </xsl:template>
</xsl:stylesheet>
```

Table 2.8 Simple stylesheet with explanations

The basic stylesheet in Table 2.8 defines one template rule that matches the "feature" child of the "rootelement" node. The remaining flow is documented in the example.

```
<?xml version="1.0" encoding="UTF-8"?>
<rootelement>
	<feature>Featurevalue</feature>
</rootelement>
```

Table 2.9 Simple XML document

```
<?xml version="1.0" encoding="UTF-8"?>
<anotherRootElement>
	<anotherfeature>Featurevalue</anotherfeature>
</anotherRootElement>
```

Table 2.10 Transformation result

Only the very basic workflow and principles of XSLT has been presented here. A complete listing of elements and features is available in (Clark 1999; Harold and Means 2002). A comprehensive introduction to XSLT is available in (Holzner 2001). The thesis does also contain a set of template examples of varying complexity.

XSLT Elements

XSLT defines 37 elements, which can be organized in three overlapping categories; root, top-level, and instruction elements. (Harold and Means 2002). Detailed explanations of the categories and elements can be found in (Clark 1999; Harold and Means 2002). This section will present one useful instruction element, the "xsl:element". A common use of XSLT is to generate new XML documents based on merging and formatting a combination of other sources. When generating new documents, there will often be a need to create new XML elements as well. While XSLT supports the use of literal result elements, using the "xsl:element" allows us to determine the new element's name at runtime.

XSLT Functions

While the XPath operations are focused on the nodes and values of those, XSLT provides additional functions with a more general application. Examples of commonly used functions are the "document()" and "current()" functions. The "document()" function allows loading external XML documents during processing. The "current()" function is of practical use in loops etc, where it represents the current node being processed. (As opposed to the context node in location paths which is relative to the location path).

Operation				
Current()	Document()	Element-available()	Format-number()	Function-available()
Generate-id()	Key()	System-property()	Unparsed-entity()	

Table 2.11 Standard XSLT Functions (Holzner 2001)

2.1.6 EXSLT

XSLT defines two mechanisms for extending the behavior provided by the processor (Clark 1999; Leung 2004). XSLT Templates may contain EXSLT functions and elements. These mechanisms allow developers to provide functionality implemented in some other language, e.g. Java, to be accessed from the templates during transformation runtime.

EXSLT Functions

EXSLT Functions provide access to the external functionality in a pattern similar to that of XSLT/XPath functions. As such, EXSLT functions can be used as part of location paths/predicates and to return generated node sets to mention some. The XSLT standard does not define any specific EXSLT functions, but a community effort to standardize common functionality is available at (EXSLT.ORG 2006).

EXSLT Elements

EXSLT elements generally provide more flexibility than what is possible with EXSLT functions. One reason for this is that the external language has access to more information about the parameters and context passed than what is available for EXSLT functions. Still, using extension elements in the templates is more cumbersome and complicated than with extension functions. The familiarity and integration of extension functions with XPath/XSLT functions is also something that should be considered.

2.2 Processing geographic data

As the introduction described the basic motivation for researching the use of EXSLT with geo data represented as GML, this section will give a top-level view of the current workflow and introduce the challenges of processing GML in a case based setting.

2.2.1 Sources of geographic data

Geographic data has been collected for a long time and is today represented on a host of formats and in many different systems without a common standard. Some systems are open, others closed. Data is also represented with different resolution levels, which makes it very difficult to integrate data from different sources. There are quite a few challenges related to this diversity, and both previous and ongoing research is working to address the problems with accessing geographic data across different sources, formats, and resolutions. The Open Geospatial Community (OGC) represents one such effort, and has addressed the need for a common standard for the encoding, transmission and query of geospatial data through the development of specifications. For the representation of geographic data, the Geographic Markup Language (GML) has been developed. This allows for the encoding of both geographic features as well as other information related to the objects. GML (as of version 3.0) is very flexible and allows for custom tailoring of the format at both domain and application levels without breaking the standard. Section XX will go into further detail about GML and its features. For the transmission and query of spatial data, the Web Feature Service (WFS) has been developed. This specification presents a standard interface for querying and transmitting data encoded as GML over the Internet by using the HTTP protocol and dedicated WFS servers following a web services pattern.

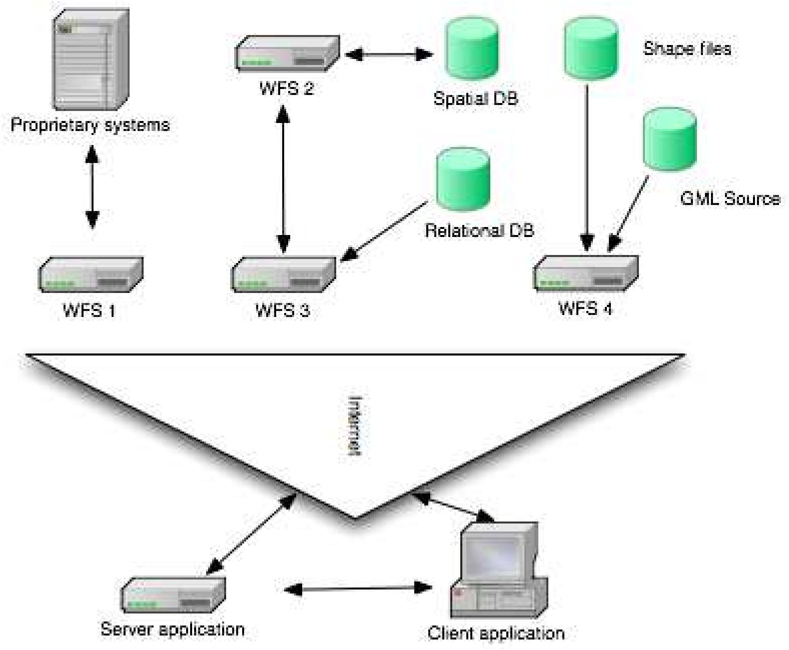


Figure 2.3 Accessing geographic data through WFS

Figure 2.3 illustrates how servers implementing the WFS standard make many sorts of spatial data available through common interfaces and connections. Output is encoded as GML. Clients and other applications can find available servers with relevant data by querying a Web Registry Service (WRS)/OGC Catalog Service, which describes available service offers (Lake 2004). The client/application can then through the standard interface both execute operations and download data as defined in the WFS standard directly from the WFS server. While there are many possibilities in the various interactions available with WFS, this thesis focuses on operations performed on the data received.

2.2.2 Accessing WFS

To give a better explanation of WFS, this section provides a walk-through of a WFS query. Geoserver, the WFS implementation used is an open source OGC compliant WFS/WMS server (Owens 2006). Geoserver uses the publicly available TIGER dataset (U.S. Census Bureau 2005) as test data and this is also used for the example. The dataset provides a broad array of real-life data from the U.S. and makes it easier to keep in sync with practical applications of researched concepts and ideas.

In a WFS request, the mandatory initial task is to define the area of focus and which features inside the area that are of interest. When posting these data to the server, it will do a search based on the request and return the data encoded as GML.

Table 2.12 Initial WFS query to be posted

```
<wfs:GetFeature service="WFS" version="1.0.0"
outputFormat="GML2"
xmlns:topp="http://www.openplans.org/topp"
xmlns:wfs="http://www.opengis.net/wfs"
xmlns:ogc="http://www.opengis.net/ogc"
xmlns:gml="http://www.opengis.net/gml"
xmlns:xsi="
http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.opengis.net/wfs
http://schemas.opengis.net/wfs/1.0.0/WFS-basic.xsd">
<wfs:Query typeName="tiger:tiger_roads">
 <ogc:Filter>
  <ogc:BBOX>
   <ogc:PropertyName>the_geom</ogc:PropertyName>
  <gml:Box srsName="http://www.opengis.net/gml/srs/epsg.xml#4326">
    <gml:coordinates>
         -73.966331198449,40.78195219458531 -73.96139758516432,40.78442509210824
    </gml:coordinates>
   </gml:Box>
  </ogc:BBOX>
 </ogc:Filter>
```

```
</wfs:Query>
```

Table 2.12 is an example of a WFS-request posted to Geoserver to get all the roads in a given area from the tiger data-set. A visualization of the area is provided in Figure 2.4

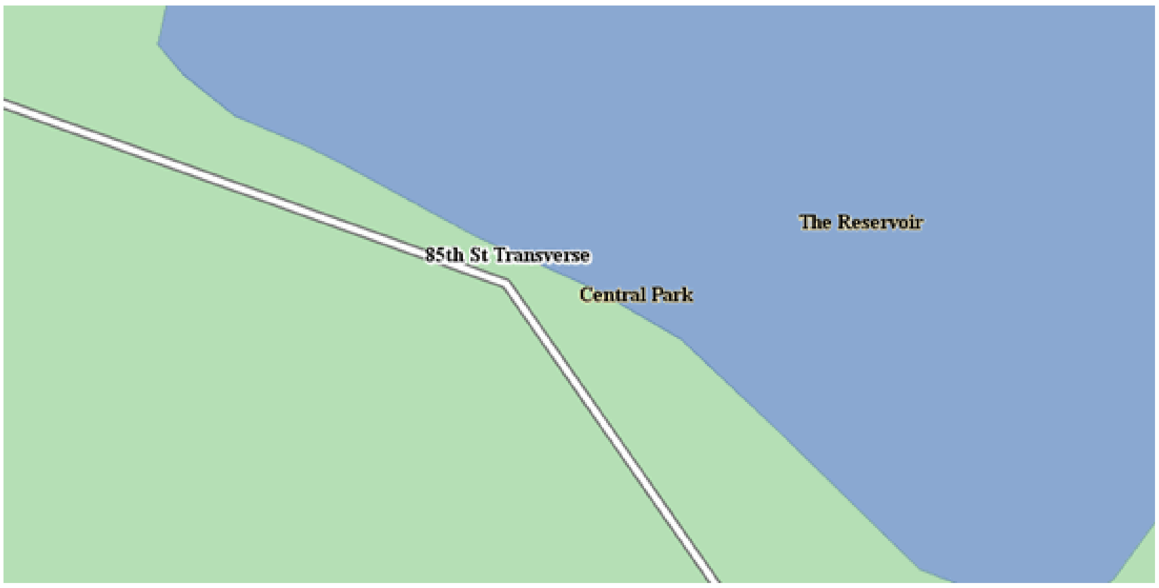


Figure 2.4 Visualization of data returned from the query in Table 2.12

Figure 2.4 visualizes the result of the query in Table 2.12. The returned data is a representation of the 85th St Transverse along with a rendering of the lake to provide some context.

Table 2.13 GML Return from Geoserver WFS query

```
<wfs:FeatureCollection
xsi:schemaLocation="http://www.census.gov
http://localhost:8080/geoserver/wfs/DescribeFeatureType?
typeName=tiger:tiger_roads
http://www.opengis.net/wfs http://localhost:8080
/geoserver/schemas/wfs/1.0.0/WFS-basic.xsd">
<gml:boundedBy>
<gml:Box
srsName="http://www.opengis.net/gml/srs/epsg.xml#4326">
<gml:coordinates decimal="." cs="," ts=" ">
-73.96738,40.781319 -73.962847,40.78438
</gml:coordinates>
</gml:Box>
</gml:boundedBy>
<gml:featureMember>
```

```
<tiger:tiger_roads fid="tiger_roads.7752">
<tiger:the_geom>
<gml:MultiLineString
srsName="http://www.opengis.net/gml/srs/epsg.xml#4326">
<gml:lineStringMember>
<gml:LineString>
<gml:coordinates decimal="." cs="," ts=" ">
-73.96738,40.78438
-73.964179,40.783234
-73.962847,40.781319
</gml:coordinates>
</gml:LineString>
</gml:lineStringMember>
</gml:MultiLineString>
</tiger:the_geom>
<tiger:CFCC>A41</tiger:CFCC>
<tiger:NAME>85th St Transverse</tiger:NAME>
</tiger:tiger_roads>
</gml:featureMember>
</wfs:FeatureCollection>
```

As is shown in the previous visualization and the GML listed in Table 2.13, the correct road is returned along with the available non-geographic properties. In the case of this specific road it amounts to a code (A41) and the name of the road (85th St Transverse).

2.2.3 Processing WFS results

Assuming the user wants to create a buffer around the received data to use in a new WFS query, there are mainly two approaches. He can either use a DOM/SAX parser to represent the data received for direct programmatic manipulation, or it can use the template based XSLT approach. Neither of the two have any native support for dealing with spatial data. This makes it difficult and complicated to do any operations on or with the geographic data. As an example, say that the previous result from the WFS server returned two roads, and that the user wants to know if and where they cross each-other. Performing this conceivably simple act would take quite a lot of effort, and what if the user needed to do more complicated operations, such as generalization or distance calculations? One could try combinations of different queries based on the returned data from the

WFS, but this might again be made difficult by such simple things as using multiple data sources and still having to deal with the details in the data format. There is a need for utilities to do operations on downloaded geographic data inside an XSLT context. An interactive approach has several good toolkits, such as Udig (Refractions Research 2006) and JUMP (JUMP 2006), but these are mostly used as standalone desktop applications. For the programmatic approach, there are several libraries available. Amongst them is the Java Topology Suite (Davis 2006), (Davis 2003). This is a very popular library used by numerous other projects to work with geospatial data. But, even though it has all the spatial functionality needed, the programmer still has to create application logic and functionality for the instantiation of the correct JTS objects to match the corresponding GML elements. This demands a rather intimate knowledge of both JTS and GML, is time consuming, and may result in overly complex systems with high coupling. As touched briefly in section 1.2, XSLT has been successfully used in other settings where the transformation of XML data to some other format is needed. XSLT is designed as a declarative language such as SQL, allowing the developer to focus on what should be done instead of the implementation details. One of the positive things about the XSLT approach is the pattern-matching ideology defined in the Processing-Model (Clark 1999). Instead of the traditional procedural dataflow, each template declares a pattern of data it matches. When running the stylesheet, data applied is matched with each template which in turn is executed if the match is a success. This allows for very flexible templates that can be used in different contexts. Additionally, being XML itself, it is completely independent of the platform and language deploying it, and can be used on a wide array of systems without ever changing the stylesheet itself. If XSLT could be extended to support standard spatial operations, it would probably not only standardize a lot of work with GML in applications, but also lower the bar for working with and including spatial data for everyone to use. If doing spatial queries on WFS and then operate on the data could be just as easy as working with ordinary web services and XSLT with functions, axis, and elements, it is tempting to assume that its use would be more widespread and spatial functionality find its way into many more applications. Another scenario is the need to lookup and merge data from two separate datasets or layers. An example of this can be taken from the tiger dataset available with the 1.3.1 distribution of GeoServer. In addition to the aforementioned road layer, the set has a layer representing points of interest (poi) in Manhattan.

Say that we were developing a website or application for tourists that want a list of attractions along the road they are heading (or any given road for that sake). To achieve this with the WFS based data source, it would then be necessary to first get the coordinates of the road strip and then do a search within the proximity of the road along its length.

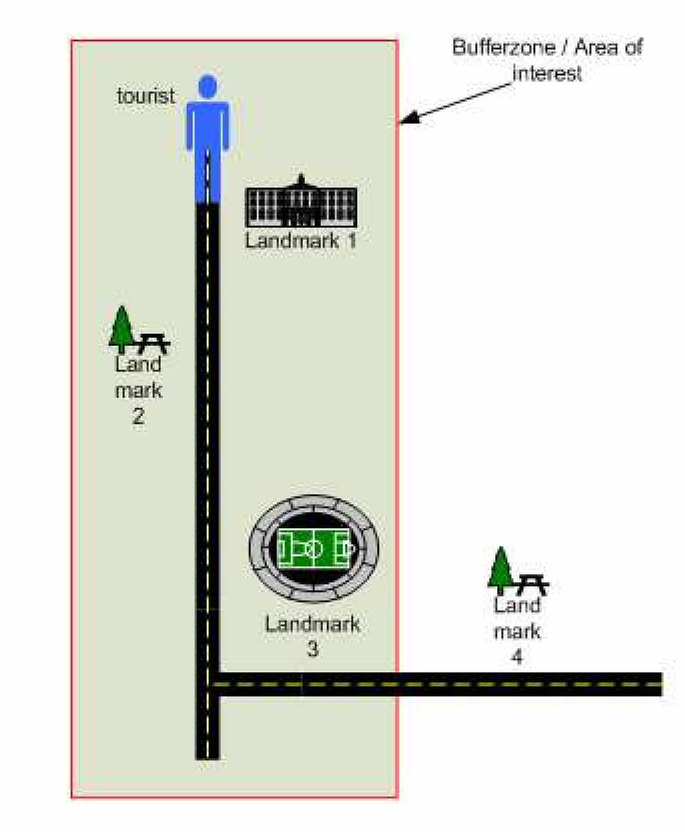


Figure 2.5 illustration of buffer scenario

Figure 2.5 illustrates the concept. The tourist walks down a road strip, and

landmarks 1,2, and 3 are of interest because they are within a given zone/proximity along the road. Landmark 4 is too far away and is not of interest. The key here is to compute the bounding box of the buffer zone and then do a new search within the landmark data. With plain vanilla XSL, this bounding box calculation would imply some rather ugly code, -if possible at all. The intuitive approach would be to use XSL or DOM manipulation to extract the bounding box of the road, and then use JTS or some other geospatial functionality directly for the calculation of the bounding box of the buffer zone, and then issue a new request to the landmark web service. The returned data would then have to go through another XSLT process or DOM manipulation for extraction and formatting of the landmarks into a list. What is needed, is a way to calculate the buffer zone's bounding box on the fly from within the initial XSLT process, so that correct requests for landmarks can be done from within XSL by way of a "document()" call or some other extension function and then integrated into the generated list. The proposed conceptual flow can be illustrated by the sequence diagram below.

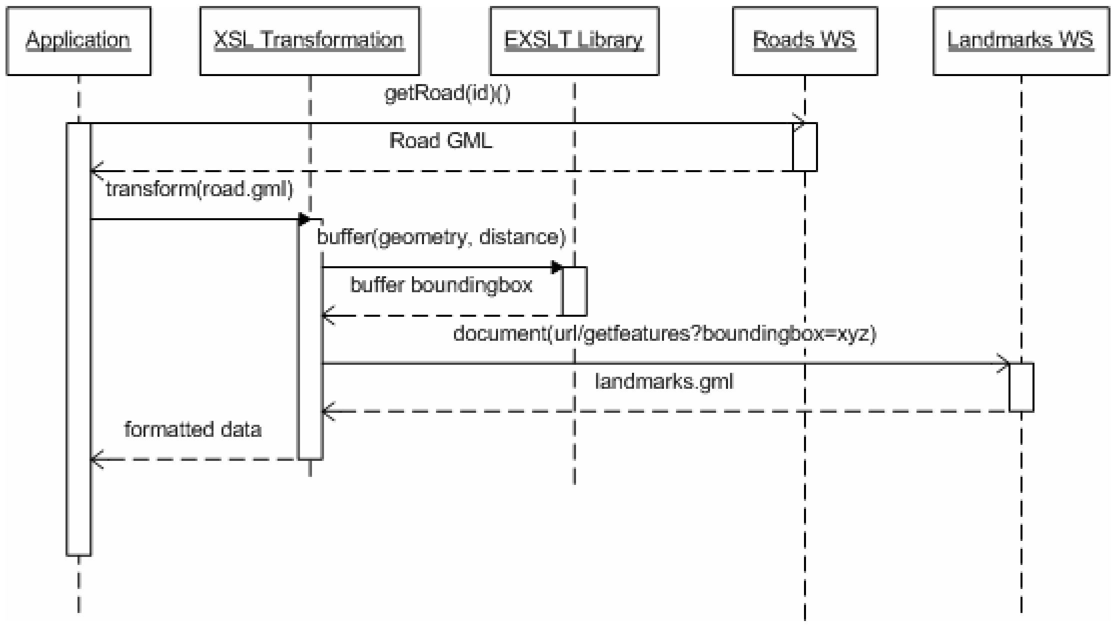


Figure 2.6 Sequence diagram showing the need for a possibility to calculate a buffer surrounding selected features of the GML.

In the diagram the application makes the initial request to the Roads WS. This can be achieved either by sending coordinates received from the Global Positioning System or such (searching for roads containing the given coordinates), or by sending the feature-id of the road (typically selected by the user in a previous window) to identify it for the WFS. The returned GML is then sent through an XSL transformation where the road's buffer zone is computed using the buffer operator (see Chapter 3 for details on the buffer operator). The returned string containing the bounding box from the buffer operator is then concatenated to a “getFeatures” request to the landmarks WFS using the “document()” function of XSLT for acquirement, or eventually some other wrapper function for external HTTP access. The returned GML can be formatted using the existing XSLT process and returned as formatted data to the application. With formatted data it is here meant anything from plain text to HTML, PDF or SVG. An alternative to using the proposed extensions for buffer operations is to implement it directly in XSLT. Although such operations probably can be implemented in XSL to a certain degree, there are several arguments not to:

- *XSLT is a language specifically designed for dealing with XML, not with the implementation of geospatial calculations.* As mentioned in section 1.2, the main motivation of moving GML processing from the generic programming languages to XSLT is to simplify the workflow, allowing developers to concentrate on the task at hand instead of the implementation details of geospatial standards. Implementing the calculations in directly in XSL is kind of the opposite, where the functionality is most easily implemented in Java or another programming language. XSLT code with a procedural workflow containing many conditionals and parameters easily gets ugly, incomprehensible, and difficult to maintain.

- *Encapsulating geospatial function as EXSLT calls follows the pattern of existing XSLT functions for operations on strings and node sets.* This thesis presents a suggestion for the mapping between a standardized set of spatial features defined in the OGC Simple Features Specification (SFS) and an EXSLT function set. This could possibly form a basis for potential inclusion in the standard XSLT functions available at a later time.
- *When the spatial functionality is integrated only through interfaces and function calls, it is easier to combine different and specialized implementations without altering the XSLT template.* Implementations of the spatial operations following interfaces based on the SFS and made available as standardized EXSLT functions allows for easier optimization of performance completely independent of XSL code. External vendors or open source projects can then create different libraries of functions following the same common interfaces but with different properties regarding performance and support for underlying elements and data sources. XSLT developers can then "upgrade" their libraries without touching the XSLT code.

2.3 Prior studies

There has not been done a large amount of research around the use of XSLT in a geospatial context. This section discusses findings of relevance and ideas from other studies with a focus on template based processing and query of XML Spatial queries and XML. To define the set of functionality to be supported it is necessary to evaluate the experiences from previous studies in combination with practical knowledge and analysis of the potential fields for use. As mentioned in the motivation, there is an intuitive interest in basing the functionality on the operations as defined in (Ryden 2005) and (Vretanos 2005) as these are the community accepted standards supported in various degrees by many projects and applications, open source and commercial alike (MySQL; Refractions Research; Davis 2003; Directions Staff 2003; Oracle 2005; ESRII 2006; Owens 2006; Schulz 2006). In (Corcoles and Gonzalez 2004), it is investigated how geographic referenced data encoded as GML can be queried in the Geospatial web, integrating spatial and non-spatial resources in a web context. Their research is of relevance as both the context and operations they discuss coincide with the motivational points of this thesis. A more detailed study on the language they have designed is presented in (Córcoles and González 2001). While they focus on query construction and how a wrapper can be used to bridge the spatial queries to a defined RDBMS by converting to spatial SQL queries, it is clear that an XSLT process could be used as the mediator in the process presented in (Corcoles and Gonzalez 2004). XSLT is suitable because it was designed to create templates for data transformation from one or more sources and data models into an output of choice and simultaneously extracting data of interest through path declarations with XPath. (Provost; W3C; Clark 1999; Holzner 2001). A common use is the generation of HTML/XHTML based on transforming and querying XML from separate sources of data, such as web services. (Corcoles and Gonzalez 2004) lists the comparative operators “cross”, ”overlap”, and ”touch” together with “area” and “length” for analysis. These can all be found in the Simple Features Specification (Ryden 2005) section 2.1.1.1-2.1.1.3. Having established that XSLT has the potential to fit into a mediator role of the model in the Simple Features Specification and that the available queries described match those of defined in it, it is interesting to note that substituting the wrapper for bridging queries to the RDBMS with the EXSLT functionality introduced in 2.1 and 3.2 should be possible as the operations available seem to overlap, - both implement the geometry operations of the Simple Features Specification (Ryden 2005). A possible advantage of using the EXSLT approach in addition to the motivations mentioned in section 2 is that while the RDBMS wrapper in (Corcoles and Gonzalez 2004) relates to a singular or limited number of databases directly available to the mediator process, the EXSLT approach as introduced in this thesis is based on WFS. WFS is available over HTTP as a web service and allows for easier integration of multiple data sources without being dependent of proprietary database drivers, syntax, and frameworks (Vretanos 2005). While both (Corcoles and Gonzalez 2004) and (Córcoles and González 2001) support the spatial queries defined in (Vretanos 2005), they are both dependent of a wrapper to translate the queries into syntax compatible with the RDBMS currently used. The need to implement a wrapper from scratch as done in (Córcoles and González 2001; Corcoles and Gonzalez 2004) to accommodate the query language is indicated as a complicating factor in both (Vatsavai 2002) and (Warnill, Soon-Young et al. 2004) and is not in line with the WFS scenario which is the context of this thesis. In (Vatsavai 2002) Vatsavai discusses the use of XQuery for spatial queries on GML using the language GML-QL, which is an extension of XQuery. XQuery is a domain-specific language for querying XML documents and features a very powerful syntax for accessing and filtering the different parts (Brundage 2004; Fernández, Malhotra et al. 2006). It is not a competitor to XPath, which is used in XSLT, but a more complicated alternative with a larger set of functionality and a procedural workflow that complements XPath/XSLT for the settings where that is needed. XQuery seems to be most commonly used in connection with database queries at the time of writing. While (Vatsavai 2002) has a focus on spatial queries using XQuery in relation to returned data from a database system, the nature of XQuery is so close to that of XSLT/XPath that the operations and syntax used is of relevance. Vatsavai has explicitly chosen to support the features as defined in (Vretanos 2005), but with an adaptation to fit the operational calls into the XQuery syntax. While (Vretanos 2005) defines operations as methods on spatial objects called with dot-notation, the examples in (Vatsavai 2002) use operations implemented as function calls where the geometries are passed as parameters. This is a syntax that matches the XSLT functions generally available and is easy to understand both in relation to (Vretanos 2005) and (Ryden 2005). Section 3.3 goes into further detail around integration of query interface.

Table 2.14

```
Operation definition in specification [2]
geometryA.operation(geometryB)

Operation example call in [18]
operation(geometryA,geometryB)
```

Vatsavai does not go into details around the implementation of the given functionality and has a perspective of using a database system as a data source. Still, it seems viable that a library built to support these operations in XSLT could also be used in an XQuery context as the operations work on the already extracted data. In (Warnill, Soon-Young et al. 2004) the authors discuss a similar extension of XQuery to support queries on moving objects. Whether to use XSLT/Xpath or XQuery would then be a matter of which scenario one is working in. A discussion of when to use XSLT/Xpath or XQuery can be found in (Brundage 2004; Fernández, Malhotra et al. 2006).

2.4 Chapter summary

This chapter has presented a view on how Web Feature Servers are sources of geographic data. Further it described the need to process returned GML data locally, and how XSLT fits into the workflow. It also discussed guidelines for the implementation of necessary spatial functionality and in the review of previous research it showed how extension of XQuery has been used to implement spatial functionality.

3 Supported functionality

As the previous research in 2.3 outlines a syntax that can be used with external functions for access from XSLT, this section will present the operations suggested for implementation and the motivation for their use.

The OpenGIS Filter Encoding Implementation Specification (FEIS) and the OpenGIS Simple Features Specification (SFS) define a set of spatial operators for comparison and analysis. While these are a required component of queries to WFS servers, they are also used in other OGC web services such as Gazetteer and Web Registry Services (Vretanos 2005). By supporting these operations we ensure that there is a common set of analysis and comparison operations shared between WFS queries and local operations from within on the loaded data set. Aside from the good practice of standard compliance, this makes both general usage and implementation easier. Support for these operations also contributes to make the XSLT approach fit into the OGC workflow. What follows is a presentation of the spatial operators from the two specifications that apply to generic geographic objects with comments on why and how they can be of use in a XSLT/template context and an API specification of their use from XSLT. Section 3.1 presents the comparison operations, while 3.2 details the analysis operations. Section 3.3 defines and exemplifies the interface determined used for the integration between EXSLT and calls to the framework implemented the spatial operations.

The SFS additionally defines operations specific to the different geometric types. While it is possible to implement these with the framework developed here, doing so is outside of the scope of this thesis.

3.1 Operations for testing spatial relations on generic geographic objects

These operators enable the user to test for the validity of certain fundamental spatial conditions, and return either true or false. In addition to being used on

WFS servers today, they are also supported on databases with spatial capabilities such as PostGIS, MySQL and Oracle. In other words they are proven to be of vital importance and should have a natural place in any system designed for work with geographic data.

Equals

Equals(anotherGeometry:Geometry):Integer - Returns 1 (TRUE) if this Geometry is 'spatially equal' to anotherGeometry. (Ryden 2005)

This operation is used to compare one geometric object to another. In a template context, this can be very useful. One practical example is the need for removal of duplicate geometries when merging datasets from multiple WFS queries.

Table 3.1

Param types	#Params	Return value
Nodes containing GML adherent geometry elements	2	True | false

Table 3.2

```
<xsl:variable name="example" select="geo:equals(elem1,elem2)" />
```

Disjoint

Disjoint(anotherGeometry:Geometry):Integer- Returns 1 (TRUE) if this Geometry is 'spatially disjoint' from anotherGeometry. (Ryden 2005)

This operation is used to check if two objects in any way cover parts of the same area (a.Disjoint(b) a b). In a practical application this can be used in a check of uniqueness of coverage. For example, when working with georeferenced real-estate data, checks on disjointness can be used to detect if someone has built something on someone else's property. If a house is not built on property a, it is disjoint to property a.

Param types	#Params	Return value
Nodes containing GML adherent geometry elements	2	True | false

Table 3.3

```
<xsl:variable name="example" select="geo:disjoint(elem1,elem2)" />
```

Table 3.4

Intersects

Intersects(anotherGeometry:Geometry):Integer- Returns 1 (TRUE) if this Geometry 'spatially intersects' anotherGeometry. (Ryden 2005)

Intersects corresponds to the intersect operator of traditional set theory applied on geographic objects. A practical use of the intersects operator is to check if an area at least partly covers another area.

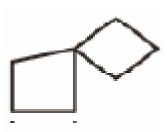
Param types	#Params	Return value
Nodes containing GML adherent geometry elements	2	True | false

Table 3.5

```
<xsl:variable name="example" select="geo:disjoint(elem1,elem2)" />
```

Table 3.6

Touches



Touches(anotherGeometry:Geometry):Integer- Returns 1 (TRUE) if this Geometry 'spatially touches' anotherGeometry. (Ryden 2005).

According to the SFS, this operation applies to all geographic objects except for points. Hence it can be used for operations such as checking if the boundaries of any two given roads touch. This again can be used in the computation of road-descriptions; if two roads touch each other, it could be possible to cross from one road to the other.

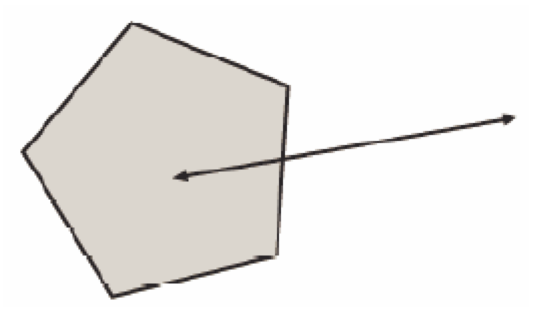
Param types	#Params	Return value
Nodes containing GML adherent geometry elements	2	True | false

Table 3.7

```
<xsl:variable name="example" select="geo:touches(elem1,elem2)" />
```

Table 3.8

Crosses



Crosses(anotherGeometry:Geometry):Integer- Returns 1 (TRUE) if this Geometry 'spatially crosses' anotherGeometry. (Ryden 2005)

Corresponds to intersects, but for use with lines and points relative to lines and areas.

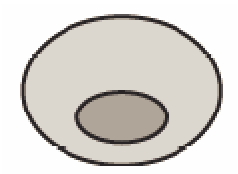
Param types	#Params	Return value
Nodes containing GML adherent geometry elements	2	True | false

Table 3.9

```
<xsl:variable name="example" select="geo:crosses(elem1,elem2)" />
```

Table 3.10

Within



Within(anotherGeometry:Geometry):Integer - Returns 1 (TRUE) if this Geometry is 'spatially within' anotherGeometry. (Ryden 2005)

This operator checks whether a geometry is situated within the area of another geometry. For example, it can be used to check if a house is within a given administrative district.

Param types	#Params	Return value
Nodes containing GML adherent geometry elements	2	True | false

Table 3.11

```
<xsl:variable name="example" select="geo:within(elem1,elem2)" />
```

Table 3.12

Contains

Contains(anotherGeometry:Geometry):Integer - Returns 1 (TRUE) if this Geometry 'spatially contains' anotherGeometry. (Ryden 2005)

Corresponds to the within operator, but check if a given geometry has another geometry within its area.

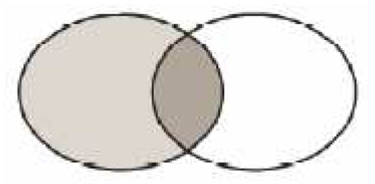
Param types	#Params	Return value
Nodes containing GML adherent geometry elements	2	True | false

Table 3.13

```
<xsl:variable name="example" select="geo:contains(elem1,elem2)" />
```

Table 3.14

Overlaps



Overlaps(anotherGeometry:Geometry):Integer - Returns 1 (TRUE) if this Geometry 'spatially overlaps' anotherGeometry. (Ryden 2005)

Param types	#Params	Return value
Nodes containing GML adherent geometry elements	2	True | false

Table 3.15

```
<xsl:variable name="example" select="geo:overlaps(elem1,elem2)" />
```

Table 3.16

Relate

Relate(anotherGeometry:Geometry, intersectionPatternMatrix:String):Integer- Returns 1 (TRUE) if this Geometry is spatially related to anotherGeometry, by testing for intersections between the Interior, Boundary and Exterior of the two geometries as specified by the values in the intersectionPatternMatrix. (Ryden 2005).

Not a part of the FEIS.

Param types	#Params	Return value
Nodes containing GML adherent geometry elements	2	True | false

Table 3.17

```
<xsl:variable name="example" select="geo:relate(elem1,elem2)" />
```

Table 3.18

3.2 Operations that support spatial analysis on generic geographic objects

In addition to the spatial test operators, there is a need to manipulate geographic data. To support this, there is a need to support the methods for spatial analysis as defined in the SFS. This section gives a listing of the required methods with brief comments on their practical use.

Distance

Distance(anotherGeometry:Geometry):Double - Returns the shortest distance between any two points in the two geometries as calculated in the spatial reference system of this Geometry. (Ryden 2005).

The distance method is designed to calculate the distance between two objects.

This can be used in a practical application such as measuring the distance between two buildings, ships, or landmarks.

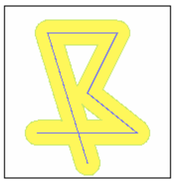
Param types	#Params	Return value
Nodes containing GML adherent geometry elements	2	XSLT Number

Table 3.19

```
<xsl:variable name="example" select="geo:distance(elem1,elem2)" />
```

Table 3.20

Buffer



Buffer(distance:Double):Geometry - Returns a geometry that represents all *points whose distance from this Geometry is less than or equal to distance.*

Calculations are in the Spatial Reference System of this Geometry.

(Ryden 2005).

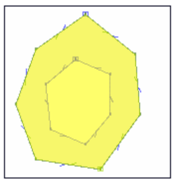
Param types	#Params	Return value
Node containing GML adherent geometry elements	1	XSLT Node set

Table 3.21

```
<xsl:variable name="example" select="geo:buffer(elem1,elem2)" />
```

Table 3.22

Convex hull



ConvexHull( ):Geometry - Returns a geometry that represents the convex hull of this Geometry. (Ryden 2005).

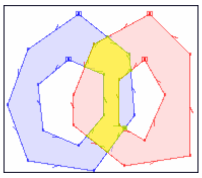
Param types	#Params	Return value
Nodes containing GML adherent geometry elements	2	XSLT Node set

Table 3.23

```
<xsl:variable name="example" select="geo:convexhull(elem1,elem2)" />
```

Table 3.24

Intersection



Intersection(anotherGeometry:Geometry):Geometry - Returns a geometry that represents the point set intersection of this Geometry with anotherGeometry. (Ryden 2005).

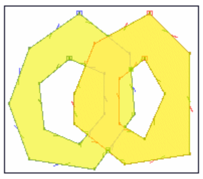
Param types	#Params	Return value
Nodes containing GML adherent geometry elements	2	XSLT Node set

Table 3.25

```
<xsl:variable name="example" select="geo:intersection(elem1,elem2)" />
```

Table 3.26

Union



Union(anotherGeometry:Geometry):Geometry - Returns a geometry that represents the point set union of this Geometry with anotherGeometry. (Ryden 2005)

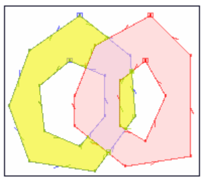
Param types	#Params	Return value
Nodes containing GML adherent geometry elements	2	XSLT Node set

Table 3.27

```
<xsl:variable name="example" select="geo:union(elem1,elem2)" />
```

Table 3.28

Difference



Difference(anotherGeometry:Geometry):Geometry - Returns a geometry that represents the point set difference of this Geometry with anotherGeometry. (Ryden 2005)

Param types	#Params	Return value
Nodes containing GML adherent geometry elements	2	XSLT Node set

Table 3.29

```
<xsl:variable name="example" select="geo:difference(elem1,elem2)" />
```

Table 3.30

Symmetric difference

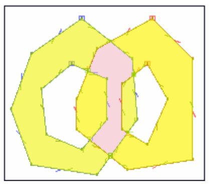


Figure 3.1

SymDifference(anotherGeometry:Geometry):Geometry - Returns a geometry that represents the point set symmetric difference of this geometry with another geometry. (Ryden 2005)

Param types	#Params	Return value
Nodes containing GML adherent geometry elements	2	XSLT Node set

Table 3.31

```
<xsl:variable name="example" select="geo:symmetricdifference(elem1,elem2)" />
```

Table 3.32

3.3 Interface integration

The interface provided to the developers of XSL templates must be powerful
enough to be of practical use, yet it must adhere to the language standards and general usage patterns for XSL. This section defines how the spatial functionality in focus can be available for XSLT as an API resembling regular functionality. The XSLT 1.0 specification allows for two kinds of extensions to XSL; functions and elements (Clark 1999). While both of these are described in Section 2.1, we have favored extension-*functions* for integration of the defined spatial functionality. Extension elements do provide the flexibility and functionality needed, but an approach using functions to make the operations available resembles the more well-known XSLT functions and fundamental XPath syntax better. More details around the use of functions versus elements are presented in 0 where the use of extension elements for configuration is discussed. It is also assumed that it will be easier to start adopting the interface when the user can build upon concepts and patterns with which he or she has experience.

Table 3.33

```
Operation definition in specification [2]
geometryA.operation(geometryB)

Operation example call in [18]
operation(geometryA,geometryB)
```

As discussed more closely in the next section, the operations to be made available can be split between the ones returning boolean values and the ones returning new geometric elements.

Table 3.34

```
1. Regular XSLT Function for string concatenation:
<xsl:variable name="strings" select="concat(string1,string2)" />

2. Proposed EXSLT function for union operations
<xsl:variable name="areas" select="geo:union(area1,area2)" />
```

3.4 Chapter summary

This chapter has presented the operations that the framework should support and the reasoning for why they should be supported. Further, a definition and explanation for the syntax used in the EXSLT calls has been given. While the functions described have been limited to standard operations of the Simple Feature Specification, operations for generalization/simplification and other spatial calculations are of relevance. Although the implementation of these has not been the main focus of the work with the thesis, the framework developed supports many operations through the functionality available in the Java Topology Suite (JTS) (Davis 2003; Davis 2006). JTS is used as an underlying framework for geospatial calculations, and is discussed further in chapter 4. Section 7.3 also presents and discusses an experimental implementation of the Douglas Peucker simplification algorithm.

4 Implementation of supported functionality

This chapter presents the design and implementation of GeoXSLT, a system supporting the defined extension functions. A top-down view of the architecture and the ideas behind the division of functionality is given in 4.1. To provide the necessary understanding of how extension calls from XSLT are integrated with the implementation, section 4.2 outlines how function calls in XSLT can be mapped to Java classes. Sections 4.3-4.5 give a detailed explanation of the various system levels outlined in 4.1.

4.1 Division of levels

Based on similarities with standard XSLT functions, examples from the Simple Features Specification (Ryden 2005), and the work of Vatsavai (Vatsavai 2002), Chapter 3 defined the functionality and syntax to be supported. The aim of this EXSLT implementation is hence to support the given operations within a context of easy use and adaptation, while at the same time keeping the door open for later performance improvements in the underlying framework.

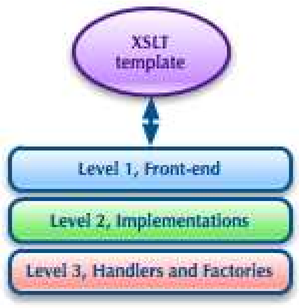


Figure 4.1: System levels

A focus has also been placed on making it very simple to integrate other operations which can operate on pre-made geometry objects. The simplicity of general use and integration of new functionality is sought done with a division of the system into three different levels. There is no need to do any changes to existing Java code to start using it, as long as the jar/class files are available on the classpath. Figure 4.1 illustrates the division of levels. As indicated by the figure, operations called from the XSL templates relate to the level 1 classes. An overview of the contents for each level is presented below:

- *Level 1, Front-end*. The general idea is that front-end classes support given operations by implementing interfaces defining method signatures and return values. No calculations are to occur at this level. Front-end classes are designed to act as wrappers, passing the request on to an implementing level 2 class for processing. By doing this, functionality implemented across several level 2 classes can be accessed through a singular class/interface. This allows for the grouping of operations by functionality under a relevant namespace, without having to change the underlying class model(s). In the thesis' implementation, a singular namespace has been created for the practical combination of comparison operations; called predicates, and for analysis operations. This matches the operation definitions in 3.1 and 3.2.

- *Level 2, Implementation*. The implementation classes are the units where the calculations are performed. The experience with the development done shows that these classes usually implement operations at a high level, as external libraries have been used for the calculations and level 3 classes handle all data conversion. With this in mind, the level 2 classes can be thought of as implementing the *logic* of the operation.
- *Level 3, Handlers and factories*. The level 3 classes are the workhorses of this framework and are exclusively used by the implementation classes at level 2. They provide functionality to convert data structures from the XSLT process into objects that support the operations performed in the implementation classes at level 2. Further, they provide functionality for conversion of the resulting data from analysis operations back into a format acceptable for return to the XSLT process. As of now, conversion between GML node structures and geometry objects of the Java Topology suite are supported.

4.2 Access to EXSLT functions from XSLT

This section gives a rundown on how extension functionality is available from XSLT templates. A special focus is on how the functionality of the implemented framework can be accessed.

To use extension functionality; EXSLT, in XSLT, the general approach is to define a namespace representing the specific group of extensions in the root element of the stylesheet (Clark 1999; Apache Xalan Community 2005; EXSLT.ORG 2006). Standard extensions, such as those defined by exslt.org are defined by a URI to a resource available on the web. The API documentation is usually available at the address for reference.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
            xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
        xmlns:math="http://exslt.org/math">
```

Table 4.1 Example of namespace declaration for standard extensions.

Standard extensions are defined by the community (EXSLT.ORG 2006) and very often implemented as an internal part of the XSLT processor. The processor used here, Xalan (Apache Xalan Community 2005), supports many of them. A complete listing of supported extensions is available online (Apache Xalan Community 2005). In the case of extensions that are not handled automatically based on an http based URI, the Java class or package name (Apache Xalan Community 2005) has to be defined in the namespace declaration of the stylesheet. This is the case with the extension framework of this thesis.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
          xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
          xmlns:geo="xalan://package[.classname]"
          >
```

Table 4.2 Example of how the a java class is tied into the "geo" namespace

In Table 4.2 an example of how Java functionality not handled automatically is made available from XSLT stylesheets processed with Xalan is listed. For all templates contained in the stylesheet, defining the "geo" namespace, the methods of the class "classname" are available as functions. Each of the operations implemented in this thesis has an example of an XSLT function call in 3.1 and 3.2. In the case of community-wide acceptance for the set of spatial extensions, a similar http based namespace mapping in Xalan as for the exslt.org functions could of course be integrated.

4.3 Implementation of Front-end/Level 1

This section gives a presentation of the level 1 architecture introduced.

The front-end classes make out the interface between the extension call from XSLT and the Java framework. When a call is made, Xalan automatically converts the XSLT types to Java objects as listed in Table 4.3.

Table 4.3 Xalan conversion of XSLT types to Java objects

XSLT Type	Java Type
Node Set	Org.w3c.dom.NodeList
String	Java.lang.String
Boolean	Java.lang.Boolean
Number	Java.lang.Double
Result Tree Fragment	Org.w3c.doc.NodeList

Methods in the front-end classes therefore need to use the parameter signatures as listed above. For practical reasons, the functionality defined in chapter 3 is accessible through one front-end class, but there is no reason why it cannot be split based on another grouping.

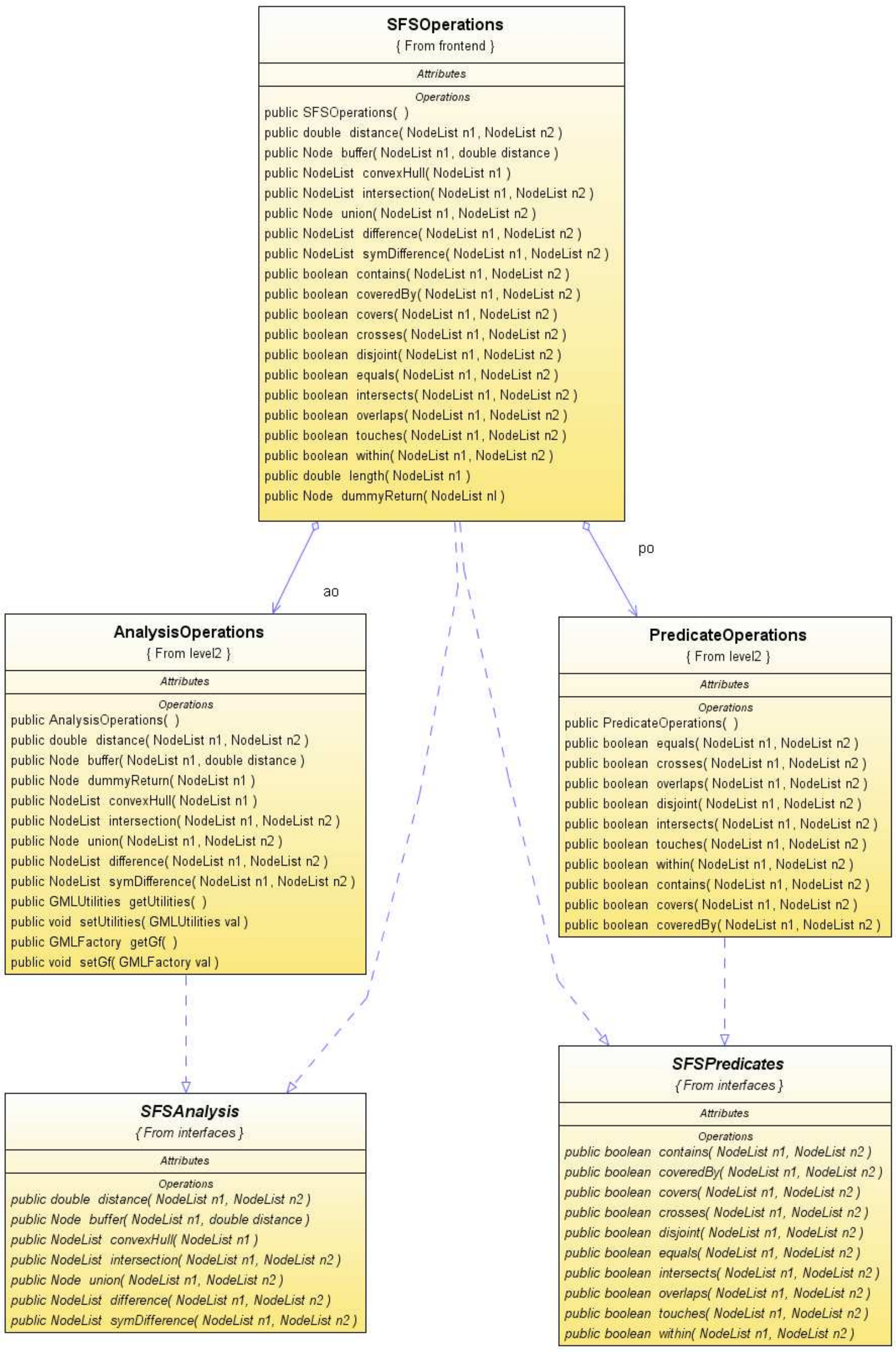

SFSOperations
{ From frontend }
Attributes
Operations
public SFSOperations( )
public double distance(NodeList n1, NodeList n2)
public Node buffer(NodeList n1, double distance)
public NodeList convexHull(NodeList n1)
public NodeList intersection(NodeList n1, NodeList n2)
public Node union(NodeList n1, NodeList n2)
public NodeList difference(NodeList n1, NodeList n2)
public NodeList symDifference(NodeList n1, NodeList n2)
public boolean contains(NodeList n1, NodeList n2)
public boolean coveredBy(NodeList n1, NodeList n2)
public boolean covers(NodeList n1, NodeList n2)
public boolean crosses(NodeList n1, NodeList n2)
public boolean disjoint(NodeList n1, NodeList n2)
public boolean equals(NodeList n1, NodeList n2)
public boolean intersects(NodeList n1, NodeList n2)
public boolean overlaps(NodeList n1, NodeList n2)
public boolean touches(NodeList n1, NodeList n2)
public boolean within(NodeList n1, NodeList n2)
public double length(NodeList n1)
public Node dummyReturn(NodeList nl)
ao
po
AnalysisOperations
{ From level2 }
Attributes
Operations
public AnalysisOperations( )
public double distance(NodeList n1, NodeList n2)
public Node buffer(NodeList n1, double distance)
public Node dummyReturn(NodeList n1)
public NodeList convexHull(NodeList n1)
public NodeList intersection(NodeList n1, NodeList n2)
public Node union(NodeList n1, NodeList n2)
public NodeList difference(NodeList n1, NodeList n2)
public NodeList symDifference(NodeList n1, NodeList n2)
public GMLUtilities getUtilities( )
public void setUtilities(GMLUtilities val)
public GMLFactory getGf( )
public void setGf(GMLFactory val)
PredicateOperations
{ From level2 }
Attributes
Operations
public PredicateOperations( )
public boolean equals(NodeList n1, NodeList n2)
public boolean crosses(NodeList n1, NodeList n2)
public boolean overlaps(NodeList n1, NodeList n2)
public boolean disjoint(NodeList n1, NodeList n2)
public boolean intersects(NodeList n1, NodeList n2)
public boolean touches(NodeList n1, NodeList n2)
public boolean within(NodeList n1, NodeList n2)
public boolean contains(NodeList n1, NodeList n2)
public boolean covers(NodeList n1, NodeList n2)
public boolean coveredBy(NodeList n1, NodeList n2)
SFSAnalysis
{ From interfaces }
Attributes
Operations
public double distance(NodeList n1, NodeList n2)
public Node buffer(NodeList n1, double distance)
public NodeList convexHull(NodeList n1)
public NodeList intersection(NodeList n1, NodeList n2)
public Node union(NodeList n1, NodeList n2)
public NodeList difference(NodeList n1, NodeList n2)
public NodeList symDifference(NodeList n1, NodeList n2)
SFSPredicates
{ From interfaces }
Attributes
Operations
public boolean contains(NodeList n1, NodeList n2)
public boolean coveredBy(NodeList n1, NodeList n2)
public boolean covers(NodeList n1, NodeList n2)
public boolean crosses(NodeList n1, NodeList n2)
public boolean disjoint(NodeList n1, NodeList n2)
public boolean equals(NodeList n1, NodeList n2)
public boolean intersects(NodeList n1, NodeList n2)
public boolean overlaps(NodeList n1, NodeList n2)
public boolean touches(NodeList n1, NodeList n2)
public boolean within(NodeList n1, NodeList n2)

Figure 4.1 Dependency diagram for implemented front-end architecture

The diagram in Figure 4.1 displays the relations between the front-end class "SFSOperations" and the underlying level 2 classes which implement the logic. Note the implementation to interfaces representing the operation definitions in chapter 3. The front-end class only has references to the underlying classes, and uses them to pass on the request.

```
  public Node buffer(NodeList n1, double distance) {
     return ao.buffer(n1,distance);
  }
```

Table 4.4 Example of operation “wrapping” in front-end class

In the listing above, “ao” is a reference to the “AnalysisOperations” class as displayed in Figure 4.1 Dependency diagram for implemented front-end architecture. The result from the implementing level-2 class is returned directly to the XSLT process after processing. In an XSLT context wrapping of calls to the implementing class means that all geographic functionality used for a given template can be accessed through one convenient namespace/prefix instead of declaring multiple namespaces. Additionally, developers are free to change between different underlying level 2 classes implementing the same interfaces by either altering level-1 source code or using injection of control (IoC) (Harrop and Machacek 2005) through Spring (Spring Community 2006) or similar frameworks. All this can be done without worrying about anything else than the simple instantiation of level 2 classes with zero parameter constructors.

4.4 Architecture of implementation classes/level 2

This section explains the workflow and architecture of the level 2 classes.

While the front end classes represent the operations available to the XSLT process, the input is only passed as parameters to the classes in level 2 as described in section 4.3.

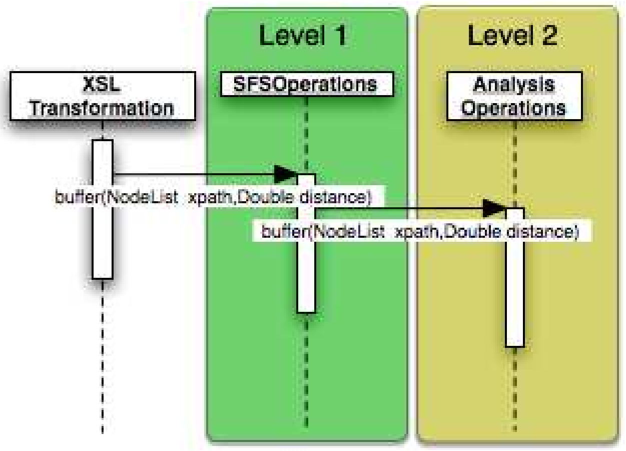

Level 1
Level 2
XSL Transformation
SFSOperations
Analysis Operations
buffer(NodeList xpath,Double distance)
buffer(NodeList xpath,Double distance)

Figure 4.2 Level 1 wrapping of operations

Figure 4.2 above gives an example of how a call to the SFS operation "buffer" propagates from the XSLT template/process to the defined level 1 interfacing front-end (green). The front-end passes the parameters on to the implementation in the level 2 class AnalysisOperations (yellow). Figure 4.3 shows how the level 2 class then uses the level 3 class GMLUtilities to construct a JTS Geometry from the GML NodeList initially passed from Xalan.

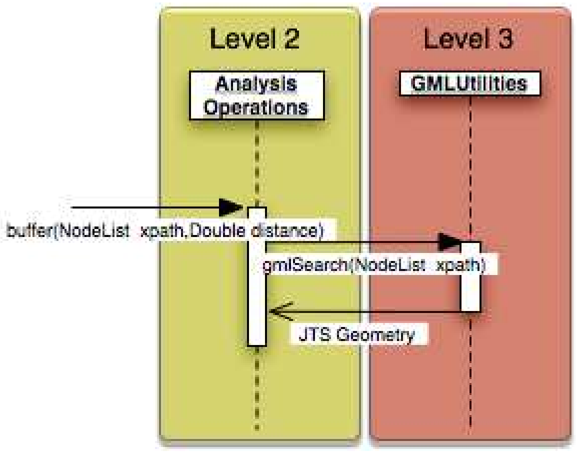

Level 2
Level 3
Analysis Operations
GMLUtilities
buffer(NodeList xpath,Double distance)
gmlSearch(NodeList xpath)
JTS Geometry

Figure 4.3 Using a level 3 class to build JTS geometries

The implementation developed in this thesis uses Java Topology Suite (JTS) (Davis 2006) as the main engine for geographic calculations. Generalizing out the building of JTS Geometry objects (Davis 2003) to level 3 classes as shown in Figure 4.3, the main area of concern for level 2 classes and their methods is to call methods corresponding to the expected calculations on the pre-made geometry objects and evaluate the results. This makes it relatively straight-forward to implement operations in level 2, and may inspire developers to integrate new functionality by using the building blocks available in the level 3 architecture. In the scope of the thesis the utility operations of level 3 make it easier to create JTS objects without unnecessary obfuscation and code redundancy. In a wider scope, it may open up for approaches inline with the Factory Pattern [28], allowing use of alternate libraries to JTS depending on the need and situation to generate the necessary geometrical representations.

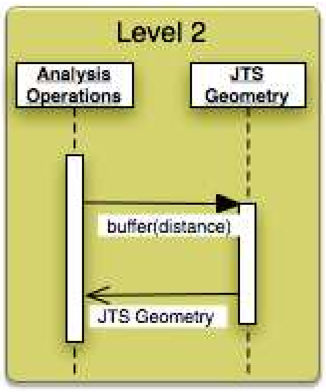


Figure 4.4 Execution of calculations on JTS geometry objects

When the calculated result for return to the XSLT process is something else than the types automatically converted by Xalan, the data is run by a level 3 class for conversion to a Xalan compatible type. In the current system, only calculations that return JTS geometries need result conversion. Table 4.5 Table 4.5 Displays Xalan accepted Java types and their corresponding XSLT type. lists XSL data types and Java mappings as accepted by Xalan. Only the subset that are of relevance to this implementation is shown. The process of converting a JTS geometry to a Xalan compatible data type is termed "JTS serialization".

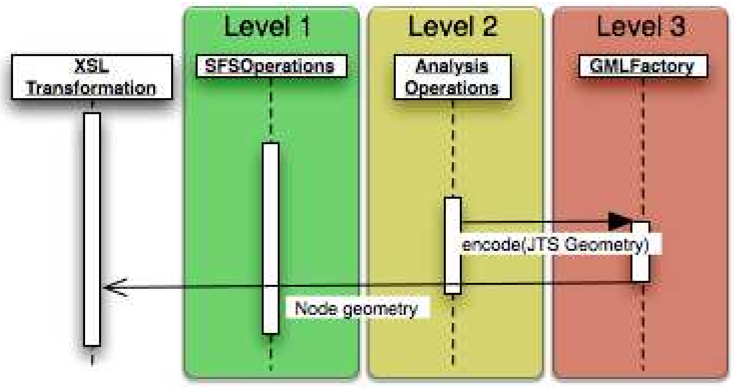


Figure 4.5 Level 2 class utilizing the level 3 architecture for serializing JTS Geometry to Xalan compatible format by calling the "encode" method of the GMLFactory class.

Java Types	XSLT Type
org.w3c.dom.traversal.Node (and subclasses) org.w3c.dom.traversal.NodeList	Node-Set
Java.lang.Double, int, double	Number
Java.lang.Boolean, Boolean	Boolean

Table 4.5 Displays Xalan accepted Java types and their corresponding XSLT type.

4.5 Implementation of the level 3 classes

This section describes the core of the framework for spatial support in XSLT.
While the front-end classes in this model act as wrappers and the level 2 classes define logic expressed on existing objects, the major trunk of development has been centered on level 3. Level 3 creates as a bridge between nodes from the XSLT process and the performed geo-operations. The functionality developed in level 3 for detecting geometries and building JTS geo-objects from node lists provided by Xalan is crucial to allow for the integration of any spatial operation defined in Chapter 3. Additionally, the library for “serializing” the geo-objects into node objects as required by Xalan is necessary for all spatial operations that are to return anything else than numeric or Boolean values.

4.5.1 Cost of use

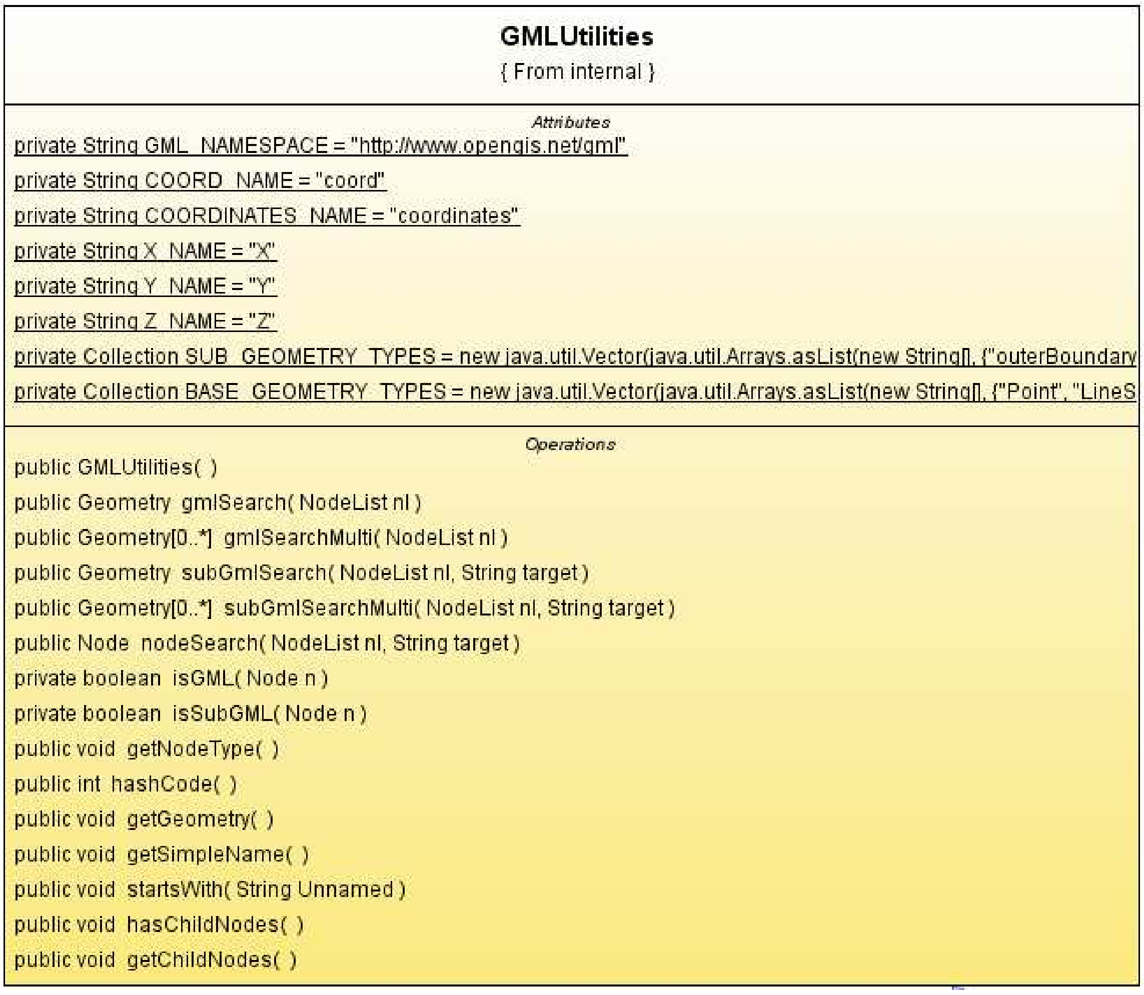
All the conversion does come at a cost, as such the bridge provided by the level 3 classes can be regarded as a performance penalty induced by the need for data conversion to support the easier and more accessible geo-functionality provided in the XSLT context. It is important to be aware that this implementation is not designed particularly with speed in mind, but as a proof of concept. Even so, as later sections will go into further detail on, there are several changes that can be applied for quick wins with regard for the speed optimizing of parsing and serialization.

4.5.2 Service interfaces

Inline with the layered and loosely coupled architecture of the whole implementation, the level 3 classes communicate with level 2 through only two operations; one for building JTS geometry objects, and another for serializing them.

4.5.3 Creation of JTS Geometries from node lists.

What follows is an overview of the workflow for the build process of geometry objects based on the node lists passed on from Xalan.



```
GMLUtilities
{ From internal }

Attributes
private String GML_NAMESPACE = "http://www.opengis.net/gml"
private String COORD_NAME = "coord"
private String COORDINATES_NAME = "coordinates"
private String X_NAME = "X"
private String Y_NAME = "Y"
private String Z_NAME = "Z"
private Collection SUB_GEOMETRY_TYPES = new java.util.Vector(java.util.Arrays.asList(new String[], {"outerBoundary
private Collection BASE_GEOMETRY_TYPES = new java.util.Vector(java.util.Arrays.asList(new String[], {"Point", "LineS

Operations
public GMLUtilities( )
public Geometry  gmlSearch( NodeList nl )
public Geometry[0..*]  gmlSearchMulti( NodeList nl )
public Geometry  subGmlSearch( NodeList nl, String target )
public Geometry[0..*]  subGmlSearchMulti( NodeList nl, String target )
public Node  nodeSearch( NodeList nl, String target )
private boolean  isGML( Node n )
private boolean  isSubGML( Node n )
public void  getNodeType( )
public int  hashCode( )
public void  getGeometry( )
public void  getSimpleName( )
public void  startsWith( String Unnamed )
public void  hasChildNodes( )
public void  getChildNodes( )
```

Figure 4.6 Class diagram of GMLUtilities

Detection of geometry representations

The level 3 class "GMLUtilities" receives a call to the "gmlSearch" method with a list of nodes (org.w3c.dom.NodeList). The list is then iterated and each node is checked by running it through the "isGML" Boolean method. The isGML method in this implementation simply checks whether the namespace is "http://www.opengis.net/gml" and then if the element's local name/tag name is among the defined names in a local collection of valid GML element names. Whilst this can be a relatively naïve approach for detection of GML elements, there should not be any significant problems to extend it for use of actual schema validation. The reason it was not done in this case is that it was not deemed necessary for a proof-of-concept implementation and would also have a significant effect on execution speed.

Construction of geometry objects

If a node is found to be a GML node, it is sent to the “create” method of “SubHandlerFactory”.

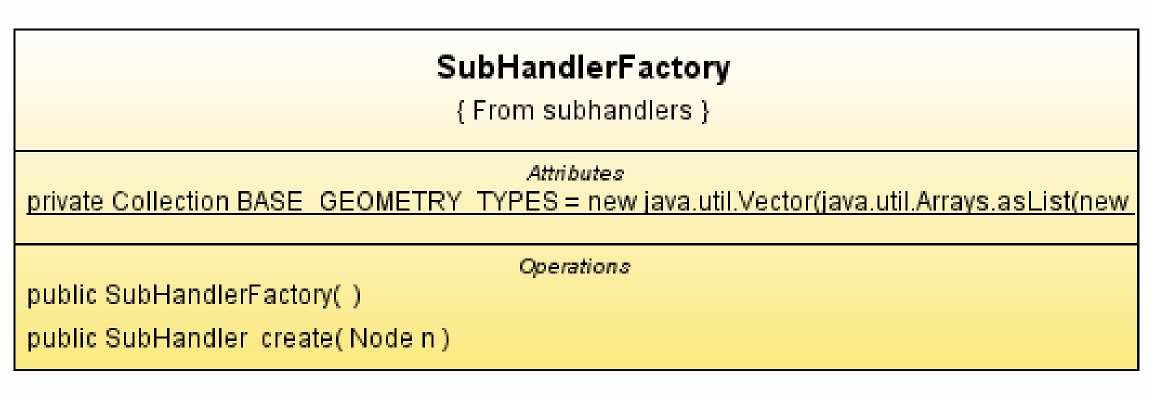

SubHandlerFactory
{ From subhandlers }
Attributes
private Collection BASE_GEOMETRY_TYPES = new java.util.Vector(java.util.Arrays.asList(new
Operations
public SubHandlerFactory( )
public SubHandler create(Node n)

Figure 4.7 Class diagram of SubHandlerFactory

The factory implementation has a line of handlers extending a “SubHandler” super class and specialized for each GML type supported. The modular design allows for the integration of better and faster handlers as developed and needed.

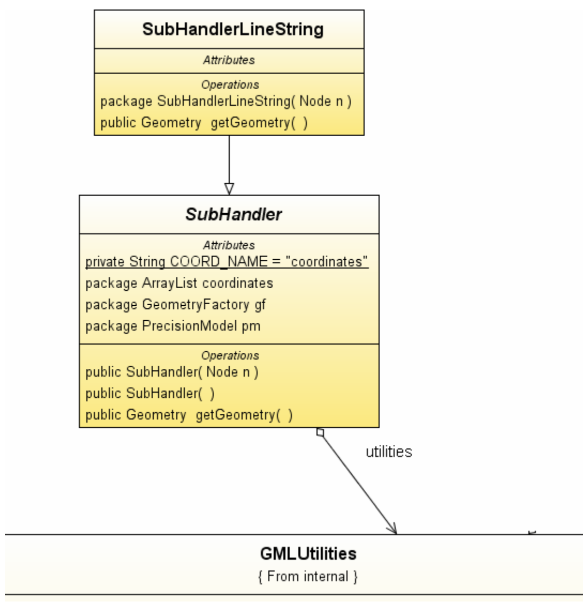

SubHandlerLineString
Attributes
Operations
package SubHandlerLineString(Node n)
public Geometry getGeometry( )
SubHandler
Attributes
private String COORD_NAME = "coordinates"
package ArrayList coordinates
package GeometryFactory gf
package PrecisionModel pm
Operations
public SubHandler(Node n)
public SubHandler( )
public Geometry getGeometry( )
utilities
GMLUtilities
{ From internal }

Figure 4.8 SubHandler super class and SubHandlerLinestring

The current “SubHandler” super class provides an instance of the GMLUtilities class, the CoordinateFactory, and the JTS library ”com.vividsolutions.jts.geom.GeometryFactory”. Figure 4.8 displays an illustrative example of the relation between the SubHandler, GMLUtilities and the specialization class for handling line string geometries.

The shared constructor of the super class initializes the common precision model (provided by the implementation's static and dedicated configuration class) and other plumbing. Additionally, it executes a search for the node containing the coordinates enclosed as a child of the GML node passed to it. The search for the element containing the coordinates is not implemented locally in the SubHandler, but is performed by a "node search" method of the GMLUtilities class, taking a node list and a target name as parameters. The reasoning behind this is that it allows for later optimization, without having to change the handler classes. Currently the node search is only a shallow iterative check for node names matching the target. The coordinate node found is then sent to the CoordinateFactory which configures itself based on the attributes Comma Separation (CS), Tuple Separation (TS), and decimal character of the coordinate node as allowed by the GML 2.1.2 specification, Section 4.3.1 (Cox, Cuthbert et al. 2002) (OpenGeoSpatial 2002). Each vertex found is passed to the "com.vividsolutions.jts.geom.Coordinate" (Davis 2004) constructor and the generated coordinate object is then stored in a list structure for return after the reading is complete.

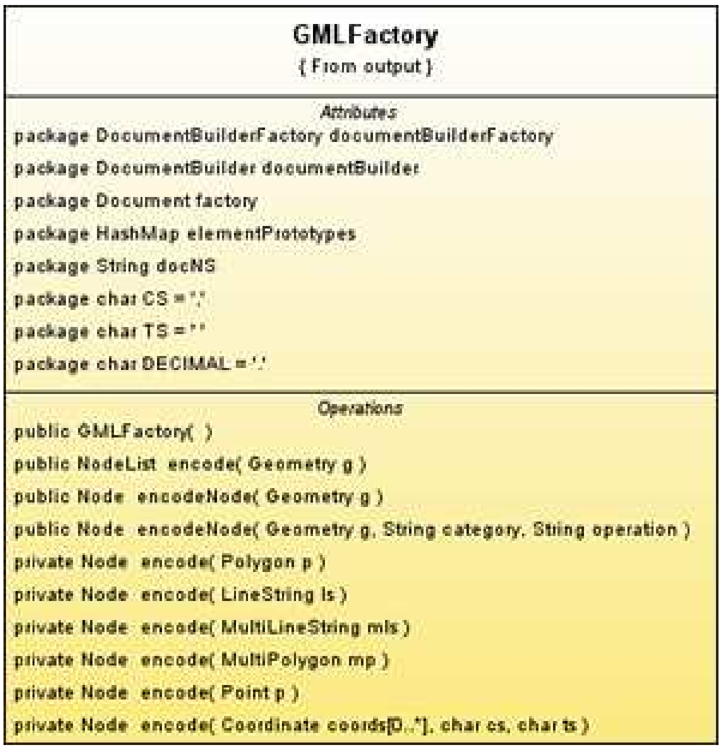
Specialized coordinate parsing in sub handlers

The specialization done in the different extensions of SubHandler varies depending on the complexity of the JTS object to be built. For handlers representing the simpler geometries point, line, and linear ring, the only extension done is to call the corresponding factory creation method of the JTS GeometryFactory class. The list structure with coordinates created by the super class is passed as a parameter. For the more complex elements such as Polygons and Multi-versions of the basic elements, a much more custom overriding of the super class' constructor has been created. In the case of polygons which consist of a mandatory outer ring and zero to many "holes", alternatives to the "gmlSearch" method are used to detect and build representations of the subelements. The polygon SubHandler extension then assembles these sub-geometries into a polygon which is returned to the original gmlSearch method call and further up to the level 2 class initiating the build. The approach is similar for the other complex geometries, with inner calls to the build the simple geometries for final assembly into the complex geometry.

4.5.4 Serialization

In the context of this thesis, serialization is defined as the process of converting/encoding JTS geometry objects to a GML compliant node structure compatible with Xalan.
After the level 2 class has executed its operations on the JTS geometry object, in the cases where the result is a new JTS geometry object, it needs to be "serialized" into a format accepted by Xalan. Table 4.5 Displays Xalan accepted Java types and their corresponding XSLT type. While Chapter 3 defines the possible return values from operations implemented.

Functions returning String, Double or Boolean values can be passed directly and converted internally by Xalan, but the majority of the analysis operations, such as union and buffer, need conversion to Node objects before the return is passed on to the XSLT process. In the same way as building JTS objects incur a time penalty, the conversion to Node also comes at a cost. The process is initiated by a call to GMLUtilities' corresponding output handler; GMLFactory's "encodeNode" method.



```
GMLFactory
{ From output }

Attributes
package DocumentBuilderFactory documentBuilderFactory
package DocumentBuilder documentBuilder
package Document factory
package HashMap elementPrototypes
package String docNS
package char CS = ','
package char TS = ' '
package char DECIMAL = '.'

Operations
public GMLFactory( )
public NodeList  encode( Geometry g )
public Node  encodeNode( Geometry g )
public Node  encodeNode( Geometry g, String category, String operation )
private Node  encode( Polygon p )
private Node  encode( LineString ls )
private Node  encode( MultiLineString mls )
private Node  encode( MultiPolygon mp )
private Node  encode( Point p )
private Node  encode( Coordinate coords[0..*], char cs, char ts )
```

Figure 4.9 Class diagram GMLFactory

The encodeNode method accepts a JTS geometry as parameter and then delegates the encoding to internal methods after determining the geometry type by using introspection. In the same way as the sub handlers for parsing incoming GML nodes from Xalan use CoordinateFactory to detect and build simple geometries based on the enclosed coordinates for final assembly into final geometry objects, the encoding handler for each geometry uses a simpler encode method private to GMLFactory to create a coordinate node with the coordinates as text and cs/ts/decimal values from internal configuration as attributes. Simpler because all data that is needed is readily available by calling the encapsulating methods of the JTS geometry object. There is no need to execute any complex searches for elements as when parsing incoming GML elements. When the coordinate object is returned to the encode method calling it, the enclosing nodes representing it are not created from scratch but rather from clones of skeleton prototype versions with the correct namespace and prefix preset. This has been done to make it easier for later changes to the elements and integration with external resources, for example by using a factory pattern or injection of control without having to alter the method bodies.

Wrapping.

The encoded geometry nodes are always enclosed within a straight forward "TheGeometry" element. There are two reasons for this. First, when the return from one operation is re-used as input for a second operation, - for example a buffer called on the results of a union, not using a wrapper makes the node list sent to the level 2 method only contain the children of the geometry element, hence only the coordinate element or sub-geometries are available for the gmlSearch method. Second, in regular GML files, each feature usually has the GML properties enclosed within an element named such as "TheGeometry", or "_geom". This wrapping of the resulting GML makes the output more manageable. At the same time, there is no problem to omit the "TheGeometry" element by using XPath on the returned data to only select the children of the "TheGeometry" element for inclusion in some other scheme. The name of this wrapper should of course be configurable to allow for easy change and to suit the usage needs of the situation. It could possibly benefit of being of configuration through schema parsing, so that the "wrapper" is automatically created with an element name matching the schema.

Parts of the application logic/division of responsibilities around handler based parsing of GML node lists have been inspired by the design of the SAX parsing/handling for GML in Geotools written by Rob Hranac. Amongst several projects utilizing this parser-logic is the popular WFS/WMS server Geoserver, something which can be regarded as an indication that this approach works quite well.

4.5.5 Basic performance optimization

The development of the GeoXSLT system has been focused on creating a proof-of-concept implementation to show that XSLT with extensions can be used for processing geodata. A limited amount of work has been done to create mechanisms for general performance optimization. While various approaches to optimization are discussed in Chapter 7, a simplistic caching prototype has been implemented as part of the level 3 classes. The motivation and application of the mechanism are discussed in Section 7.2.2 while this section will focus on the implementation.

The caching has been implemented as a "Geometry Cache", which is an object storing previously constructed geo objects. When a new GML node is detected during the previously described "gmlSearch", a check with the Geometry Cache is done before eventual construction is begun. If it turns out that the GML node has been processed earlier, the previously constructed geo object is returned. This way, a significant portion of the construction phase is saved for GML nodes that are used repeatedly during XSLT transformation. In practice, the implementation of the Geometry Cache is a simple object with accessors for a Hash Map containing the geo objects. The fact that all Java objects have a distinct hashcode unique within the application runtime is used to distinguish the node lists passed from Xalan (Sun 2004). When a geo object is created, the hashcode from the original node list passed from Xalan is used as the hash key. As a consequence, the request for the creation of a geo object only has to pass its node list (as received from Xalan) hashcode to the Geometry Cache for the immediate return of a pre-existing JTS Geometry representation.

4.5.6 Possibilities for improvement.

There are several areas of the implementation which can presumably benefit greatly from specific improvements. The improvements can be said to revolve around three axes: functionality / runtime versatility (schema parsing/validation and automatic configuration of handlers according to the schema), speed (feature caching, faster search and construction algorithms), and quality of the program architecture (better division of classes, injection of control, extraction of interfaces, cleanup of code).

4.6 Chapter summary

This chapter has presented the architecture, components, and workflow of the system for supporting spatial extensions in XSLT. A description of EXSLT integration with Java with regard to this implementation has been used to show how the implementation relates to the transformation process. Reasoning behind the design has been explained as well as weaknesses and areas ripe for improvement.

5 Testing

The testing of this implementation aims to produce experience and statistical data to help answer the research objectives as defined in 1.1. Tests on constructed data have been designed to validate the correct functioning of the system, as described in research objective 1. Further performance tests and operations on real data are described to uncover aspects around flexibility and performance inline with research objective 2. The experiences with implementing the tests forms the basis for further discussion of research objective 3. Tests of the performance optimization introduced in Section 4.5.5 is covered in Section 5.1.5.

5.1.1 Validation of correct functioning.

To be of any practical use it is important that the calculations and returned data can be trusted. Research objective 1; can the XSLT language in combination with extensions be used for the processing of geodata, needs a combination of both quantitative and qualitative observations to be answered. Quantitative for evaluation of whether the implementation works as expected, -indicating that it is technically viable, qualitative with regard to practical evaluation of general use. The implementation has been validated by creating test cases for selected operations where the results are compared with the returned data of a third-party tool. Martin Davis at Vivid Solutions, the creators of JTS, has created a powerful visualization tool for testing the JTS library. The tool, called Testbuilder, accepts 1-2 geometries encoded as Well Known Text (WKT) which are then drawn on a canvas (Davis 2004). Both the predicate and analysis operations available in JTS can then be executed. WKT is a format for defining spatial geometries as strings of text and was specifically defined to make it possible to load spatial data into spatially enabled databases. The format was defined by the OpenGIS Consortium “Simple Features for SQL” specification (Ryden 2005) and is also a part of the ISO "SQL/MM Part: 3 Spatial" (Stolze 2003). As there is a mapping between the spatial objects represented between GML and WKT, generating test data for either is simply a matter of encoding the same coordinates within each of the two encapsulating formats. The validation of this implementation has been achieved by generating various test data for execution both in Testbuilder and in an XSLT context. The results from both predicate and analysis operations have then been compared and the data from Testbuilder have been considered authoritative to which the XSLT generated result must adhere for the test to be valid.

5.1.2 Measurement of performance

To answer the research objective "What limitations on factors such as performance, flexibility and scalability will this approach imply?" there is a need to provide data on how the implementation's performance varies with changing amounts of data input. Before measuring the performance of the implementation, it is in place to provide a definition of how performance can be interpreted and which aspects that are covered here. In (Steve Wilson 2000) performance is defined as a collection of the following factors:

- Computational Performance
    - Deals with the optimization of algorithms to use as few instructions as possible,
- RAM footprint
    - Optimization of memory usage
- Startup time
    - How to minimize time to bootstrap an application
- Scalability
    - How does an application handle large loads
- Perceived performance
    - How does the user experience execution time.

While all of these aspects are of universal interest, the aspects of performance under focus in this report are scalability and perceived performance. Computational performance and RAM footprint have not been deemed necessary to develop a working prototype. Later work could focus on designing and implementing changes to provide optimization for these aspects of the performance definition. A more detailed explanation of how the scalability and performed performance results are perceived is provided in chapter 6, 7.2 and 7.2.2 where the findings are presented and discussed. When measuring the scalability and perceived performance, the focus has as been on observing the time cost of the bridging between geometry objects and nodes/node lists as introduced in chapter 4. Therefore, it is vital to keep tabs on several steps in the process. Watches has been placed on total execution time for the EXSLT function call as a whole, the building of each geometry object involved in the operation, execution time for the JTS operation, and encoding/serialization back to a Xalan compatible node.

The log4J logging toolkit (Log4j Community) was first considered for saving the test results, but due to the (at least as understood by the author) limitation to only log text strings at singular points of time there was a problem. The test process has a defined need to log information on start, stop, context and operation performed, in addition to details on number of vertices, length and precision. With log4J output to single strings, logging seemed cumbersome when everything would have to be either aggregated in long text strings or spread across several entries. s each operation generates several events, using logger based on log4J would hence imply a need to create relatively complex handling for post processing of the generated log files with thousands of entries. Instead of doing this, a simple system inspired by log4J but with support for the required logging of multiple fields associated with each event, has been created as part of the testing framework of this thesis' implementation. The logger works like a very simple and crude version of log4J, but instead of logging formatted text strings to file or database handles, an "event factory" is used to create objects of an "Event" class. An event is created with a timestamp representing the start time. The creation of a logging event is the last operation done before the operation to be logged is to take place. After completion of the operation the event is stopped by setting a second timestamp. After the timer is stopped, each data field to be associated with the event is set through encapsulation methods in the object before the event finally is appended to a log object accessible through a static method of the event factory. Accessible in a static context to ensure thread safety and to keep the log and event objects accessible with as little time overhead as possible. The log object maintains a list of events as an ArrayList which is serialized to a MySQL database when a certain size has been reached or the log is finalized through its shutdown hook (Sun 2004). In practice, it is only serialized after all tests have executed. Log entries stored in MySQL are then accessible for analysis with e.g. Microsoft Excel through ODBC. Because nested events are logged; all build, JTS execution and serialization events are sub parts of a call to an extension function, the time spent logging data for the sub parts will accumulate and increase the time spent executing the parent event. As the times presented are not 100% accurate due to external factors such as operating system and machine ware, the accumulations of time spent logging are ignored because the data nonetheless provides the necessary indications on tendencies and relative data for comparison.

5.1.3 Selection of tests on constructed data

Due to the extensive amount of work to create usable test-cases for possible operations and data types, the testing had to be limited to a small number of operations performed on a relatively small amount of constructed data. Testing the workings and performance of the JTS library itself has been considered secondary. The focal point of all testing performed has been to validate the correct functioning of the implementation as a whole, and at the same time to measure how different kinds of data influence the bridging when converting between node lists, JTS geometries, and back to nodes for return to Xalan. It is important to note that JTS performance has been measured in the test cases implemented, but the results from those operations cannot necessarily be projected onto assumptions about other JTS operations as their internal calculations may or may not induce significantly different execution times.

Test data has been generated for variances of the simple geometry types line strings, points and polygons. The reason why complex versions of geometries have been left out is that these generally may be regarded as aggregations of the simple ones and hence not prioritized here. All tests on constructed data have been designed to be performed without the performance optimization introduced in Section 4.5.5. Each operation measured has been performed on two distinct geometries of same type, but with differences in positions. See Figure 5.1 for a visualization of test geometries.

Operations selected for testing

Only a subset of the functions defined in chapter 3 has been tested, this section presents which operations that have been chosen for testing with the different geometry objects.

Line Strings

For predicate operations on line strings, the “crosses” operation (3.1) has been selected for measurements of total time to complete. The crosses operation is very central in many real-life operations such as e.g. testing whether two roads cross each other. For analysis operations on line strings, the “union” operation (3.2) has been selected for measurements of total time to complete. The tests and experiments performed Chapter 6, 7.2.2, and 7.5 feature examples of the practical value of this operation.

Points

The “equals” operation (3.1) has been used to test predicate operations on points. For analysis operations, only a dummy call to the bridge has been used. The dummy call builds a JTS geometry based on the point which is then immediately serialized back to a GML node without any JTS calculation is executed. This is because relevant analysis operations available for execution on points typically will return other data types than points. As such it has been deemed more interesting to only focus on measuring the performance of the bridge when building and serializing the exact same point. Serialization of polygons and line strings are performed and measured in the other test cases.

Polygons

For polygons the “intersects” operation (3.1) has been used to test the total time to complete for predicate functionality on polygons. For analysis the “union” (3.2) operation has been used for testing.

Parameters tested

This section introduces the test parameters used with the tested operations and geometries, and explain the reasoning behind the choices made. Examples of parameter variation and visualizations of test data are provided.

The parameters to be tested with different factors of variance have been chosen because it is assumed that they represent factors that will have an impact on operation execution.

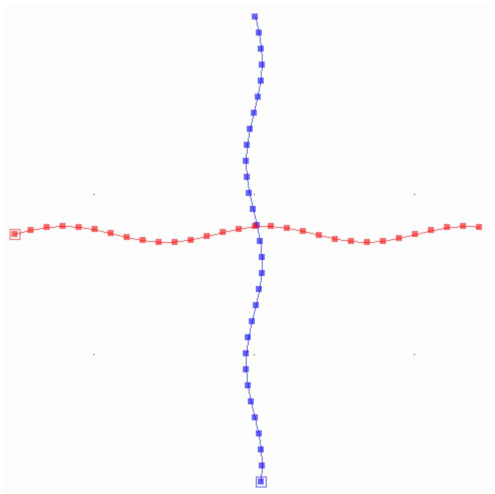
Below is a list of the parameters used:

- *Vertices.* Increasing the number of vertices in a geometry may have an effect on all phases of the execution process, as it introduces both more complexity and a larger amount of data to process.
- *Digits.* While increasing the number of digits does not necessarily have an impact on precision levels, it has an impact on the total size of data necessary to represent the geometry.
- *Holes.* This parameter only have an effect on polygons, where the holes represent both increased complexity and more data with extra processing requirements.

Testcase	Parameter				
Line Strings					
Vertices	3	30	300		
Points					
Digits	2	3	4	5	6
Polygons					
Vertices	5	41	401		
Holes	0	1	10	40	
Digits	2	3	4	5	6

Table 5.1 Test cases with constructed data

Table 5.1 lists the variations of parameters in the tests performed. Parameter values are based what is assumed to be variations with relevance to real-life data.



To reflect the variance of complexity in the execution time for JTS' calculations no lines created are straight. This is so that internal calculation simplification of vertices should be kept to a minimum. The wave shape is also considered to make the test cases more realistic in comparison to straight lines.

Figure 5.1 Visualization of crosses test on line strings with 30 vertices each.

All lines are sinus curves, and polygons are built as functions of sinus/cosines. This gives for predictable patterns which is visually easy to validate using Testbuilder. All test data has been generated with scripts written in Perl and encoded in both GML and WKT versions for easier validation. For each test case, two geometries with equal properties but different values have been created. For example, with the "crosses" test, one horizontal and one vertical line has been generated for each change in the test parameters, while the "intersects" test is designed to be executed on two polygons. Each test case has been repeated 100 times, while the whole test process is controlled and performed by JUnit.

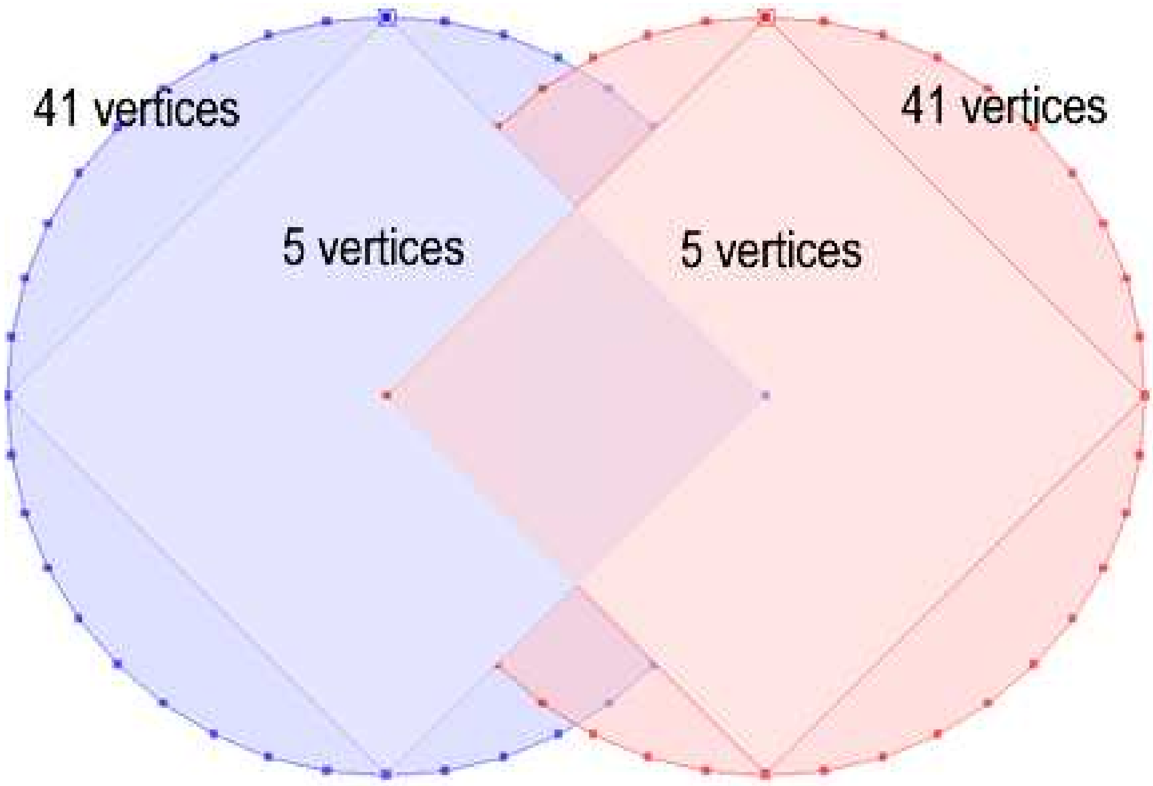


Figure 5.2 Visualization of two intersects tests performed on polygons with 5 and 41 vertices

Samples of test data are available in the appendix.

5.1.4 Tests on real data

The tests and samples described so far have all been specifically constructed to validate a given case inline with objectives to gather data on performance and scalability for isolated extension functions. To answer the research objectives defined in chapter 1, there is a need for a set of more qualitative experiences with scenarios closer to what can be expected in a real-life setting. To provide a reality check for comparison with the samples and the research objective, a composite test on sampled data has been designed. Data from Tana, a small village in Northern Norway, consisting of buildings and road stubs has been made available by the Norwegian Mapping Authorities and used for the purpose of real-life test scenarios in the work with this thesis. Buildings are represented as points and road stubs as line strings. The data consists of 9848 points and 2418 line strings. The data has been divided into ten smaller subsets to enable testing of how the system performs with a varying number of features and complexity. Table 5.2 lists the different subsets created and details on the number of features and share data size.

Case	Roads	Buildings	Size (kb)
1	3	24	16
2	16	108	61
3	25	214	116
4	56	450	244
5	201	1 132	643
6	425	2 145	1 242
7	787	4 701	2 641
8	1 690	8 163	4 773
9	2 243	9 452	5 692
10	2 418	9 848	5 985

Table 5.2 Subsets of the Tana data

The visualizations of the Tana data sets below show roads as grey lines, and buildings as red dots.

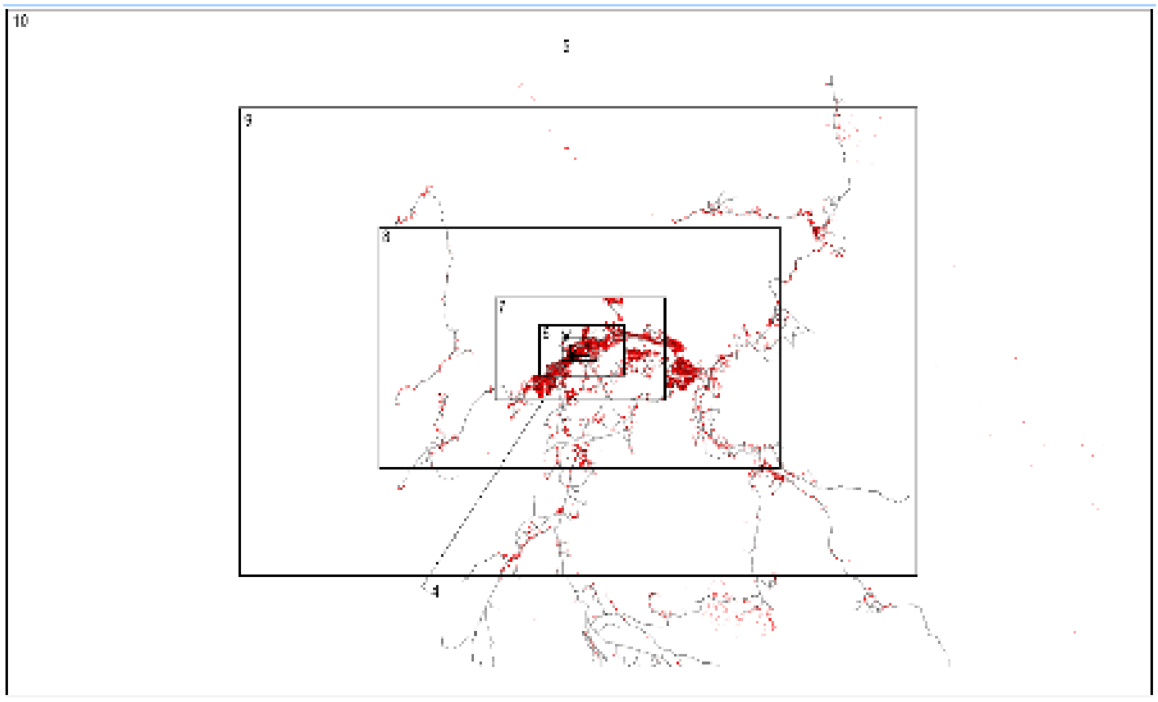


Figure 5.3 A visualization of the Tana data set, with the bounding boxes of test cases 4-10 engraved

The test have been designed to both validate the results of the constructed samples with greater variance in the data sets, to gain experience with real samples, and evaluate some of the possibilities made available with spatially enabled XSLT.

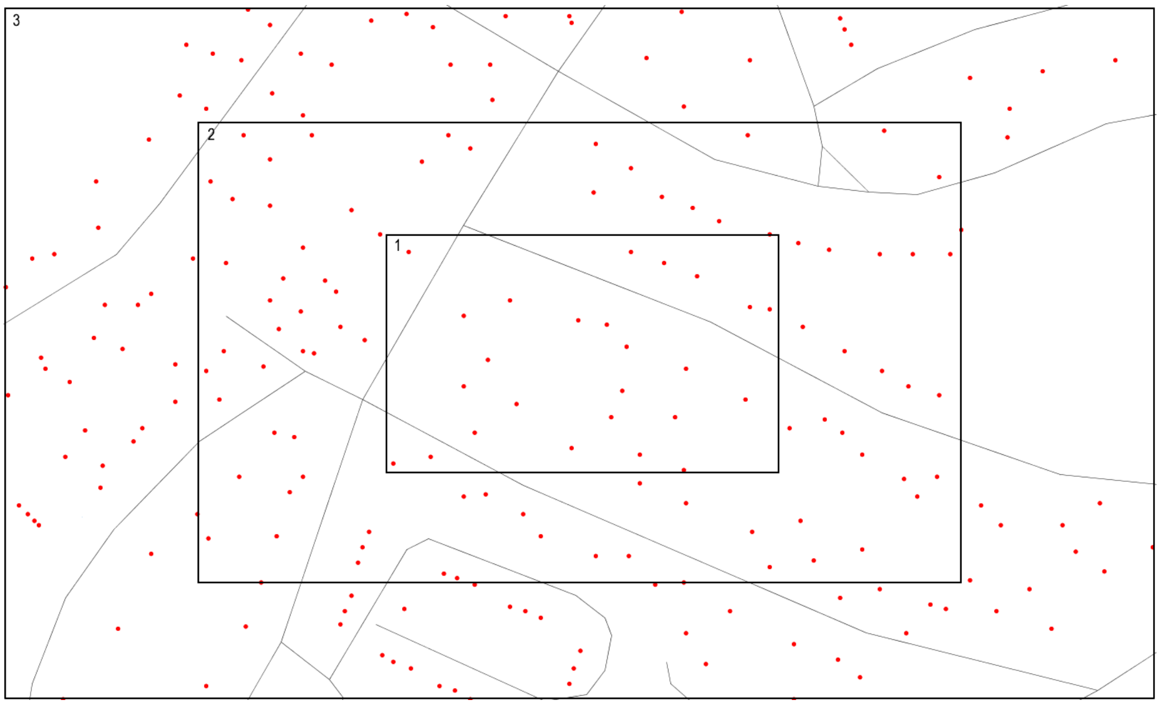


Figure 5.4 A visualization of the Tana data set with bounding boxes 3-1 engraved

The task defined for this test, is to find and count all buildings situated within 20 meters from any road. Easy as it sounds, this is a rather complex task demanding the chaining of several geo operations to be achieved. The two templates used are shown in the listings below.

```
<xsl:template match="/">
 <xsl:message>Starting...</xsl:message>

Buildings:
<xsl:value-of select="count(wfs:FeatureCollection/gml:featureMember[topp:bulroad])"/>

Roads:
<xsl:value-of select="count(wfs:FeatureCollection/gml:featureMember[topp:Road])"/>

 <xsl:message>Features counted...</xsl:message>

 <xsl:variable name="roadUnion">
    <xsl:call-template name="unionRoads">
     <xsl:with-param name="roads"
       select="wfs:FeatureCollection/gml:featureMember[topp:Road]/topp:Road/topp:the_geom/*"/>
    </xsl:call-template>
  </xsl:variable>
 <xsl:message>Road union created</xsl:message>

 <xsl:variable name="roadBuffer"
              select="spatial:buffer(xalan:nodeset($roadUnion)//gml:MultiLineString,20)"/>

 <xsl:message>Road buffer created</xsl:message>

 Number of buildings within 20 meters from the road:
 <xsl:value-of select="count(wfs:FeatureCollection/gml:featureMember[
                    (descendant::topp:bulroad/topp:the_geom/gml:Point) and
                    (spatial:within(
                           descendant::topp:bulroad/topp:the_geom/gml:Point,
                           $roadBuffer/gml:*[position() = 1]
                          )
                     )
                                        ]
              )"/>
</xsl:template>
```

Table 5.3 Main template used for the practical tests with the Tana data

```
<xsl:template name="unionRoads">
  <xsl:param name="roads"/>

  <xsl:variable name="roadCount" select="count($roads)"/>

  <xsl:choose>
    <xsl:when test="$roadCount = 1">
      <xsl:copy-of select="$roads"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:variable name="fiftypercent" select="floor($roadCount div 2)"/>

      <xsl:variable name="left">
        <xsl:call-template name="unionRoads">
         <xsl:with-param name="roads" select="$roads[position() &lt;= $fiftypercent]"/>
        </xsl:call-template>
      </xsl:variable>

      <xsl:variable name="right">
             <xsl:call-template name="unionRoads">
               <xsl:with-param name="roads" select="$roads[position() &gt; $fiftypercent]"/>
              </xsl:call-template>
      </xsl:variable>

    <xsl:copy-of select="spatial:union(
                                                xalan:nodeset($left)//*[(local-name() = 'MultiLineString')
                                                                              or
                                                                             (local-name() = 'LineString')
                                                                            ]
                                                ,
                                                xalan:nodeset($right)//*[(local-name() = 'MultiLineString')
                                                                               or
                                                                               (local-name() = 'LineString')
                                                                              ]
                                              )
      "/>

     </xsl:otherwise>
    </xsl:choose>
</xsl:template>
```

Table 5.4 Divide and Conquer recursion implemented in XSLT to union a large amount of line string segments without causing stack overflow.

First, all the road stubs are unioned into one multi line string which is stored in a variable. To avoid trouble with stack overflow, a classical divide and conquer recursion strategy inspired by (Sedgewick 2002; Novatchev and Tyszko 2006) has been applied with the "unionRoads" template. The resulting union of roads is then used in a buffer operation, creating a multi polygon centered on the road. The number of buildings within the road buffer is then matched by the XPath statement enclosed in a count function, matching all building representations having a point property which is within the road buffer. Note the use of standard extension function "xalan:nodeset" which converts result tree fragments (Apache Xalan Community 2005) from executed templates to be used in a node list context.

As the set of data is quite large (ca 6 MB), just its share size indicates that it is ill suited for use in a web context with regard for acceptable user waiting time caused by both processing and network transmission (Nielsen 1994; Nah 2004). Reducing the bounding box will result in both a smaller amount of data to transfer and less features to process. By scoping the set down to various smaller sizes represented by smaller bounding boxes we try to evaluate the performance and find the threshold for when the extended XSLT approach can be feasible in an interactive web/networked context as well as practical limits.

The data has been imported into Geoserver and extracted as WFS calls to assure realistic formats and full schema compliance. See appendix for samples of data.

5.1.5 Testing basic performance optimization

The tests with real data have been repeated with the Geometry Cache introduced in Section 4.5.5 applied, measuring and comparing the performance gain with that of the base implementation.

5.1.6 Test platform

The tests have been performed by executing a JUnit test case from within the NetBeans 5.5 beta 2 IDE on Windows XP. The hardware used was a Lenovo T60p laptop, with 1GB of RAM and an Intel Centrino Duo processor running at 2 GHz.

5.2 Chapter summary

This chapter has presented the different aspects under focus and scenarios designed to test them. Examples/visualizations of both tests and data sets have been given. The material created has been tested and forms the source of the findings presented in chapter 6.

6 Findings

This chapter presents the findings from the tests defined in chapter 5. First, the average performance times for the different operations and geometries on constructed data are presented in 6.1.1, 6.1.2, and 6.1.3. Summaries of all findings from tests with constructed data are presented in 6.1.4.

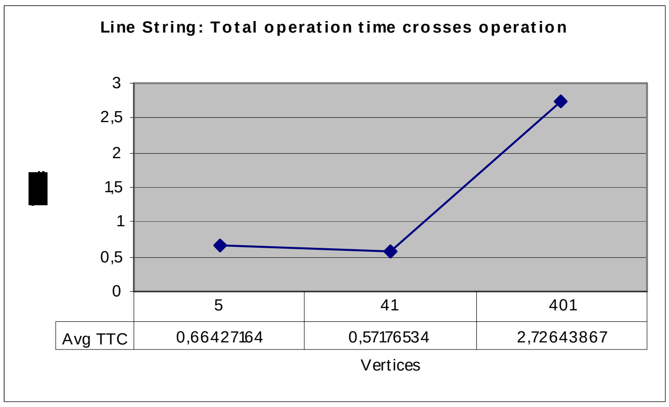
The results from practical tests performed on real data from the Tana data set (introduced in 5.1.4) are presented in 6.2.1. The tests have been performed using both the base line and optimized approaches as presented in Sections 4.5.5, 5.1.4, and 5.1.5.

6.1 *Findings from tests with constructed data*

The aim of the test process with constructed data has been to uncover correlations between the different parameters and changes in the time needed for execution of the extension functions. The following is a presentation of findings found to be of significance, and a walkthrough of their indications.

Unless noted, all times are in milliseconds. TTC is short for “time to complete operation”. Results from the tests can be used for uncovering strengths and possible bottlenecks.

6.1.1 Total execution time predicate operations

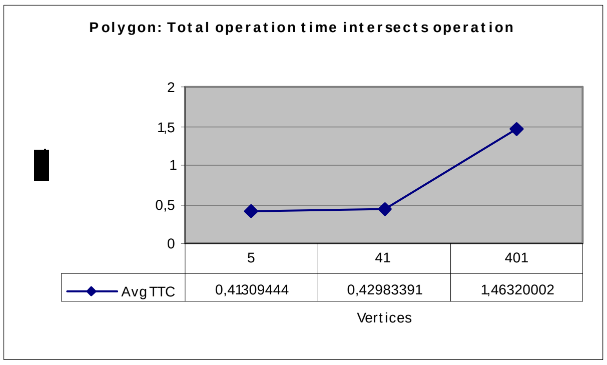


	5	41	401
Avg TTC	0,66427164	0,57176534	2,72643867

Figure 6.1 Total execution time for crosses on line string as the number of vertices increases

Results from measurements of total time spent for predicate operations on line strings and polygons indicate that the number of vertices has an impact on time needed for processing; the time increases as the amount of vertices grows. Still, Table 6.1 shows a higher effi-

Figure 6.2 Total execution time for intersects on polygon as the number of vertices increases.



	5	41	401
Avg TTC	0,41309444	0,42983391	1,46320002

ciency at higher volumes of vertices. This contributes to the impression that the GeoXSLT framework scales reasonably well.

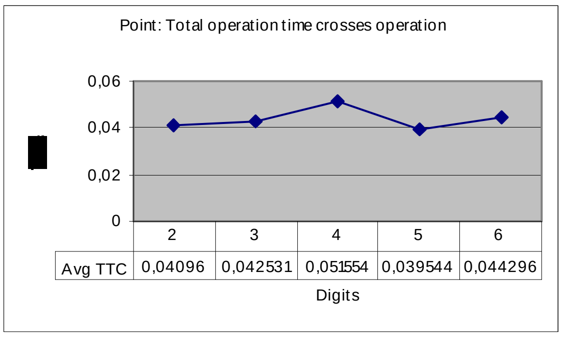
The ratio between the number of vertices and time to complete (vertices/ttc) clearly shows an increase of efficiency in the number of vertices processed pr unit of time as the number of vertices pr call increases up to 400 vertices in Table 6.1.

Vertices	Vertices pr ms
5	12,10376978
41	95,38568048
401	274,0568579

Table 6.1 Vertices processed pr millisecond, efficiency increases as amount of vertices grows

This can be an indication of the initial start-up cost for starting the conversion process from node list to JTS Geometry, and that the overhead of using the bridge for JTS construction gets less significant as the amount of vertices increases.

Variation of digits/coordinate length



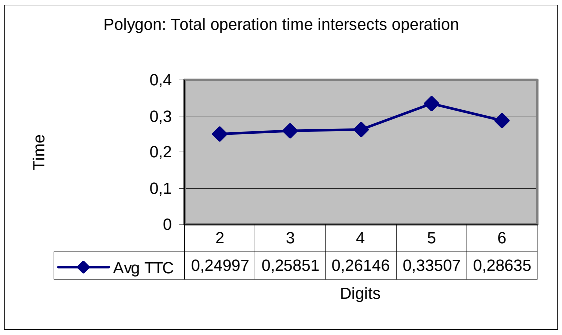


Figure 6.3 Variation of digits on crosses operation on points. Figure 6.4 Variation of digits on intersects operation on points.

The results based on tests with the crosses and intersects operations indicate that the number of digits have a negligible effect on the total time for the execution of predicate operations.

Variation of holes in polygons

The results from running the “intersect” predicate operation on polygons with a varying number of holes indicate that holes are complex and expensive to process.

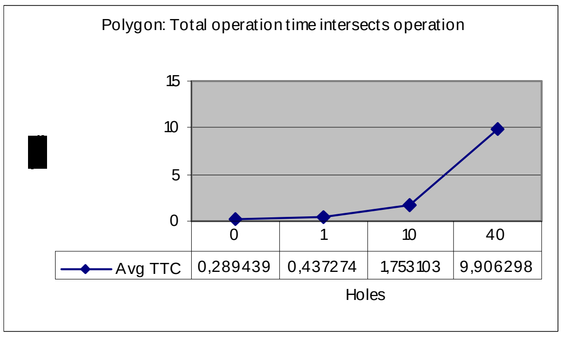


Figure 6.5 Relation between number of holes in a polygon and the time needed for executing the “intersects” predicate operation.

The holes were all of equal size, with a diameter of 20 and 10 vertices. Table 6.2 lists the ratio between the number of holes and total execution time (holes/ttc) as the amount of holes increases.

Holes	Holes/TTC
1	2,286898036
10	5,704172347
40	4,037835243

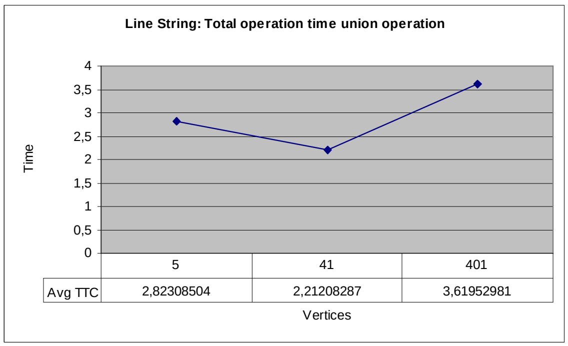
Table 6.2 Changes in ratio between Time To complete and number of holes

Based on both the table and figure, it seems that increasing the number of holes reduces the efficiency pr hole. Figure 6.18 shows how most of the time spent for predicate operations involving polygons with holes is used by actions on the geometry object. This may be interpreted as that the geometric complexity of polygons with many holes demands more resources for predicate calculations than the build process of creating the JTS geometry it self.

6.1.2 Total execution time analysis operations

Analysis operations seem to be generally more expensive than predicate operations. This is intuitive, as the category of operations demands more accurate results than true/false, and often involve the extra step of outputting and serializing a result geometry.

Variation of vertices



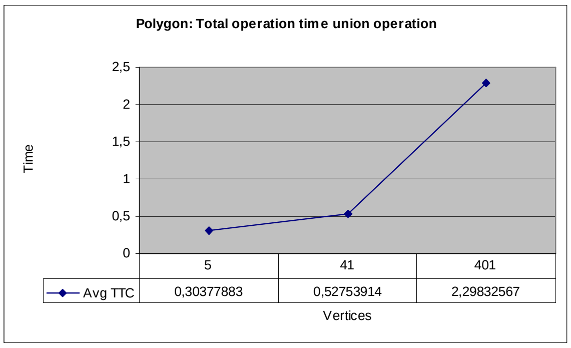
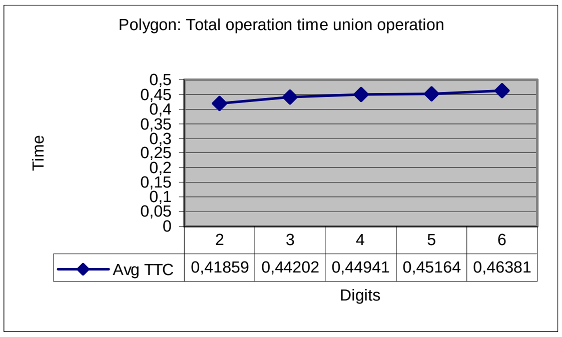


Figure 6.6 and 6.7 Correlation between number of vertices and total operation time for union operations on line strings and polygons

Execution times for analysis operations as represented by the union operation on line strings and polygons generally follow the same pattern as for predicates, but with a higher cost. In the graphs above the vertices have been varied between 5 and 401.

Variation of digits

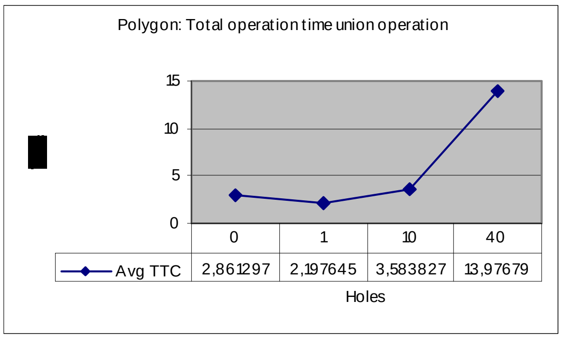
Observations indicate that the number of digits have a relatively small impact on the time cost for union operations on polygons. This adheres to the results from tests of predicate operations with variations of digits.



Digits	2	3	4	5	6
Avg TTC	0,41859	0,44202	0,44941	0,45164	0,46381

Figure 6.8 Variations of digits and holes for union operations on polygons

Variation of holes



Holes	0	1	10	40
Avg TTC	2,861297	2,197645	3,583827	13,97679

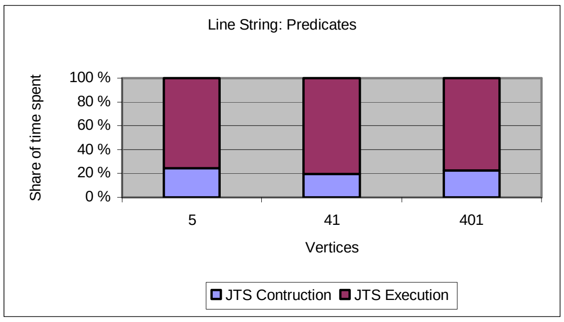
Figure 6.9 Variations of holes for union operations on polygons

Polygons with holes are expensive to process, as originally shown by the predicate observations.

6.1.3 Division of Total Time To Complete between the different operations

The graphs in this section show how time spent is divided between construction, calculation, and serialization phases. The graphs represent the most interesting findings because they give indications on the cost of integrating spatial functionality with XSLT relative to the overall time and calculations performed. As such, they make up the foundation for deciding where performance optimization can be applied with the most effect. The findings here need to be viewed inline with the total time spent on each operation. For example, for points, the relative time spent for constructing and serializing is large, but the total time of the whole operation is significantly smaller than for that of line strings and polygons.

Line Strings



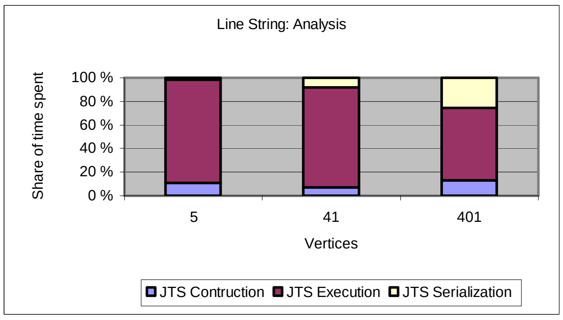


Figure 6.10 Variation in relative time spent on JTS construction for predicates on line strings

Figure 6.11 Variation in relative time spent on JTS construction for analysis' on line strings

For predicate operations on line strings, the time cost of XSLT integration is generally around 20%. For analysis operations, the time to serialize the results back to a node for use in the XSL process causes the time cost to increase more. Increases in the number of vertices gives for substantially longer time needed to serialize the result, seeming to stabilize at around 25% of ttc for serialization and 10% for construction. In total, the results indicate a time cost of ca 20%-35% for the integration of analysis operations on line strings with XSLT, depending on the number of vertices involved.

Points

The indication that the start-up cost is high relative to the calculation done on the JTS Geometry is confirmed when the division of time between construction and calculation is presented here. The cost for construction can be expected to be between 70-80% for predicate operations on points, regardless of variance in digits. The high relative cost should be seen in view of the short total time spent for point operations compared to operations on other geometries. Section 6.1.1 and 6.1.2 provide comparisons and visualization of measured time. As an example, the average total time for performing the crosses operation on points with 6 digits is ca 0.04 ms, while it

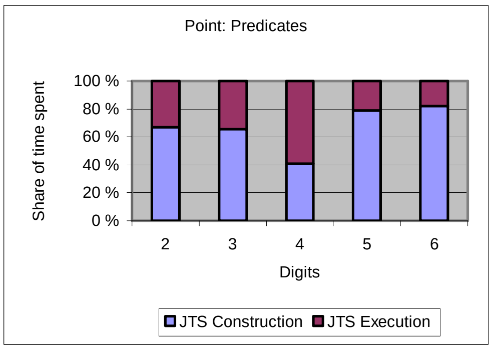


Figure 6.12 Variation of digits, relative cost of construction for points in predicate operations

for polygons is ca 0.29 milliseconds. Much of this difference can probably be explained by a *minimum time* needed for starting up the process for building the JTS geometry. When the JTS calculation time is so small, the build time becomes more significant.

No operations have been executed as part of the test cases which gives an analysis operation on two points with point return. As the next best thing a dummy operation which just builds geometries based on point input for immediate serialization is used to provide the following ratios between JTS build and JTS serialization for points.

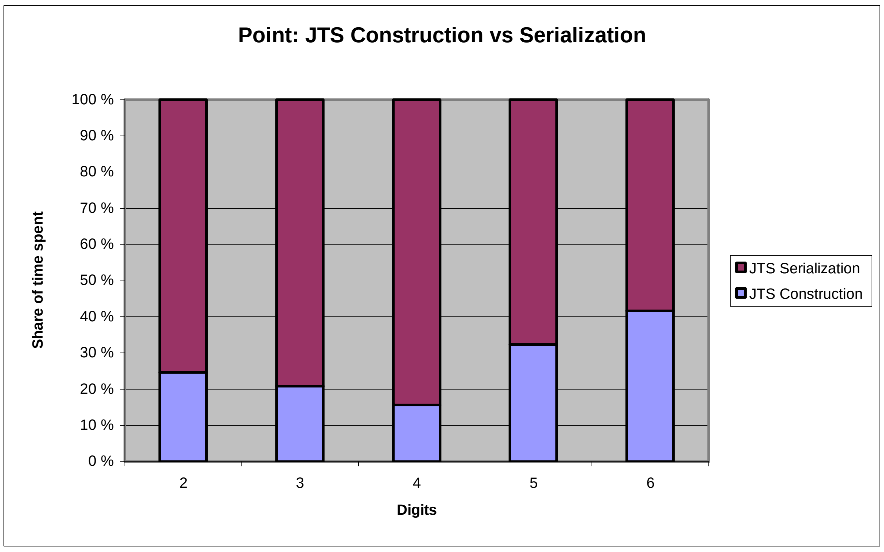


Figure 6.13 Construction time relative to serialization time for points, variation of digits

The results indicate that for points, serialization is clearly more expensive than construction with regard to time cost, but as with the results of relative times for predicate operations on points, it is important to keep in mind the short total time compared to that of other geometries.

Polygons

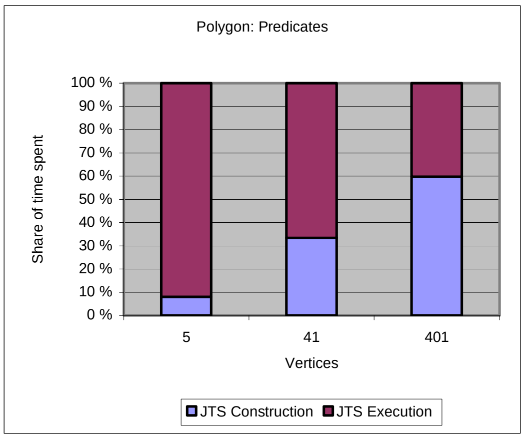


Figure 6.14 Polygons; relation between construction and calculation phases in predicate operations as the number of vertices is increased.

This section presents the relative times of the construction, calculation, and serialization phases for polygon geometries.

According to these results, the relative time cost for constructing JTS geometries is not stable with regard for variations in vertices.

While the cost is relatively stable around 20% for line strings, it increases from 10% - 40% for polygons as the number of vertices is increased. This may be due to a more complex build process when the parsed coordinates are passed to the JTS Geometry factory while the calculation phase has more constant time consumption. Analysis operations on complex polygons are something that could probably benefit from more detailed research to uncover possible bottlenecks in polygon creation.

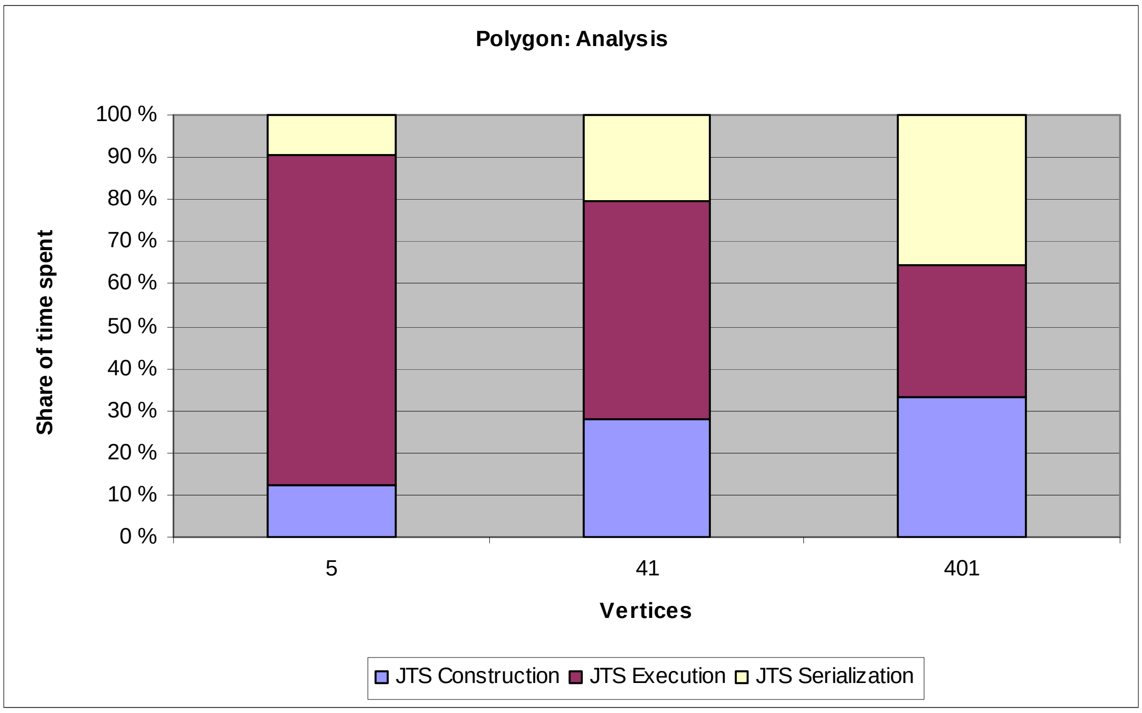


Figure 6.15 Polygons; relation between construction, calculation, and serialization phases as the number of vertices is increased

The pattern found in the construction phase is similar in the time required for serialization. At the 401 vertices observation of Figure 6.1 above, the total required time is at its worst with the construction and serialization phases when combined representing ca 75% of the time cost for performed operations.

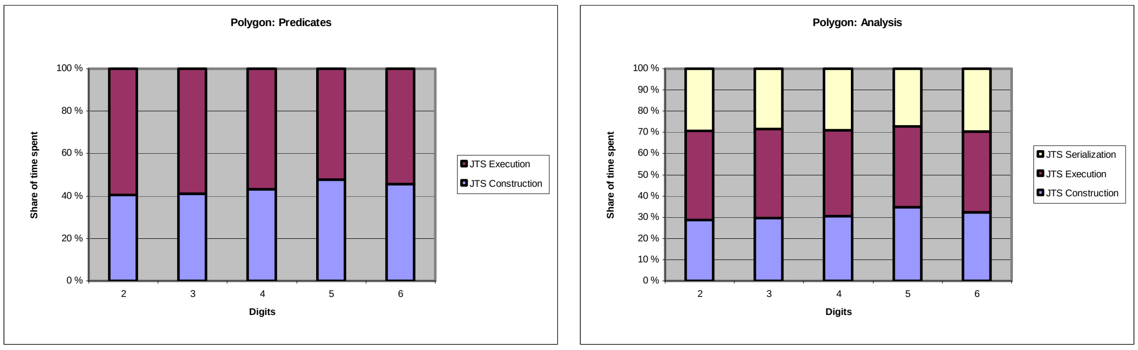
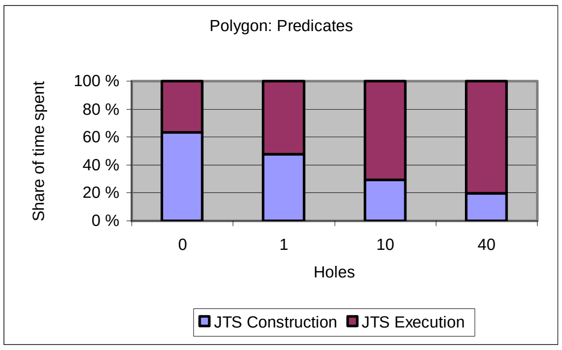


Figure 6.16 Polygons; relative cost of construction in predicate operations with variance of digits

Figure 6.17 Polygons; relative cost of construction and serialization with variance of digits

The time cost for predicate operations on polygons is relatively high but stable at around 40% relative to the total time to complete. This is significantly more expensive than the 20% for line strings, but logical, as the construction process for polygons is more complicated than that of line strings. The serialization in an analysis perspective is also stable, and levels off at around ca 30% as shown in Figure 6.17. The total time cost averages out at 60% in total relative to the ttc for the integration of analysis operations on polygons in an XSLT context.



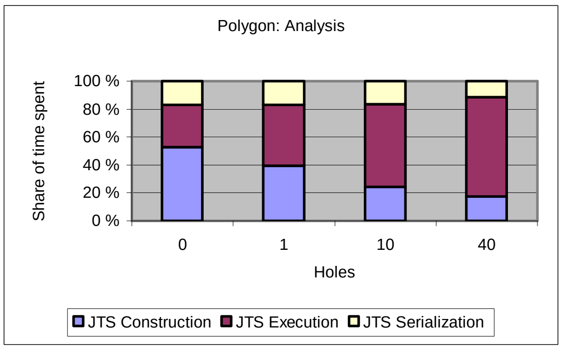


Figure 6.18 Polygons; relative cost of construction in predicate operations with variance of holes.

Figure 6.19 Polygons; relative cost of construction and serialization in analysis operations with variance of holes.

As observed in the diagrams for operation running time, the processing of holes in a polygon is expensive with regard to time cost. These observations of time as divided between calculations and construction/serialization confirm that indication. The evident decline in relative construction time as the number of holes increases attests how the construction/serialization processes become less important factors in the ttc as the number of holes increases.

6.1.4 Summary of findings from tests with constructed data

Parameter	TTC	Construction	Calculation
Vertices			
3	0.66	24.66%	75.34%
30	0.57	19.88%	80.12%
300	2.73	22.47%	77.53%

Table 6.3 Predicate performance on line strings

Parameter	TTC	Construction	Calculation	Serialization
Vertices				
3	2.82	10.80%	87.42%	1.78%
30	2.21	7.16%	84.81%	8.03%
300	3.62	12.87%	61.78%	25.35%

Table 6.4 Analysis operations on line strings

Parameter	TTC	Construction	Calculation
Vertices			
2	0.04	67.06%	32.94%
3	0.04	65.47%	34.53%
4	0.05	40.83%	59.17%
5	0.04	78.76%	21.24%
6	0.04	82.26%	17.74%

Table 6.5 Predicate operations on points

Parameter	TTC	Construction	Serialization
Vertices			
2	0.05	24.64%	75.36%
3	0.05	20.83%	79.17%
4	0.07	15.62%	84.38%
5	0.04	32.33%	67.67%
6	0.03	41.60%	58.40%

Table 6.6 Construction/Serialization for points

Parameter	TTC	Construction	Calculation
Vertices			
5	0.41	7.96%	92.04%
41	0.43	33.39%	66.61%
401	1.46	59.77%	40.23%
Digits			
2	0.25	40.57%	59.43%
3	0.26	41.14%	58.86%
4	0.26	43.11%	56.89%
5	0.34	47.63%	52.37%
6	0.29	45.71%	54.29%
Holes			
0	0.29	63.22%	36.78%
1	0.44	47.49%	52.51%
10	1.75	29.29%	70.71%
40	9.91	19.73%	80.27%

Table 6.7 Predicate operations on polygons

Parameter	TTC	Construction	Calculation	Serialization
Vertices				
5	0.30	12.38%	78.34%	6.09%
41	0.53	27.75%	51.73%	27.05%
401	2.30	33.37%	31.32%	61.12%
Digits				
2	0.42	21.91%	55.82%	40.95%
3	0.44	21.76%	57.27%	40.25%
4	0.45	23.34%	54.62%	41.71%
5	0.45	27.58%	50.67%	41.78%
6	0.46	25.52%	51.15%	43.48%
Holes				
0	2.86	11.13%	85.29%	35.55%
1	2.20	14.53%	79.16%	28.20%
10	3.58	16.48%	72.22%	22.12%
40	13.98	14.33%	76.27%	13.88%

Table 6.8 Analysis operations on polygons

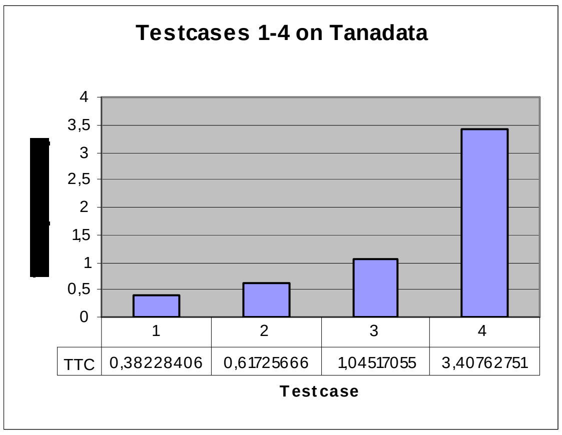
6.2 *Findings from tests with real data*

This section presents the results of the tests performed on the Tana data set.

The baseline approach presented first, uses the implementation of the operations as tested with the constructed data and hence processes everything straight forward. The Optimized approach takes advantage of caching geometry objects which are used many times, and as such saving most of the overhead used to construct the JTS geometry repeatedly.

The description of the tests done can be found in Sections 5.1.4 and 5.1.5.

6.2.1 Base-line approach



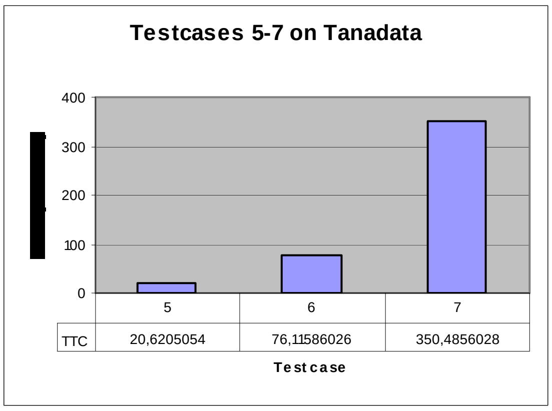
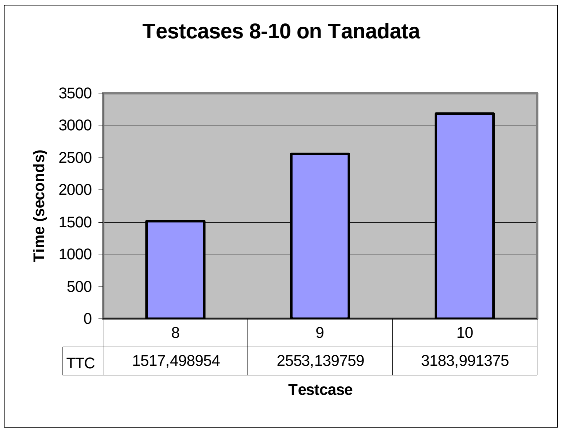


Figure 6.20 Performance times for tests 1-4 on Tana data.

Figure 6.21 Performance times for tests 5-7 on Tana data.



	8	9	10
TTC	1517,498954	2553,139759	3183,991375

Figure 6.22 Performance times for tests 8-10 on Tana data.

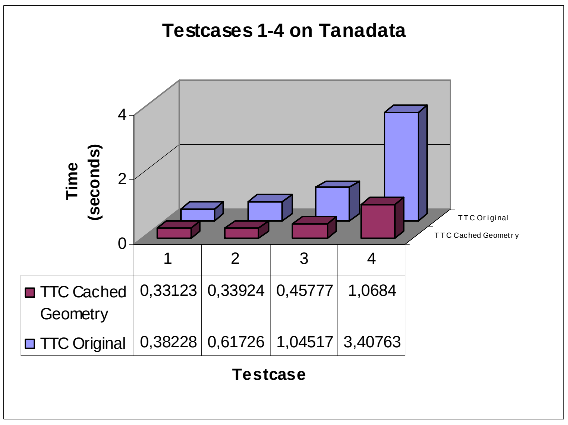
The practical test of counting houses within 20 meters from the road has been executed on all ten scenarios. Note that the results are shown in seconds as opposed to milliseconds used on the diagrams covering results on isolated operations for constructed data. See 5.1.4 for descriptions of the test, scenario details and visualizations of the Tana data. The results show that using a non-optimized approach, things start to slow down somewhere between case 4 and 5 with an ever increasing time cost as the data set gets larger. It is important to be aware that the data sets given by the different bounding boxes are not linear in growth. The discussion around performance thresholds can be found in chapter 7.

To understand the reasons behind the growth, it is necessary to analyze the stylesheet defined in 5.1.4 and the spatial extensions used in view of the test results from singular operations on constructed data. As earlier mentioned, the templates use a recursive approach to union all roads into one multi line string which is then used to construct a buffer representing the 20 meter zone as explained in Section 5.1.4. A major source of time consumption is the repeated use of the "within" function as shown in Table 5.3. The within function is used to check if a point (building) is within the 20 meter zone (buffer) of the road. Each time the function is called, the same buffer variable needs to be converted to a geo object using the GeoXSLT framework. No efficiency is gained by reusing geometries. The within function is called up to 9848 times depending on the number of buildings (points) in the data set with large polygons representing the buffer (Table 5.2 presents the different test cases). This shows that the baseline implementation of the GeoXSLT framework has a weakness when it comes to repeated use of geometries.

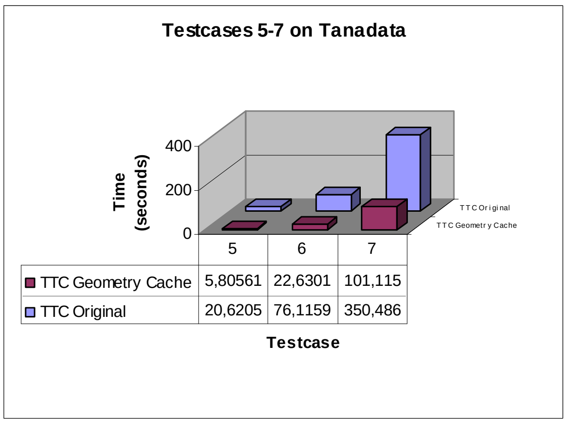
6.2.2 Tests of basic performance optimization

A through optimization of the JTS construction process has been outside the scope of this thesis. This section presents tests with the “Geometry Cache” mechanism as presented in Sections 4.5.5 and 5.1.5. The Geometry Cache minimizes the time needed for repeated construction of geometries already used by keeping ready built geo objects at hand. This section presents the findings from running the tests with Tana data (Table 5.2 presents the different test cases).

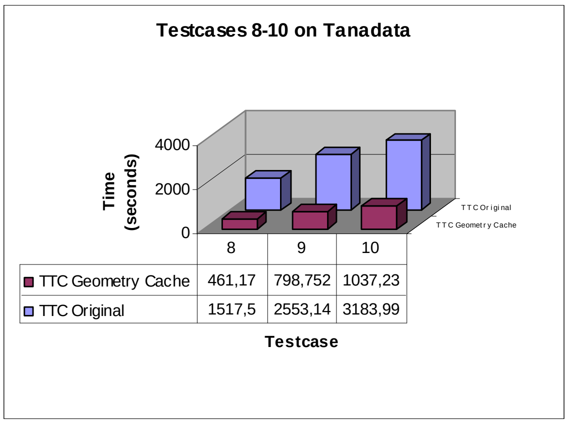
Visualizations of execution times and performance increase relative to the base line approach are shown in Figure 6.23.



Testcase	1	2	3	4
TTC Cached Geometry	0,33123	0,33924	0,45777	1,0684
TTC Original	0,38228	0,61726	1,04517	3,40763



Testcase	5	6	7
TTC Geometry Cache	5,80561	22,6301	101,115
TTC Original	20,6205	76,1159	350,486



Testcase	8	9	10
TTC Geometry Cache	461,17	798,752	1037,23
TTC Original	1517,5	2553,14	3183,99

Figure 6.23 Performance increase when using “Geometry Cache” on Tana data

In general, the test cases gain a 60 % performance increase by using the Geometry Cache.

6.3 Chapter summary

This chapter has presented the results of the tests defined in chapter 5 and provides an overview of performance and how the bridging between JTS and the XSLT process affects it. Further it has shown experiences with real data, and how a simple caching mechanism can be used to increase performance significantly.

7 Discussion

This chapter discusses the research objectives in view of the findings and previous research. The research objectives are answered as follows:

Research Objective 1

To answer research objective 1, section 7.1 presents a line of arguments concluding with the general suitability of XSLT with Extensions for the processing of GML.

Research Objective 2

Performance

Section 7.2 gives a discussion on the general performance issues and aspects, before detailing on the various phases of constructing, calculating, and serializing geometry representations in an XSLT context.

Flexibility

As an example on the flexibility and feasibility of the system implemented, section 7.3 features argumentation and references to practical examples on why and how generalization/simplification can be performed with XSLT, shedding a new view to the conclusions of previous research. The demonstration around integration of simplification operations in Section 7.5 also shows the ease with which new functionality can be integrated. It also presents suggestions for new enhancements opening up to even more power for the transformation process.

Research Objective 3

Section 7.4 gives a walkthrough of the simplification of the development process achieved by bringing spatial functionality to XSLT.

The chapter is ended with a walkthrough of a practical application of the GeoXSLT framework where the features discussed are demonstrated in Section 7.5.

7.1 The suitability of XSLT with extensions for processing Geo-data

Research objective 1 places a focus on the possibility of using XSLT with extensions as a medium for processing geospatial data. The experiment done here is based on the assumption that data is of an XML format with at least the geo part of features encoded inline with the GML schema (herein the Simple Feature Specification) as specified by Open Geo Spatial/ISO (Ryden 2005). Research objective 1 as defined in 1.1 is hence interpreted as a question regarding the technical possibility of processing GML and performing spatial operations on the data from within the XSLT process. As GML is a XML format (Cox, Cuthbert et al. 2002; OpenGeoSpatial 2002; Lake 2004), and XSLT is a stylesheet language for XML (Clark 1999), there is no technical reason why XSLT cannot work with GML. Non-spatially aware processing of GML with XSLT has also been documented possible by other research papers (Harrie L. 2003). General extension of XSLT is defined by W3, and support for developing such extensions is provided in major processors. The integration of spatial functionality through extensions of the XSLT language has previously been introduced by Lehto and Sarjakoski (Lehto and Sarjakoski 2005) as a field open for more research. Harrie and Johansson refute the use of XSLT/extensions for geospatial operations due to the inherent lack of “object interaction” in XSLT (Harrie L. 2003). While the re-iteration capabilities of XSLT are not as evident as those of e.g. XQuery, Object interaction and contra indications on the lack of such in practice are discussed with closer detail in 6.3.1. In general, the system created as part of this thesis provides an implementation of XSLT extensions for certain geospatial functions implemented in Java. The implementation has been tested with subsequent interesting findings confirming that technically, it is very possible to extend XSLT with spatial capabilities. A certain time overhead for conversion of data between the stylesheet and processing extensions must be expected, but as discussed in 6.2 and 6.4.2, the cost can be acceptable and in some cases outweighed by the potential performance gained by less wait for network transmission time. These findings give a clear indication that XSLT with extensions certainly can be used with success for transforming and processing spatial data encoded as GML.

7.2 Performance; Limitations and possibilities

This section evaluates the findings from developing and testing the extension framework. Section 7.2.1 discusses the performance of singular operations and how the total time is divided between the different phases of the extension functions. It also presents some aspects of the construction process that do need improvements. Section 7.2.2 focuses on the performance in practical settings with real data. Findings are interpreted and discussed in view of view of different perspectives on time. This section also discusses enhancements to improve general performance of the implemented system, and evaluates the significant effect of the “Geometry Cache” introduced in Section 4.5.5.

7.2.1 Performance of singular functions

The findings from measuring the performance and workings of specific extension functions in chapter 6 provide clear indications that there is a given overhead for the two-way conversion of data between XSLT and JTS. This section attempts to identify and analyze the reasons for the overhead cost to open up for further improvements and patterns of use. The processing time of an extension function is made up of two or three phases depending on the return type of the operation. The construction phase, where JTS geometry objects are created from node lists; the calculation phase, where calculations on the JTS objects are executed; and the serialization phase, where results from the calculation are converted to node lists for return to Xalan.

Construction phase

In a predicate context where the result is a Boolean value without need for serialization, the construction phase consumes ca 20% of the total execution time for line strings and 55-88% for points. For polygons, the cost is relatively stable at 40%-50% for geometries without holes, and varies between 10%-60% for polygons with holes. As operations on polygons with holes generally seem to take the most time to complete, we will focus them.

Predicate operations		**Polygon**	**Line String**	**Point**
	Vertices	1.46320002	2.72643867	
	Digits	0.28635205		0.04429613
	Holes	9.9062982		
Analysis operations				
	Vertices	2.29832567	3.61952981	
	Digits	0.46381313		0.03397368

Holes	13.97678751

Table 7.1 Maximum values for the time to complete for all test cases performed on the constructed data as described in chapter 5 and 6. Times are in milliseconds.

Table 7.1 shows the maximum recorded values for all test cases on constructed data. Details, visualizations, and parameter values are available in chapter 6, which presents the findings of the tests specified in chapter 5.

Polygons

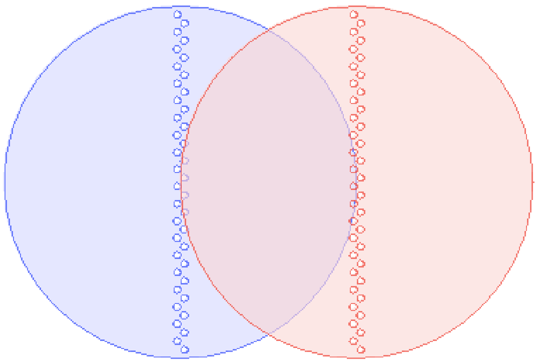


Figure 7.1: Polygons with 40 holes

For polygons with holes, it is interesting to note that the fluctuation is very visible as shown in Figure 6.19 Polygons; relative cost of construction and serialization in analysis operations with variance of holes. In the diagram one can clearly see that the time cost of the construction phase decreases relative to the calculation (JTS Execution) phase as the number of holes increases. This is interesting as the holes used in the polygons are identical; it is only the number of instances that is increased. In other words, increasing the number of holes in a polygon seems to be more expensive with regard to the calculation phase than the construction phase. The results from chapter 6 indicate that in general, operations on polygons with holes require the longest time to execute (build, calculation, and serialization combined). This is also shown in Table 7.1. While optimizing the calculations of JTS has been outside the scope of this thesis, it is of interest to understand the reasons behind the costs of constructing the JTS geometry object. By such an understanding one is able to provide input for improvements of the build process. The observation that the construction phase is relative to the calculation phase more expensive for smaller amounts of holes brings us to the impression of a start-up cost for geometry parsing that does not scale down very well for polygons with holes in particular. The build phase is implemented in the level 3 classes which are documented in 4.5. As documented with closer details in 4.5, the construction phase is organized as a loop iterating through the children of the node list as received from Xalan. The first node found to be a GML node is then sent to a “SubHandlerFactory” for delegation of the correct handler. What makes the polygon handler differ from other handlers is that while points and line strings can extract the coordinate string for direct parsing, the polygon handler has to execute two additional searches within its own node list. The first search is for the singular “outer boundary”, and the second for a possible series of “inner boundaries”; holes. For each linear ring found in the outer boundary or inner boundaries a sub handling equally expensive as for the line strings or points has to be executed. This means that for polygons, the construction phase involves two searches and the creation of minimum one basic geometry instance (linear ring for the outer boundary) in addition to the processing in common with other simple geometries. On top of this, a validation and possible rebuild of the outer and inner rings is done to bridge between the difference between JTS and eventual conformance with the polygon definition of ISO 19107 (ISO/TC211 2003) in the data. ISO 19107 defines polygons to have counter clockwise outer rings and clockwise holes. This is understood to be necessary for some algorithms and bearing calculations where it is necessary to know the up and down of a polygon. The GML specification references the ISO 19107 and states that it should be conformant . The SFS-SQL specification (ISO 19125-2) (Ryden 2005) enforces no rule on this. In JTS this “normal form” is defined inversely, with outer ring clockwise and holes counter clockwise (Davis 2003) (Köbben 2005). While I have not timed the cost of this validation/rebuild process, it could be worth investigating how expensive it is with regard to time, and if polygon parsing in the construction phase should assume that the coordinate sequences of outer and inner rings are provided in a valid/conformant state for JTS normal form. It is possible to convert JTS polygons to JTS normal form with the provided normalize method (Davis 2003; Davis 2004), but as noted in (Köbben 2005) and in Javadocs, it returns rings in a CW-CCW fashion, - which is the opposite of the ISO standard. The maintainer of JTS seems reluctant to change this according to (Köbben 2005). The other possible area of high cost in the polygon build process is the mentioned searches executed to find the linear rings of inner and outer borders. As the two searches are executed on the same node list, - which has already been iterated (shallowly) by the original search detecting the polygon, it could very well be that time and resources could be saved if either the original search was able to do a deep search, or at least that the two searches executed in the polygon handler were combined into one. By doing this we can assure that none of the nodes in the list are read more than once, and also avoid the possible start-up overhead of initiating three searches instead of one. The issues around rationalizing the number of searches for sub geometries are also of relevance for the performance of multi geometries, as these are made up of a number of basic geometries as “members” of the complex geometry in the same way as a polygon has “children” that define its multiple geometric properties.

Possibilities of improvement for the construction phase

Coordinate parsing

The parsing of pure coordinate strings is common for all geometries and is a field that also has areas for improvement. While use of the “nodeSearch” function to find the coordinate node is explained in 4.5, it is quite similar in logic to the various gmlSearch methods used with polygons as described in Chapter 3. This implies that also this method could be marginalized by allowing the original iteration of the node list to detect the node element containing the coordinate string.

Generalization and use of static types

During testing, a relatively high spread in the sampled observations was detected. This can to a certain degree be explained with the not ideal test-platform which was vulnerable for among many things Windows’ up and down prioritizing of threads while running (several non related utilities were running on the computer at the time of testing). But also the fact that the code has several redundant class instances local to the various handlers. This can possibly result in a poorer runtime performance with more variance in execution times. A major improvement, both with regard to the execution speed and memory use would be to do a refactoring process of the handlers and generalize out common class instances and to a certain degree place them in static contexts. Another advantage of this would of course be the general improvement in code quality and readability.

Calculation phase

The calculation phase consists of calling the corresponding methods on the generated JTS object of the extension function called. Performance measured relative to the other phases is available in section 6.1.3. In this thesis, no optimizations or changes to the JTS methods have been implemented. Even so, it should be of no problem to use various functionality of for calculation optimization by calling or setting those properties from the level 2 classes were the calculation methods are called. The technical and developer references of JTS both mention several approaches for increased performance (Davis 2003; Davis 2006).

Serialization phase

The serialization of JTS objects to Node objects compatible with Xalan is a straightforward process documented in 4.5.4. Findings from serialization of line strings, points, and polygons in Chapter 6 indicate that as the series of coordinates or complexity of the geometry to serialize increases, the time needed for serialization increases as well. This correlation between data to output and required time to serialize, does not necessary match the precision level or number of vertices in the calculation; as it is the complexity of the resulting geometry that gives the size of the result to serialize. But, if e.g. the precision level or number of digits is high, it will make the impact of a large and complex result geometry more powerful. The implementation does have shallow prototypes of node objects representing various geometry objects for quicker serialization by cloning instead of building them from the ground up. But this collection could be more elaborate. With shallow it is here meant that the prototypes to a very little degree are singular node elements with namespace, prefix and name set. The encoding process has to clone each node, insert eventual values (such as coordinates in a coordinate node), and then attach the nodes together. By having deeper prototypes; nodes which have all the components required ready available instead of being singular entities, the time required to assemble the serialized nodes can be reduced by an unknown margin. This would probably be of most benefit to those cases were the serialization phase is large relative to the other phases. Such cases are serialization of points and larger polygon and line strings.

7.2.2 Execution performance in practical contexts

The implemented system for enabling spatial transformations is assumed to be used as a component in applications working in mainly two distinct perspectives. One is the interactive perspective defined to be a setting where someone or something to a certain degree expects immediate response to operation calls. Examples are standalone applications or browser based clients accessing data directly, and server applications such as web services using the library to provide some kind of middleware data transformation or processing for external applications, such as in (Lehto and Sarjakoski 2005). (Nielsen 1994;

Nah 2004) are some of the many papers providing guidelines and categorizations of tolerable waiting time. A presentation of the categorization as adopted in this thesis is presented in section 6.2.2.1 below. The other perspective represents typical batch operations, where the tasks are regarded as jobs to be executed independently of users or processes waiting for the results right then. In the batch context, immediate response seems to be of generally less importance, with the main focus set on throughput capabilities, job scheduling, possibilities of handling larger amounts of data, and performance gains by running several jobs (processes) in parallel to mention some.

Acceptable waiting time for interactive contexts

According to (Nielsen 1994), basic advice on response times for interactive applications has been more or less unchanged within the last thirty years. In (Nielsen 1994) Nielsen has set out limits that operations within an interactive context should adhere to provide an optimal user experience regardless of the application implementation. Below are the limits as interpreted in this thesis:

Instantaneous: 0.1 second. This limit specifies the max time an operation can take while still giving an impression of instantaneous reaction. Such response is expected for e.g. displaying letters on the screen as one's typing in a word processor or visual response when clicking on GUI components. If the operation takes more time than 0.1 second the user will notice that the computer is working on something.

Noticeable: 1.0 second. This is the max time noticeable operations can take without the user sensing the system as "sluggish". If an operation needs more than 1 second to complete, it should provide some kind of indication that it is working. (Cursor shaped as a time glass is suggested by Nielsen).

Tolerable: 10-15seconds. For general contexts, if an operation needs more than 10 seconds to complete, the user should get an indication of the remaining time before the task is completed. Still, in (Nielsen 1994; Nah 2004) it is observed that in a web-context, users can accept up to 15 seconds for a page to download, as web users have "been trained to endure so much suffering that it may be acceptable to increase the limit value to 15 s" (Nielsen 1994).

The interpretations in 6.2.1.2 review the findings in view of these categorizations and discuss how the implemented library fits into each of them.

Interpretation of findings with regard to waiting time in an interactive context

All total execution times used in these discussions are based on findings from the tests performed on the Tana data-set in 5.2 and with the cache prototype enabled. The cache was used because it yields considerably higher performance than the original/underlying implementation. See Sections 4.5.5, 5.1.5, 6.2.2, and the section about caching below for details on execution times and discussion of the cache prototype.

Transformations within instantaneous time

Given a realistic scenario with an application utilizing the library, the stylesheet would probably be loaded and ready for execution. Still, the job fetching the geospatial data (GML) to be processed typically needs to load it from sources situated on externally, such as WFS servers. Just fetching the data would in many cases probably imply a waiting time of more than the maximum of 0.1 seconds to be within the "instantaneous" limit defined in 6.2.1.1 regardless of how data is processed locally. Disregarding this, it is here assumed that the GML has already been loaded, e.g. by accessing it through a local cache, or that the application thanks to its spatial capabilities (as mentioned in 1.2) is re-iterating on data available from previous calls during its session.

The findings indicate that the relatively complicated transformation (as of the transitive template use) used in the Tana data-set cannot transform much data within 0.1 seconds.

Case	Time to complete
1	0,33122813
2	0,339242278
3	0,457767119
4	1,06840199

Table 7.2 Time needed to complete test cases 1-4 on Tana data as defined in 5.1.4 and reported in 6.2. Time measured in seconds.

Table 7.2 shows that for the smallest subset tested, consisting of 3 road stubs and 24 buildings encoded in a 16 kb file (see 4.7.4 for details on test) time was exceeded with more than 300% relative to the 0.1 second limit. Among the possible interpretations of this result is the indication of XSLT as not being an ideal approach for instantaneous transformations of data, but also that the cost of using the extensions is too high, or that the test executed is too complicated. To find out more about the smallest possible time the specific transformation can be executed on the Tana data using the test equipment (see

4.7.5), two cases with even smaller subsets have been created. These tests have only been tested using the cache-optimized implementation.

Case	Area	Roads	Buildings	Size (kb)	Time to complete
0.1		1	1	2	0,049887346
0.2		1	7	5	0,055344185

Table 7.1 Extra small subsets of Tana data to probe for minimum possible execution time on operations defined in 5.1.4

These findings indicate that the implementation has a relatively low capacity for the amount of features/size of data-set that can be processed instantaneously.

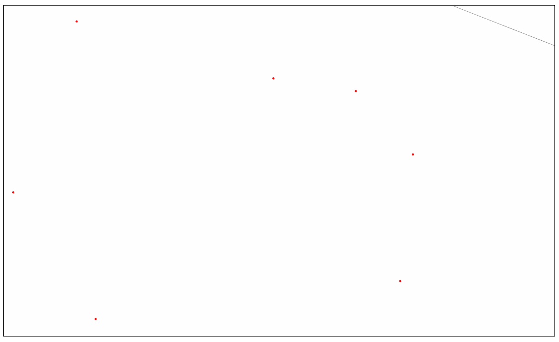


Figure 7.1 Visualization of test case 0.2.

But, as it seemingly is possible to achieve the performance with a minimum of features processed there are approaches that can possibly circumvent the challenge to a certain degree. One alternative is implementing an incremental transformation process; processing and rendering only smaller bits of the feature-set to provide a stream of updates, somewhat like how Google Earth renders and displays satellite and aerial photos as they are received in the client. This would probably incur a performance loss with regard to total processing time, due to the increased time overhead for the multiple transformations and renderings.

Possibilities for transformations within noticeable time

While it does not seem that the possibilities for instantaneous transformations are optimal, the feature capacity is quite larger when operations can occur within a 1 second time frame. Findings from the Tana data-set indicate that the complex transformations tested can be executed within 1 second for data sets a little smaller than case 4 (4.7.4).

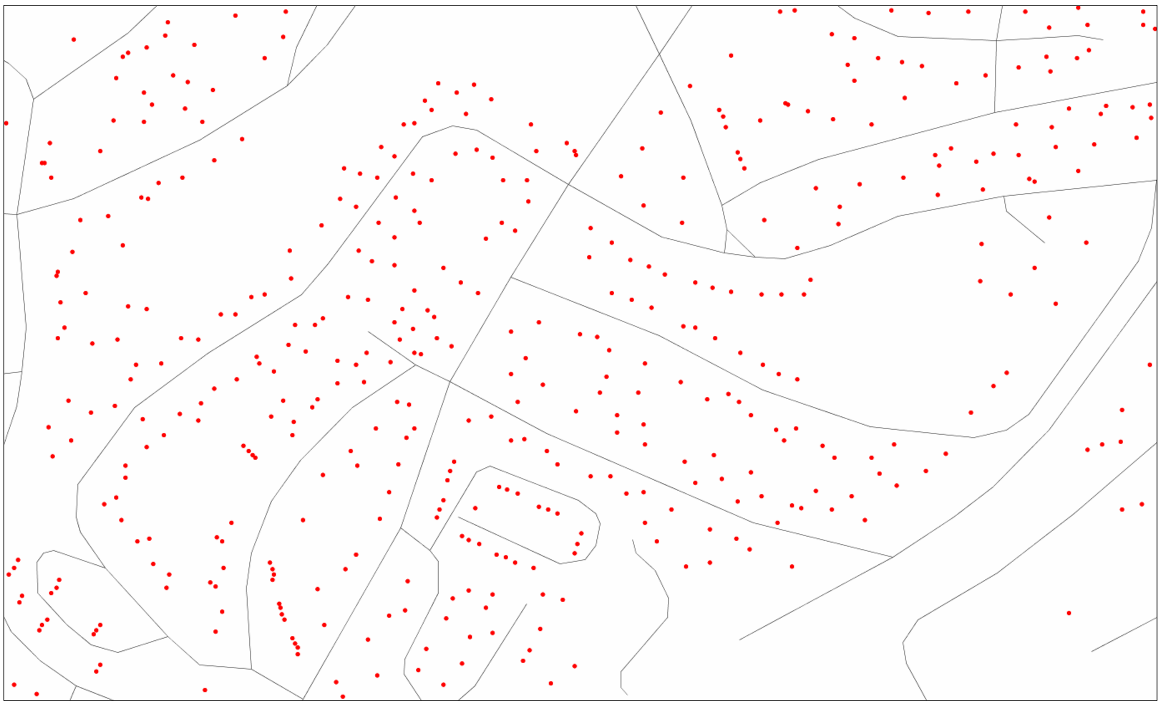


Figure 7.2 Visualization of Tana data, test-case 4

As the figure above shows, data sets of similar size as case 4 of the Tana data are large enough to be of practical use in contexts such as web mapping etc.

Possibilities for transformations within tolerable time

For transformations within a timeframe of 10-15 seconds, relatively large amounts of features can be processed. In the Tana data-set, cases 5 and 6 are performed within a timeframe in the vicinity of the limits for tolerable time.

Case	Area	Roads	Buildings	Size (kb)	Time to complete
5		201	1 132	643	5,805609651
6		425	2 145	1 242	22,63007589

Table 7.3 Tana test cases 5 and 6 with findings

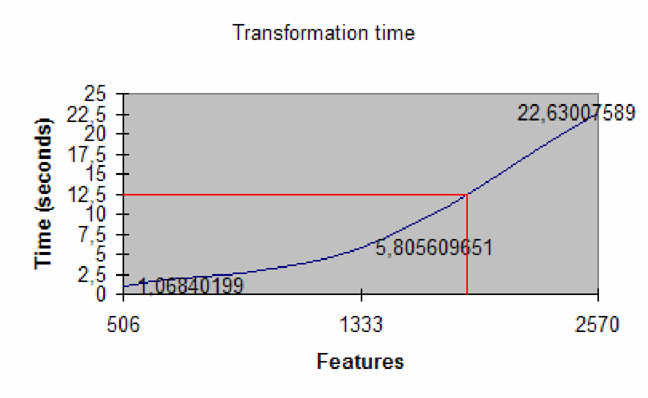


Figure 7.2 Relation between features (buildings + roads) and processing time for Tana data, cases 3-6

The diagram to the left is used to illustrate the approximation on the likely amount of features the implementation will be capable of processing within the tolerable waiting time. 12.5 seconds has been chosen as the intersection point, as this is between the two suggested limits (10-15) in 0 and should allow for some flexibility considering the estimation's accuracy. The intersection along the Features axis occurs at ca 1948 features. As the average ratio between road and building features of test-case 5 and 6 is 1:5.33944981 we can estimate that within a tolerable time, the implementation can transform a dataset of ca 307 roads and ca 1536 buildings with the relatively complex templates.

Processing time vs. Transmission time

In (Lehto and Sarjakoski 2005) the authors have created a system which is using WFS as data source and produce SVG visualizations of generalized data which are then rasterized to PNG. The conversion of GML returned from WFS to SVG is achieved with XSLT using extensions to support the generalization operation. While they do not go into implementation details, and do not report on specific experience with the extensions, they identify two bottlenecks.

1. The request and transmission of data returned from the WFS takes 30-50% of total processing time
2. Rasterization of SVG to PNG 34-54% of total processing time

In addition, as they are working with mobile terminals, they observe that transmission of generated PNG files take almost four times of that spent for generating the image (with PNG files varying between 85-133kb). (Lehto and Sarjakoski 2005) This indicates that for an application using WFS to get geospatial data for visualization, the time induced by using XSLT to transform the data will be of minor importance relative to request/transmission of source data and rasterizing. In a mobile context, the processing time will in anyway be dwarfed by the transmission time of the result to the client given the file sizes as presented by (Lehto and Sarjakoski 2005). These findings are inline with the results from testing done with the Tana data: GML files get relatively large and require quite some time to download, thus it may be the amount of data to transfer that is the main limiting factor of the transformation process, not the XSLT process itself.

Interpretation of findings with regard to a batch context

In a batch context, the performance speed is also of importance, but along with other parameters, it is of equal relevance to consider the capabilities of handling larger sets of data than what is possible in an interactive context. The implemented library has successfully been tested with up to 12266 features in test case 10 of the Tana data. While this did work, special care had to be placed in the design of the templates. It seems that straightforward recursion results in stack-overflow errors at relatively shallow depths (ca 1000) for Xalan based implementations. A “divide-and-conquer” approach to recursion solved this.

Enhancements for the improvement of general performance

This section gives a walk-through of different approaches to optimize the speed of construction and serialization phases. While the issues discussed in 0 and 0 focused on weaknesses and possible improvements in the existing implementation, the points below introduce ideas and features independent of the existing code base.

Caching

An approach with seemingly great potential for performance improvement is to cache the parsed geometry objects. The idea presented here is based on the observation that constructing JTS geometries is an expensive process. In many cases an operation is executed on the same feature (geometry) many times. Such is the example with the Tana-data. To count how many buildings that are situated within 20 meters from the road, each building (point) is checked for presence within a polygon representing a 20 meter distance buffer from the road. The check results in the same polygon node list representation being sent to the “within” extension operation with each and every point. An early prototype of the cache approach has been implemented and tested with the Tana data. See Sections 4.5.5 and 5.1.5, and 6.2.2 for implementation details and test results. Using the cache seems to yield higher performance even for very small sets of features. While the overhead of reparsing the same polygon several times is relatively low compared to the cache approach with regard to total execution time for very small datasets, the time saved by using cached versions of the polygon is close to 50% already by test case 2. (See Chapter 5 for details on the various test cases of the Tana data). Table 7.4 below shows the percentage wise improvement in execution time for each test case of the Tana data when using the cached approach.

Test case	Improvement
1	13,36%
2	45,04%
3	56,20%
4	68,65%
5	71,85%
6	70,27%
7	71,15%
8	69,61%
9	68,71%
10	67,42%

Table 7.4 Improvements in performance time for the cached approach compared with original (non cached) approach.

Whilst performance gains have already been demonstrated with the prototype, there is a need to test this further and check for potentially faster lookups than what is done in the prototype. It should also be done some research into a more exhaustive use of the approach, as it today is only used on multi geometries and line strings. Interesting perspectives would be to check for possibilities on caching whole operations, to save both construction and calculation time. Of course, the caching of serialized geometry object would also be of interest to research closer. It is important to keep all such functionality open for easy customization and configuration by the users. This is discussed further in 7.3.2.

Indexing

An interesting alternative or compliment to caching parsed geometries and results of operations is to keep a full or partial index table with references to all features in the GML being processed. An identified bottleneck in the implemented approach is the search and construction process of geometry objects. By having an index of all features, a quick look-up based on the node list hash code (always provided for all Java objects, (Sun 2004)) would eliminate any iteration on node lists during execution. The table could update its pointers from pointing to “raw” node data to parsed versions of the geometry objects as they get parsed. More advanced implementations in e.g. web contexts, could retain the cached objects in a static index class, so that multiple transformation sessions can benefit of the cached objects instantaneously.

7.3 Flexibility

With regard to research objective 2, this section discusses aspects around the possibilities and limitations of flexibility to the transformation process implied by using GeoXSLT.

7.3.1 Integration/Extension of functionality

The modular design of the GeoXSLT framework was created with openness for integration of additional spatial functionality in mind. That is why the architecture as presented in Section 4.1 is divided into *front-end (level 1)*, *implementation (level 2)*, and *handlers and factories (level 3)* levels. The core functions of converting back and forth between geospatial objects and XSLT is contained in level 3 and accessible to implementations of geo operations in level 2 through a small interface. As long as the geospatial (JTS) objects created by GeoXSLT support the operations desired, implementing spatial functionality can be achieved in just a few steps. In general, it is just a matter of defining the interface in level 1 and a straight forward implementation of logic in level 2. The rest is handled by the GeoXSLT framework. The practical application presented in Section 7.5 includes a thorough example of the simplicity with which GeoXSLT can be extended with additional spatial functionality. Although GeoXSLT was designed and tested with Xalan/XSLT, the use of node lists as parameters opens up for use in pure Java applications as well.

7.3.2 Ideas for improving flexibility

Configuration

The configuration alternatives of the GeoXSLT implementation are in the current version hard coded in a static configuration class and set directly in various classes. This is not inline with the idea of enabling independence from the underlying software implementation. There are several approaches to solve this. While the implementation today only uses extension *functions* which are mapped to the corresponding methods, the XSLT specification also allows for the use of extension *elements*. These are as implied by the name distinct elements contained within an extension namespace. The specification defines them as "…The element extension mechanism allows namespaces to be designated as **extension namespace**s. When a namespace is designated as an extension namespace and an element with a name from that namespace occurs in a template, then the element is treated as an instruction rather than as a literal result element. The namespace determines the semantics of the instruction" (Clark 1999). In other words, the extension elements can be used to pass initial instructions to the processor/framework with configurations of precision models, validation levels, and more. Additionally, the interaction between extension elements and the Java implementation avails more context information than ditto for function extensions. For example, Xalan allows access to both the XSL processor context (org.apache.xalan.extensions.XSLProcessorContext) and the extension element (org.apache.xalan.templates.ElemExtensionCall) from the Java handler of the extension. These objects give the handler access to the complete stylesheet, XML sources (very interesting with regard for indexing) and more (Apache Xalan Community 2005). While this approach certainly is of interest for more research, there is a need to be aware of a limitation. An important performance factor is the support of compiled stylesheets; translets or XSLTC (Apache Xalan Community 2005). The XSLTC XSL processor translates stylesheets into Java class files which then can be used repeatedly for transformation with high performance. The Xalan XSLTC processor does not support the use of extension elements. Another and maybe more viable approach is to use a framework such as Spring for configuration. Spring provides what is called "Injection of Control" which allows Java classes implementing the bean interface to be configured with XML files and hence be configured during runtime (Harrop and Machacek 2005). Implementing this should not give any difficulties, only minor changes to the Configuration class and a general clean-up of code.

Geographic axes

There is already a large set of axes available in XPath (Holzner 2001). These axes are very powerful for the specification of filters and various stylesheets in this thesis apply them in relatively complex queries. Still, in a context of spatial data, there is a need to orientate in a geometric perspective. There should be a possibility of defining a pattern matching elements along a given vector. With this functionality, one could do a search for all elements (or special elements) at e.g. 90 degrees of the current element. Implementing this can be done in many different ways, but this thesis suggests an EXSLT function using the extension bridge provided here. By getting the coordinates of the center-point for each geometry object, calculating the angle between two points is very much possible. JTS (and others) also provide the necessary means for calculations between different SRS/EPSG. It is also of interest to note that JTS provides a "bearing" method which can be used to calculate in which way different features point. The JTS "distance" method is also of relevance, especially with regard for cases where one need to add non-geographic features to a map, such as labels, signs, and text to be placed at certain distances from the targeted feature.

Spatial document function in XSLT

A powerful feature of the XSLT language is the “document(URI)” function (Clark 1999; Holzner 2001). For transformations on non-spatial XML this function can be used to interact with external resources during transformation. This way, transformations can be performed on a minimal source tree and incrementally load external resources as needed. Usually, the document function is used on local files or URLs accessible over HTTP GET. When used in a GML context, there is a challenge with the complex query/filter format used during interaction with WFS servers. Table 2.12 lists a typical WFS query. What is needed is a document(URI) function which can take a payload containing the query/filter that is sent to the URI with the HTTP POST method. By such, the document function could be used to fetch features as needed during runtime achieving the same advantages as for non-spatial XML. As an example one could use data from one WFS source for the initial source tree and select some of the features based on calculations with the GeoXSLT framework. For the selection one could then fetch other details from a second WFS. Everything performed from within the same XSLT template and transformation.

7.4 Consequences for development

This section provides an answer to research objective 3, and argues how the integration of spatial functionality made possible by the implementation created will have a positive effect on practical development with spatial data.

7.4.1 Less surrounding complexity

The work with creating and performing all the test cases has shown the simplicity of processing spatial data with the implemented GeoXSLT framework. While other approaches require the user to setup and configure various supporting toolkits just to start working on the data, experience shows that this approach allows loading and processing spatial data without any changes to the existing code underlying the transformation. This means that one can use any XSLT processor supported by the framework and start processing GML with spatial functionality without changing a thing. This allows the developer to focus on using the spatial functionality, instead of implementing or integrating it. All that is necessary is to provide the jar file containing the spatial extension frame work on the classpath. This allows for easy integration into existing systems, e.g. web applications with JSTL (Sun 2006) based JSP pages, where spatial transformations now can be performed with minimal code without having any impact on other parts of the application.

7.4.2 Code readability

When compared with general programming languages, XSLT is a simpler language with focus on the flow and transformation of data. Previous research and Chapter 3 defined the need for syntax on the spatial functionality that follows the same patterns as those native to XSLT. By such, the simplicity already inherent in XSLT is conserved, and usage of the extensions intuitive for developers. The implementation of the framework supporting the spatial operations succeeded in achieving this, as demonstrated in the stylesheets used for testing both constructed and real data in Chapter 6. The result is that standard spatial functionality as defined in Chapter 3 integrates fully with the standard XSLT used in the templates without obfuscating the code readability. Additionally, the simple syntax of the operations defined is intuitive and easy to use. The extensions open up for different approaches to the creation of XSLT templates, and it is up to the developer whether to pack much functionality into a singular XPath statement, or to focus on readability by spreading the logic across several operations or templates. The spatial functions work just as well with any approach.

An issue that one should be aware of is the challenges associated with “re-processing” the output of templates already ran inside the current XSLT process. The output from templates is represented as result-tree fragments, which only supports string operations (Clark 1999). While this is discussed with more detail in Section 7.5, it is not specifically related to geospatial data, and as showed, solving it by converting the fragments to node sets is not a problem.

7.4.3 Reuse

Application level

The Model-View-Controller (MVC) pattern defines an approach to programming where

application and business logic represented by the “model” is kept separate from the presentation/user interface, represented by the “view”. A “controller” mediates the interaction between the two components (Buschmann, Meunier et al. 1996).
With regard to development patterns focusing on the separation between presentation and application logic, XSLT can be a viable approach to generate the views used for presentation. This is assuming that the underlying data model can be represented as XML. While MVC have several benefits, such as support for multiple views of the same model and “pluggable” views and controllers (Buschmann, Meunier et al. 1996), an acknowledged liability is the intimate connection between view and controller (Buschmann, Meunier et al. 1996). When using XSLT for processing geospatial data, the controller has to accommodate the handling of spatial operations, hence tightening the connection between the view (XSLT template) and the controller. This makes generalization of the application code in the controller more difficult, obfuscates the controller implementation, and complicates the use of different views of the same model/using the same view on multiple controllers.
By introducing spatial functionality to the XSLT process, we have made it easier to decouple the XSLT view from the controller and model. The increased decoupling can make it easier to reuse both the XSL template and support better code generalization in the controller. The increased level of distinction between the view and controller, may also make it easier for developers to focus on the task at hand: Developers of views can concentrate on creating the view without interfering with the controller or other application level code.

Template level

Code reuse is well supported at the template level of XSLT. By developing general templates for handling various features, cartographic and non-cartographic alike, specific transformations can be a matter of just applying the templates desired.

7.4.4 Possibilities for extension

Integrating custom or missing functionality into the framework has been an important goal in the design of the implementation done. Chapter 4 discusses the aspects around design and implementation with regard for integration; while Section 7.5 presents an example of how custom spatial operations can be integrated using the construction and serialization facilities provided by the framework. The working prototype and demonstration in Section 7.5 give weight to the claim that the modular architecture supports extensions with new functionality quite well.

7.4.5 Possibilities for implementation on other platforms

Libraries similar to Java Topology Suite with full or partial support for SFS are available for most popular platforms of development. Only a few are mentioned here. While these support the calculations needed, the amount of work needed to implement the bridged as done for JTS is unknown.

- Perl and C/C++: GDAL/OGR. C API and Perl modules (http://www.remotesensing.org/gdal/ogr/ + http://search.cpan.org/~sderle/Geo-GDAL-0.11/)
- .NET: “nettopologysuite”; port of JTS. May also be able to C API. (http://code.google.com/p/nettopologysuite/)

7.5 *Practical application*

This section walks through an example of use by applying the GeoXSLT framework to a practical task.

In (Harrie L. 2003) Harrie and Johansson describe a method for real-time generalization and visualization of GML data. While XSLT is used for generation of the SVG file (visualization), a separate Java program is used to perform the integration and generalization of data. During this process there is a defined need to have an interaction between the different geometry representations involved. (Harrie L. 2003) provides several examples of scenarios requiring this feature, terming it “interaction between objects”:

- Solve spatial conflicts (e.g. making sure symbols representing one feature do not cover other features)
- Integrate service data such as icons/arrows etc with cartographic data
- Aggregation of objects

Harrie and Johansson chose not to use XSLT for the generalization of data because “...XSLT transformations only treat one object at a time, it is not possible to implement methods which involve interactions between objects”. (Harrie L. 2003). This thesis is not going to claim that placing the generalization and integration of data in a separate Java program is wrong by any means. Still, it is important to provide argumentation for why it may actually be possible to perform the generalization and integration in an XSLT context and still be achieving object interaction. By using XSLT to perform the generalization process, the extra step of running the data through the extra Java program before using XSLT for SVG visualization can be saved. Containing the process within XSLT does also facilitate the possibility for developing a more general (Java) system underneath, with better possibilities for code/class generalization and re-use, deploying custom XSLT templates for specific cases.

Example

As the arguments for using XSLT are of significance, there is a need to understand how it can be implemented. By demonstrating how interactions between objects are possible in XSLT and a description of relevant functionality available in JTS (which with relative ease can be integrated into the level 2 classes), this thesis attempts to inspire further research on XSLT based generalization. The demonstration is based on test case 4 of the Tana data. First, the roads are simplified using the DouglasPeucker algorithm (Sedgewick 2002; Novatchev and Tyszko 2006) to showcase simple generalization. Then, *based on interaction with the simplified geometry object representing the roads*, a buffer indicating the 20 meter area used in the test cases with the Tana data (5.1.4 and 6.2) is created. This is to showcase that XSLT can interact with results from previously called templates within the XSLT process, and hence makes object interaction possible. Everything is visualized as SVG using the transformation process itself, and buildings within the 20 meter buffer are given a distinct coloring. As a frame of reference, the original visualization of data set 4 from Tana is shown in Figure 7.3.

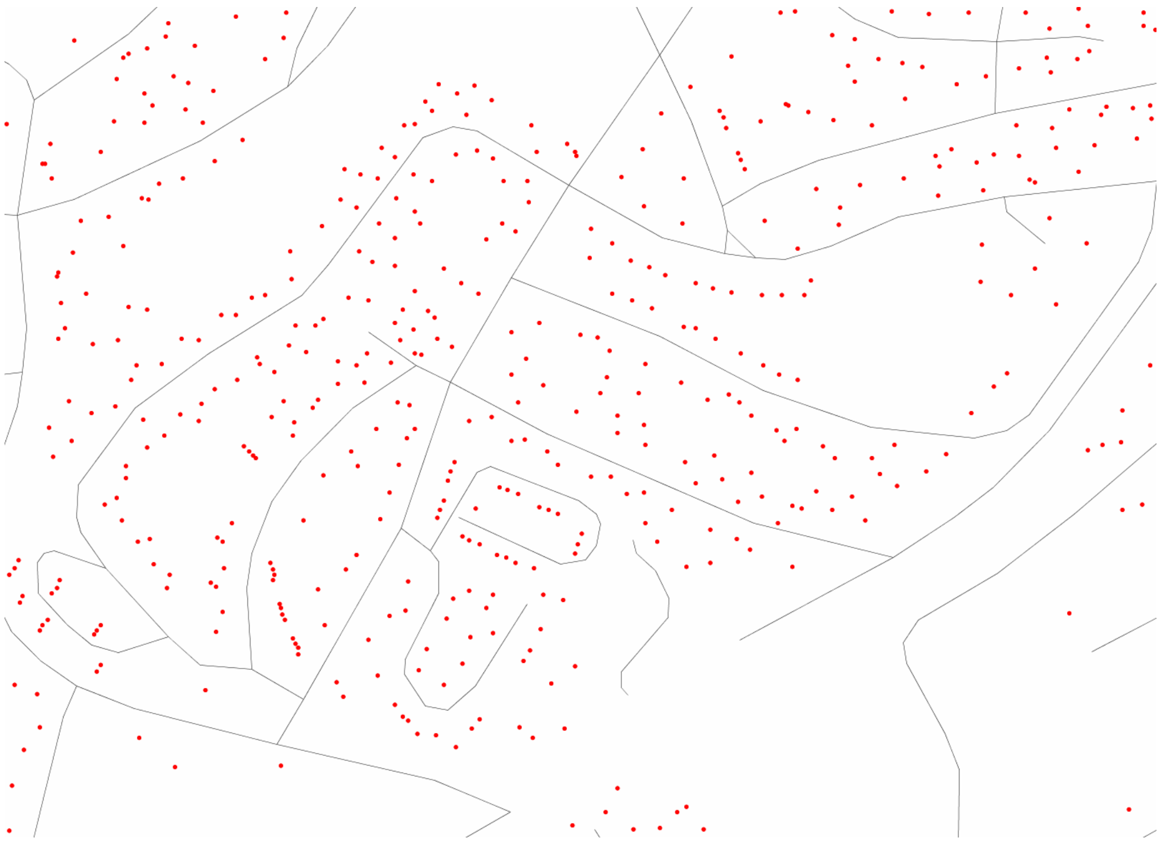


Figure 7.3 WMS (Geoserver) visualization of Tana data set 4

In Figure 7.3 no simplifications or alterations have been made. The GML data representing the features is passed on to a relatively simple XSLT stylesheet utilizing the JTS bridge created as a part of the thesis. Table 7.5 below lists a very small subset of the GML data representing the feature set. The GML has been extracted directly from a Geoserver WFS instance and is shown to provide an understanding of the transformation flow.

```
<?xml version="1.0" encoding="UTF-8"?>
<wfs:FeatureCollection
xmlns:wfs="http://www.opengis.net/wfs" xmlns:topp="http://www.openplans.org/topp"
xmlns:gml="http://www.opengis.net/gml" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.openplans.org/topp
http://localhost:8080/geoserver/wfs/DescribeFeatureType?typeName=topp:bulroad,topp:Road
http://www.opengis.net/wfs http://localhost:8080/geoserver/schemas/wfs/1.0.0/WFS-basic.xsd">

  <gml:boundedBy>
    <gml:Box srsName="http://www.opengis.net/gml/srs/epsg.xml#32633">
      <gml:coordinates xmlns:gml="http://www.opengis.net/gml" decimal="." cs="," ts=" ">356145,7766050
357617,7767139
      </gml:coordinates>
     </gml:Box>
    </gml:boundedBy>

    <gml:featureMember>
        <topp:bulroad fid="bulroad.674">
          <topp:the_geom>
            <gml:Point srsName="http://www.opengis.net/gml/srs/epsg.xml#32633">
                <gml:coordinates xmlns:gml="http://www.opengis.net/gml" decimal="." cs="," ts=" ">
                    357403,7766898
                 </gml:coordinates>
              </gml:Point>
          </topp:the_geom>
            <topp:type>Residential</topp:type>
        <topp:status>2</topp:status>
        <topp:number>192563771</topp:number>
        <topp:started>10101</topp:started>
        <topp:updated>20001120</topp:updated>
   </topp:bulroad>
  </gml:featureMember>
  <gml:featureMember>
   <topp:Road fid="Road.854">
    <topp:the_geom>
     <gml:MultiLineString srsName="http://www.opengis.net/gml/srs/epsg.xml#32633">
      <gml:lineStringMember>
       <gml:LineString>
        <gml:coordinates xmlns:gml="http://www.opengis.net/gml" decimal="." cs="," ts=" ">
          356709,7766479 356756,7766427
        </gml:coordinates>
       </gml:LineString>
      </gml:lineStringMember>
     </gml:MultiLineString>
    </topp:the_geom>
    <topp:type>Municipal</topp:type>
    <topp:roadNumber>5076</topp:roadNumber>
    <topp:roadClass>V</topp:roadClass>
    <topp:date>19980714</topp:date>
   </topp:Road>
  </gml:featureMember>
```

Table 7.5 A minimal subset representing content and structure of data used in the generalization process

As mentioned earlier, the Tana data-set is made up of many small lengths of line strings representing roads. During a generalization process, one can choose between the approaches of simplifying the line strings one by one, maintaining full control and keeping non cartographic feature data associated (see Table 7.5 for examples of non-cartographic feature data; “roadNumber”, “type” etc), or to union all the lengths into one multi line string which then is simplified. In this case the latter approach has been chosen, as only the generalization aspect is of direct relevance to the demonstration. The union of all the road segments is done using the same template as listed in Table 5.4 and stored in a variable.

```
<xsl:variable name="roadUnion">
    <xsl:call-template name="unionRoads">
      <xsl:with-param name="roads"
                     select="wfs:FeatureCollection/gml:featureMember[topp:Road]/topp:Road/topp:the_geom/*" />
    </xsl:call-template>
</xsl:variable>
```

Table 7.6 Calling the unionRoads template and storing the resulting multi line GML element variable “roadUnion”.

The “roadUnion” variable does at this point contain a result tree fragment, which according to (Clark 1999) cannot support any other operations as those available for string values. This is probably the main reason why Harrie (Harrie L. 2003) concludes that interactions between [processed] objects are not possible within XSLT. While it is correct according to the plain vanilla XSLT specification, in practice Xalan provides handling to work around the problem. Xalan provides a built-in extension function called “nodeset”.

When passed a result tree fragment, it returns an equivalent set of nodes. This approach is applied in the demonstration when simplification is performed on the “roadUnion” variable.

```
<xsl:variable name="roadSimplified" select="exp:simplify(xalan:nodeset($roadUnion)//gml:MultiLineString,20)" />
```

Table 7.7 Performing Douglas Peucker simplification on the union of roads.

As is listed in Table 7.7, the result tree fragment of the “roadUnion” variable is converted to a node set and passed on to an extension function mapped to the Douglas Peucker simplification algorithm. The number, “20” denotes the tolerance threshold for vertex distance used in the simplification. The “exp” namespace prefix denotes the experimental group of functions containing the simplify function.

```
public Node simplify(NodeList n1,double distanceTolerance) {
      Geometry g1 = utilities.gmlSearch(n1);
      Geometry result = DouglasPeuckerSimplifier.simplify(g1,distanceTolerance);
      return gf.encodeNode(result);
   }
```

Table 7.8 Implementation of simplification extension function in a level 2 class.

Table 7.8 lists the experimental implementation of the simplification algorithm. Note how the thesis’ implemented framework reduces bidirectional conversion between nodes and JTS geometry objects to a simple method call. While this is an experimental implementation of simplification functionality without any performance evaluations, it does showcase the ease with which additional functionality can be integrated almost in a plug-in fashion without any detailed knowledge of GML processing or JTS. No change to the framework or other level 2 classes was necessary for this “plug-in” of simplification functionality. The “DouglasPeuckerSimplifier” class is a part of the “com.vividsolutions.jts.simplify” package available with JTS.

After returning from the simplify call, the now simplified union of roads is stored in the “roadSimplified” XSLT variable. To calculate the area surrounding the road with 20 meters, a buffer is created. The resulting multi polygon is stored in a “roadBuffer” variable.

```
<xsl:variable name="roadBuffer" select="gis:buffer($roadSimplified//gml:MultiLineString,20)" />
```

Table 7.9 Calculating the 20 meter “buffer-area” to surround the simplified union of roads.

GML representing the simplified version of the road and the buffer surrounding it are now in place. All that is left is to transform the GML into SVG with distinct coloring of the different components.

```
<!—
Draw the outer border of the multi polygon representing the 20 m area (buffer) outside the simplified union of roads
 -->
<xsl:apply-templates select="$roadBuffer//gml:outerBoundaryIs" />

<!—Draw the inner border of the multi polygon (inside part of the buffer; fill++) -->
<xsl:apply-templates select="$roadBuffer//gml:innerBoundaryIs" />

<!—Draw the simplified union of roads on top of the buffer -->
<xsl:apply-templates select="$roadSimplified//gml:LineString" />

<!—Draw all the points representing houses WITHIN the 20 meter buffer -->
<xsl:apply-templates select="wfs:FeatureCollection/gml:featureMember[
        (descendant::topp:bulroad) and
        (gis:within(
                    descendant::topp:bulroad/topp:the_geom/gml:Point,
                    $roadBuffer/gml:*[position() = 1]
                   )
         )
                                                                 ]" mode="inside"/>

<!—Draw all the points representing houses OUTSIDE the 20 meter buffer -->
<xsl:apply-templates select="wfs:FeatureCollection/gml:featureMember[
        (descendant::topp:bulroad) and
        (not(gis:within(
                    descendant::topp:bulroad/topp:the_geom/gml:Point,
                    $roadBuffer/gml:*[position() = 1]
                   ))
         )
                                                                 ]" mode="outside"/>
```

Table 7.10 Applying transformation to SVG for the generated components

Table 7.10 lists the XSL used to render the GML in SVG. Note how houses are rendered in two sets, depending on a check for their existence within the buffer zone defined by the “roadBuffer” variable. This demonstrates that it is very well possible to place non-cartographic data with care for not covering important cartographic details. E.g. in the case of placing arrows or labels in a map, the “within” check (or any other relevant SFS function) used in the table could be used to validate the position. For recalculation of label positioning etc, one can use “distance” and the “compass”/“bearing” functionality as defined in 7.3.2.

While generalization/simplification has not been tested in detail with regard to performance and multiple algorithms within this thesis, the walk-through presented above gives a clear indication that generalization can very well be achieved within in an XSLT perspective, and that the “interaction” argument of (Harrie L. 2003) is not necessarily a problem. The rest of the stylesheet, performing the transformation from GML to SVG, is available in the appendix.

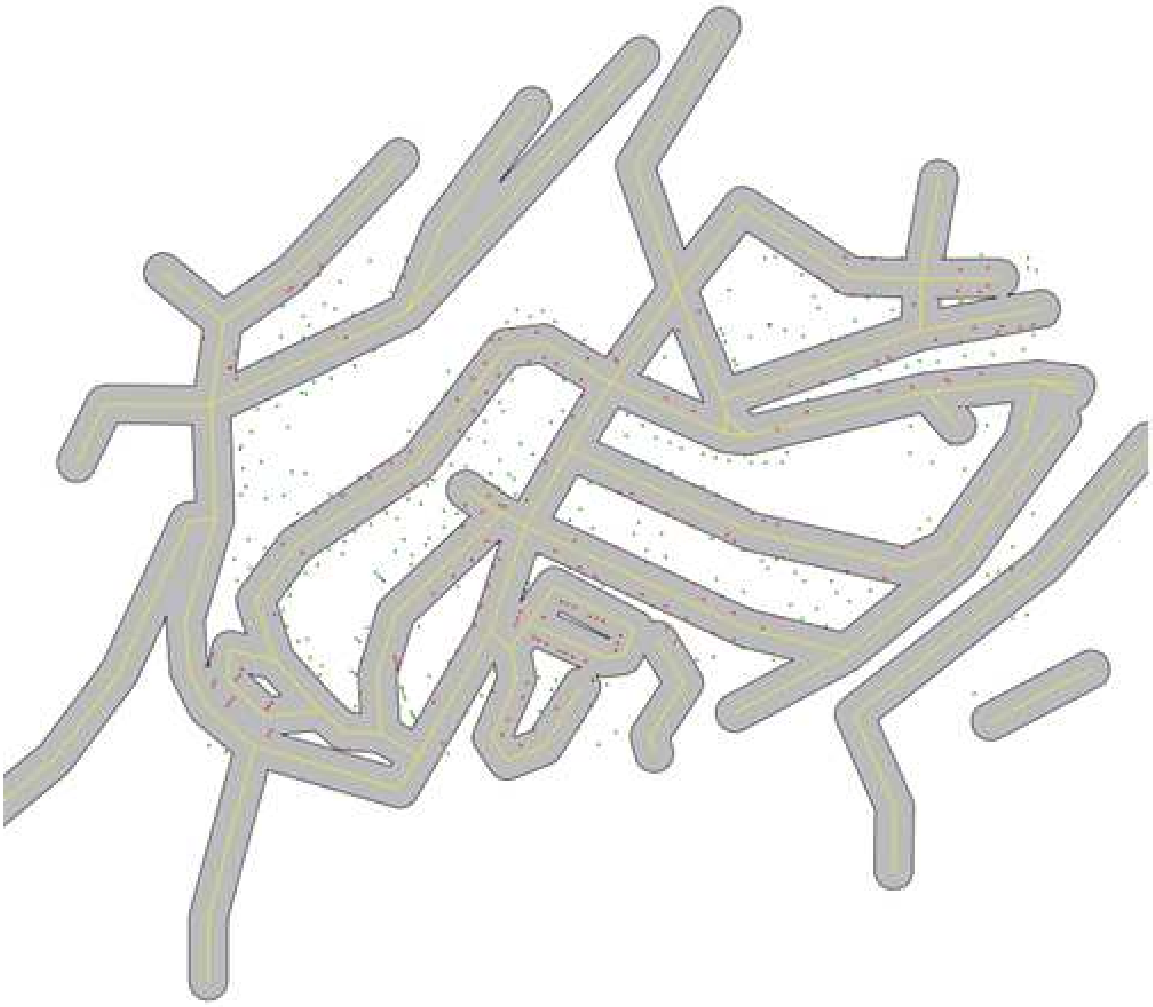


Figure 7.4 SVG visualization of Tana data set for case 4, simplified with a distance tolerance of 20 m

Figure 7.4 shows the final result of the walk-through. The generated rendering is in no way optimized with regard to the available possibilities of SVG. The simplified road is rendered yellow, the buffer area grey with dark borders. Houses outside the buffer are colored green, while those within the buffer are red.

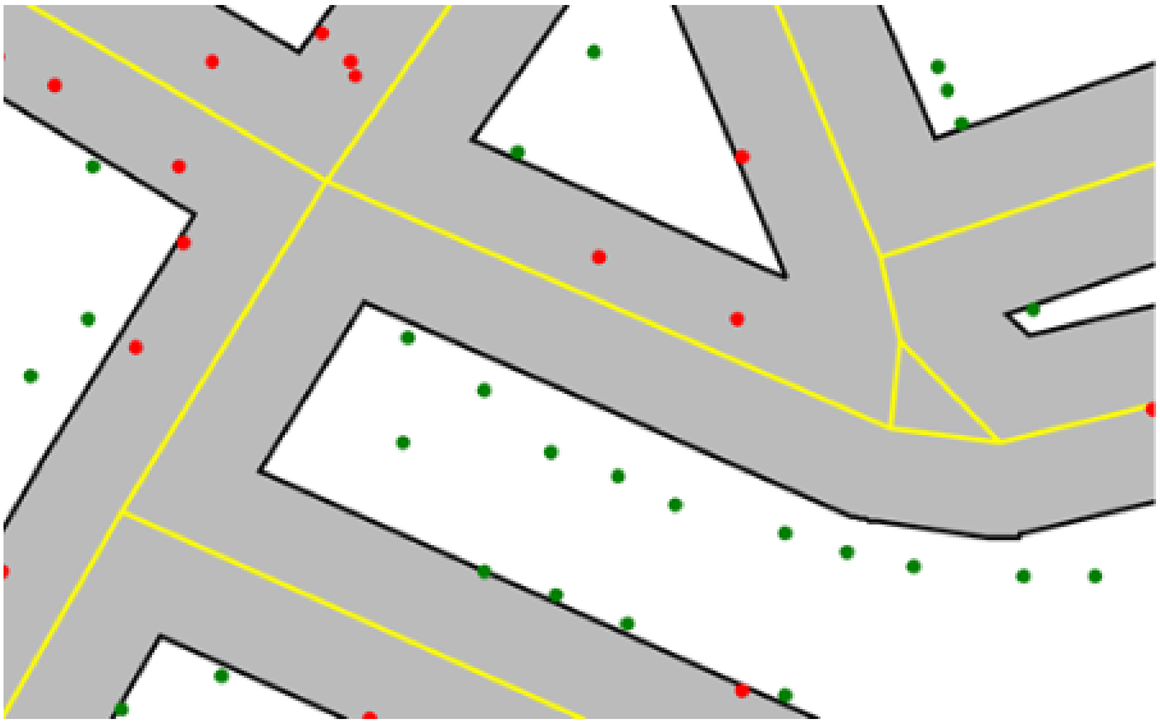


Figure 7.5 Close-up, subset of figure 7.4

8 Conclusion

This chapter summarizes the research in view of the research objectives before a short presentation of the major contributions and an outline of further work is provided.

8.1 Claimed results

Processing geospatial data with XSLT

The work with this thesis shows that it is technically possible to process and transform geospatial data encoded as GML with XSL templates and extensions. The experience with the developed GeoXSLT framework indicates that spatial operations possible with libraries such as the Java Topology Suite can be integrated into the XSLT process once the GML representations are converted to geo objects. The prototype does just that.

Performance

Tests done with real data shows that in an interactive context, spatial XSLT transformations of data can be performed with success within the different limits of time as set forth by the usability community. While the transformations achievable within *instant time* are limited, chances are that these limitations would also apply to other approaches. For scenarios occurring within a timeframe of *tolerable time*, relatively complex transformations can be applied on data sets so large that network transmission time and other factors external to the transformation become issues. In a batch perspective, tests involving relatively complex transformations on data sets consisting of more than 11 000 features have shown that with considerate use of recursion, the approach proposed in this thesis can handle large amounts of data.

Geometry	Construction	Serialization
Line String	10-12%	2-25%
Polygon	12-27%	6-61%

Table 8.1 Overhead for analysis operations

Geometry	Construction
Line String	20-25%
Polygon	8-64%
Point	40-82%

Table 8.2 Overhead for predicate operations

Geometry	Construction	Serialization
Point	15-42%	58-85%

Table 8.3 Construction vs. Serialization for points

With regard for the time overhead necessary for conversion back and forth between the transformation process and the spatial extensions, the experience from design and development of the prototype shows that the integration of spatial functionality in an XSLT process can be divided into three phases The phases are: construction of spatial objects from node sets, spatial calculations, and serialization of the results back to a node structure compatible with the XSLT process. The overhead implied by the integration of spatial capabilities with XSLT consists of the construction and serialization phases.

Tables 8.1 and 8.2 present indications on the percentage wise overhead imposed by the GeoXSLT framework for analysis and predicate operations. Note that the high overhead for operations on points is caused by the very fast point calculations; the total time to complete operations on points is significantly smaller than for any other geometry. See Section 6.1.4 for a complete summary on performance measured.

Table 8.3 presents findings from running a “dummy” test on points without any calculations; consisting of only the construction and serialization phases. It indicates that serialization is more expensive than the construction phase.

Tests and findings in this thesis indicate that the cost of making geospatial operations available to XSLT is significant. Nonetheless, other factors such as possibilities for shorter workflow and reduced network traffic should be counted in. This thesis has also shown that significant performance gains are achievable by the implementation of different optimization mechanisms. The simple caching described in Sections 4.5.5, 5.1.5, and 6.2.2 improved overall performance by 60%.

Flexibility

The approach described and tested in this thesis provides sufficient flexibility for many uses and scenarios. This is because the combination of flexibility inherent in XSLT with the complete integration of spatial extensions provided by the prototype results in a working platform for transformation and processing spatial data. Additionally, the modular architecture of the framework created allows for easy integration of new functionality and operations.

The combination of a well integrated set of extension operations for the XSLT language with a modular and extensible supporting framework results in a flexible platform that can be customized to support transformation of geospatial data in a multitude of scenarios. Examples of this are given in the tests with real data (Sections 5.1.4, 5.1.5, 6.2.1, 6.2.2) and the practical application described in Section 7.5. The examples and argumentation presented have shown that spatial operations can be combined, chained, and used in XPath expressions. Likewise, simple extensions of the standard functionality provided have been showcased to illustrate the modular architecture. An approach to the "object-interaction" necessary for generalization/simplification operations (Harrie L. 2003) has also been demonstrated to work.

Development facilitation

The experience with practical applications and tests of the prototype shows that it is easy to start developing templates when no special measures or configuration need to be done for utilization of the spatial functionality. The developer's focus can as such be concentrated on expressing the logic with traditional XSLT and the simple API defined. The impression from using the prototype so far is that the threshold for development of geospatial applications is lowered allowing for more time and resources to be spent on creativity.

8.2 Major contributions

The contributions of this thesis are

- A working prototype for easy integration of spatial functionality in transformation of geodata encoded as GML.
- An exslt.org style API which is closely mapped to the Simple Feature Specification.
- Experience and examples from practical work with XSLT transformations of GML using the spatial extensions.

8.3 Future work

The future work with the GeoXSLT framework should look into the suggested improvements of Chapter 7. In addition some other possibilities are listed below.

Automatic mapping of function calls

Many Level 2 implementations of operations defined in the Simple Feature Specification follow a very similar pattern. This opens up for the idea of a facility providing automatic mapping of JTS functionality with XSLT based on the Java introspection. With such a facility on would, ideally, just have to call the spatial extension function from XSLT and have the operation automatically executed.

Performance optimization

The overhead of using GeoXSLT is significant. More research should be done to improve the performance.

Spatial libraries

The current implementation of GeoXSLT depends on Java Topology Suite (JTS) for spatial calculations. It would be of interest to se how other libraries available could be used. An idea is to create support for other libraries using a plug-in approach. There are several databases that support the Simple Features Specification.

Other platforms

GeoXSLT needs to be available on other platforms than Java. Section 7.4.5 lists possible alternatives to Java Topology Suite for use in other platforms.

Release of GeoXSLT source code

GeoXSLT is released under a GPL license to be of use for everyone and open up for contributions to further development.
The source code is available at http://www.svisj.no/fredrik/geoxslt

9 Appendix

9.1 Sample of constructed test data

```
<?xml version="1.0"?>
<testdata xmlns:gml="http://www.opengis.net/gml">
<!--

HORIZONTAL LINE:
PARAMS:
$VAR1 = {
          'vertices' => 3,
          'ybuf' => 6,
          'step' => 2,
          'xmulti' => 1,
          'ymulti' => 1,
          'x' => '0.11',
          'xbuf' => 10,
          'decimals' => 2
        };
WKT:
LINESTRING(10.11 6.03,12.11 6.50,14.11 6.86,)

VERTICAL LINE:
PARAMS:
$VAR1 = {
          'x' => 0,
          'y' => '4.11',
          'vertices' => 3,
          'step' => 2,
          'ybuf' => 0,
          'xmulti' => 1,
          'ymulti' => 1,
          'xbuf' => 12,
          'decimals' => 2
        };
WKT:
LINESTRING(12.86 4.11,13.00 6.11,12.90 8.11,)

-->

<gml:featureMember>
		<dummy_line fid="horizontal_1">
				<TheGeometry>
						<gml:LineString>
								<gml:coordinates decimal="." cs=" " ts=",">10.11 6.03,12.11
6.50,14.11 6.86,</gml:coordinates>
						</gml:LineString>
				</TheGeometry>
		</dummy_line>
</gml:featureMember>


<gml:featureMember>
		<dummy_line fid="vertical_1">
				<TheGeometry>
						<gml:LineString>
								<gml:coordinates decimal="." cs=" " ts=",">12.86 4.11,13.00
6.11,12.90 8.11,</gml:coordinates>
						</gml:LineString>
				</TheGeometry>
```

```
          </dummy_line>
</gml:featureMember>

</testdata>
```

9.2 Sample of real data

This is a minimal sample of the Tana data set to illustrate the structure and composition.

```
<?xml version="1.0" encoding="UTF-8"?>
<wfs:FeatureCollection xmlns:wfs="http://www.opengis.net/wfs"
xmlns:topp="http://www.openplans.org/topp" xmlns:gml="http://www.opengis.net/gml"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.openplans.org/topp
http://localhost:8080/geoserver/wfs/DescribeFeatureType?typeName=topp:bulroad,topp:Road
http://www.opengis.net/wfs http://localhost:8080/geoserver/schemas/wfs/1.0.0/WFS-basic.xsd">
          <gml:boundedBy>
                    <gml:Box srsName="http://www.opengis.net/gml/srs/epsg.xml#32633">
                              <gml:coordinates xmlns:gml="http://www.opengis.net/gml" decimal="." cs=","
ts=" ">357015,7766577 357502,7766771</gml:coordinates>
                    </gml:Box>
          </gml:boundedBy>
          <gml:featureMember>
                    <topp:bulroad fid="bulroad.2545">
                              <topp:the_geom>
                                        <gml:Point srsNa-
me="http://www.opengis.net/gml/srs/epsg.xml#32633">
                                                  <gml:coordinates xmlns:gml="http://www.opengis.net/gml"
decimal="." cs="," ts=" ">357080,7766653</gml:coordinates>
                                        </gml:Point>
                              </topp:the_geom>
                              <topp:type>Outhouse</topp:type>
                              <topp:status>2</topp:status>
                              <topp:number>192574250</topp:number>
                              <topp:started>10101</topp:started>
                              <topp:updated>19940210</topp:updated>
                    </topp:bulroad>
          </gml:featureMember>
          <gml:featureMember>
                    <topp:Road fid="Road.907">
                              <topp:the_geom>
                                        <gml:MultiLineString srsNa-
me="http://www.opengis.net/gml/srs/epsg.xml#32633">
                                                  <gml:lineStringMember>
                                                            <gml:LineString>
                                                                      <gml:coordinates
xmlns:gml="http://www.opengis.net/gml" decimal="." cs="," ts=" ">357015,7766698 357127,7766654
357205,7766613 357286,7766585 357364,7766577 357389,7766583 357406,7766595 357488,7766710
357498,7766735 357502,7766771</gml:coordinates>
                                                            </gml:LineString>
                                                  </gml:lineStringMember>
                                        </gml:MultiLineString>
                              </topp:the_geom>
                              <topp:type>Municipal</topp:type>
                              <topp:roadNumber>5088</topp:roadNumber>
                              <topp:roadClass>V</topp:roadClass>
                              <topp:date>19980714</topp:date>
                    </topp:Road>
          </gml:featureMember>
</wfs:FeatureCollection>
```

9.3 SVG/Generalization stylesheet

For the sake of readability, the complete file is available at

http://www.svisj.no/fredrik/geoxslt

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:wfs="http://www.opengis.net/wfs" xmlns:gml="http://www.opengis.net/gml"
xmlns:topp="http://www.openplans.org/topp"  xmlns:svg="http://www.w3.org/2000/svg"
xmlns:exslt="http://exslt.org/common" xmlns:xalan="http://xml.apache.org/xalan"
xmlns:exp="xalan://frontend.ExperimentalOperations" xmlns:gis="xalan://frontend.SFSOperations" ex-
clude-result-prefixes="wfs gml topp svg exslt xalan exp gis">
   <xsl:output method="xml" version="1.0" encoding="UTF-8" indent="yes" doctype-public="-
//W3C//DTD SVG 1.1//EN" doctype-system="http://www.w3.org/Graphics/SVG/1.1/DTD/svg11-flat-
20030114.dtd" />
   <xsl:template match="/">
      <svg xmlns="http://www.w3.org/2000/svg" width="100%" height="100%">
         <g transform="rotate(180, 500,500) scale(-1 1) translate(-1000 0)">

            <xsl:element name="svg" namespace="http://www.w3.org/2000/svg">

               <xsl:attribute name="width">1000</xsl:attribute>
               <xsl:attribute name="height">1000</xsl:attribute>


               <xsl:variable name="bbcoords">
                  <xsl:value-of select="wfs:FeatureCollection/gml:boundedBy/gml:Box/gml:coordinates"/>
               </xsl:variable>


               <xsl:variable name="bbMin" select="substring-before($bbcoords,' ')" />
               <xsl:variable name="bbMax" select="substring-after($bbcoords,' ')" />

               <xsl:variable name="vb_x" select="substring-before($bbMin,',')" />
               <xsl:variable name="vb_y" select="substring-after($bbMin,',')" />

               <xsl:variable name="vb_width" select="substring-before($bbMax,',') - $vb_x" />
               <xsl:variable name="vb_height" select="substring-after($bbMax,',') - $vb_y" />
               <xsl:attribute name="viewBox"><xsl:value-of select="concat($vb_x,' ',$vb_y,' ',$vb_width,'
',$vb_height)"/></xsl:attribute>

               <xsl:element name="g" namespace="http://www.w3.org/2000/svg">



                  <xsl:variable name="roadUnion">
                     <xsl:call-template name="unionRoads">
                        <xsl:with-param name="roads" se-
lect="wfs:FeatureCollection/gml:featureMember[topp:Road]/topp:Road/topp:the_geom/*" />
                     </xsl:call-template>
                  </xsl:variable>
                  <xsl:message>Road union created</xsl:message>

                  <xsl:variable name="roadSimplified" se-
lect="exp:simplify(xalan:nodeset($roadUnion)//gml:MultiLineString,10)" />

                  <xsl:variable name="roadBuffer" se-
lect="gis:buffer($roadSimplified//gml:MultiLineString,20)" />


                  <xsl:apply-templates select="$roadBuffer//gml:outerBoundaryIs" />
                  <xsl:apply-templates select="$roadBuffer//gml:innerBoundaryIs" />
                  <xsl:apply-templates select="$roadSimplified//gml:LineString" />
```

```
                        <xsl:apply-templates se-
lect="wfs:FeatureCollection/gml:featureMember[(descendant::topp:bulroad/topp:the_geom/gml:Point)
and (gis:within(descendant::topp:bulroad/topp:the_geom/gml:Point,$roadBuffer/gml:*[position() = 1]))]"
mode="inside"/>
                        <xsl:apply-templates se-
lect="wfs:FeatureCollection/gml:featureMember[(descendant::topp:bulroad) and
(not(gis:within(descendant::topp:bulroad/topp:the_geom/gml:Point,$roadBuffer/gml:*[position() =
1])))]" mode="outside"/>

                    </xsl:element>
                </xsl:element>
            </g>
        </svg>

    </xsl:template>


    <xsl:template match="topp:bulroad" mode="inside">
        <xsl:variable name="gmlcoords" select="topp:the_geom//gml:coordinates"/>
        <circle cx="{substring-before($gmlcoords,',')}" cy="{substring-after($gmlcoords,',')}" r="1"
style="stroke:red; stroke-width:1; fill:red" xmlns="http://www.w3.org/2000/svg"/>
    </xsl:template>

    <xsl:template match="topp:bulroad" mode="outside">
        <xsl:variable name="gmlcoords" select="topp:the_geom//gml:coordinates"/>
        <circle cx="{substring-before($gmlcoords,',')}" cy="{substring-after($gmlcoords,',')}" r="1"
style="stroke:green; stroke-width:1; fill:green" xmlns="http://www.w3.org/2000/svg"/>

    </xsl:template>

    <xsl:template match="topp:Road">
        <xsl:variable name="gmlcoords" select="topp:the_geom//gml:coordinates"/>
        <polyline points="{$gmlcoords}" style="stroke:black; stroke-width:4;fill:none;"
xmlns="http://www.w3.org/2000/svg" />
        <polyline points="{$gmlcoords}" style="stroke:yellow; stroke-width:1;fill:none;"
xmlns="http://www.w3.org/2000/svg" />
    </xsl:template>

    <xsl:template match="gml:LineString">
        <xsl:variable name="gmlcoords" select=".//gml:coordinates"/>
        <!-- <polyline points="{$gmlcoords}" style="stroke:black; stroke-width:4;fill:none;"
xmlns="http://www.w3.org/2000/svg" />                                    -->
        <polyline points="{$gmlcoords}" style="stroke:yellow; stroke-width:1;fill:none;"
xmlns="http://www.w3.org/2000/svg" />
    </xsl:template>

    <xsl:template match="gml:outerBoundaryIs">
        <xsl:variable name="gmlcoords" select=".//gml:coordinates"/>
        <polyline points="{$gmlcoords}" style="stroke:black; stroke-width:1;fill:#BBBBBB;"
xmlns="http://www.w3.org/2000/svg" />
    </xsl:template>

    <xsl:template match="gml:innerBoundaryIs">
        <xsl:variable name="gmlcoords" select=".//gml:coordinates"/>
        <polyline points="{$gmlcoords}" style="stroke:black; stroke-width:1;fill:white;"
xmlns="http://www.w3.org/2000/svg" />
    </xsl:template>


    <xsl:template name="unionRoads">
        <xsl:param name="roads"/>
        <xsl:variable name="roadCount" select="count($roads)" />

        <xsl:choose>
            <xsl:when test="$roadCount = 1"><xsl:copy-of select="$roads" /></xsl:when>
            <xsl:otherwise>
                <xsl:variable name="fiftypercent" select="floor($roadCount div 2)" />
                <xsl:variable name="left">
                    <xsl:call-template name="unionRoads">
                        <xsl:with-param name="roads" select="$roads[position() &lt;= $fiftypercent]" />
                    </xsl:call-template>
                </xsl:variable>
                <xsl:variable name="right">
                    <xsl:call-template name="unionRoads">
                        <xsl:with-param name="roads" select="$roads[position() &gt; $fiftypercent]" />
                    </xsl:call-template>
                </xsl:variable>
                <xsl:copy-of select="gis:union(xalan:nodeset($left)//*[(local-name() = 'MultiLineString') or
(local-name() = 'LineString')],xalan:nodeset($right)//*[(local-name() = 'MultiLineString') or (local-name()
= 'LineString')])" />

            </xsl:otherwise>
        </xsl:choose>
    </xsl:template>

</xsl:stylesheet>
```

10 References

Apache Xalan Community. (2005). "Xalan Extensions Library." from http://xml.apache.org/xalan-j/extensionslib.html.

Apache Xalan Community. (2005). "Xalan Project website." from http://xalan.apache.org/.

Apache Xalan Community. (2005). "XSLTC." from http://xml.apache.org/xalan-j/extensions_xsltc.html.

Beaujardiere, J. d. l. (2006). OpenGIS® Web Map Server Implementation Specification 1.3.0.from http://portal.opengeospatial.org/files/?artifact_id=14416.

Biron, P. V., K. Permanente, et al. (2004). W3C Recommendation: XML Schema Part 2: Datatypes Second Edition.from http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html.

Bray, T., J. Paoli, et al. (2006). W3C Recommendation: Extensible Markup Language (XML) 1.0 (Fourth Edition), W3C.from http://www.w3.org/TR/2006/REC-xml-20060816.

Brundage, M. (2004). XQuery: The XML Query Language, Addison Wesley Professional. 0-321-16581-0.

Bunge, M. (1967). Scientific Research, Springer, Berlin **2**.

Buschmann, F., R. Meunier, et al. (1996). A System of patterns, John Wiley & Sons. 0-471-95869-7.

Cabrera, L. F., C. Kurt, et al. (2004). Whitepaper: An Introduction to the Web Services Architecture and Its Specifications. Web Services Technical Articles, Microsoft.from http://msdn.microsoft.com/webservices/webservices/understanding/advancedwebservices/default.aspx?pull=/library/en-us/dnwebsrv/html/introwsa.asp.

Clark, J. (1999). XSL Transformations (XSLT) Version 1.0.from http://www.w3.org/TR/xslt

Clark, J. and S. DeRose (1999). W3C Recommendation: XML Path Language (XPath) Version 1.0, W3.from http://www.w3.org/TR/1999/REC-xpath-19991116

Corcoles, J. E. and P. Gonzalez (2004). Integrating GML Resources and Other Web Resources. Proceedings of the Database and Expert Systems Applications, 15th International Workshop on (DEXA'04) - Volume 00, IEEE Computer Society.

Córcoles, J. E. and P. González (2001). A specification of a spatial query language over GML. Proceedings of the 9th ACM international symposium on Advances in geographic information systems. Atlanta, Georgia, USA, ACM Press.

Cox, S., A. Cuthbert, et al., Eds. (2002). OpenGIS® Geography Markup Language (GML) Implementation Specification version 2.1.2, Open GIS Consortium Inc.

Davis, M. (2003). JTS Topology Suite, Technical Specifications, Vivid Solutions.http://www.vividsolutions.com/JTS/bin/JTS%20Technical%20Specs.pdf.

Davis, M. (2004). JTS Javadoc, from http://www.vividsolutions.com/JTS/javadoc/index.html.

Davis, M. (2004). "TestBuilder Demonstration/Screenshots." from http://www.vividsolutions.com/JTS/screenShots.htm.

Davis, M. (2006). "Java Topology Suite Website." from http://www.vividsolutions.com/JTS/JTSHome.htm.

DeRose, S., R. D. Jr., et al. (2002). W3C Working Draft: XML Pointer Language (XPointer) W3C.from http://www.w3.org/TR/xptr/.

Directions Staff (2003). Microsoft SQL Server: Future Plans for Supporting Spatial Data. Directions Magazine.

ESRII. (2006, August 17 2006). "Interoperability and Standards;OGC Support." 2006, from http://www.esri.com/software/standards/ogc-support.html.

EXSLT.ORG. (2006). "EXSLT Community Website."   Retrieved October 10, 2006, from http://www.exslt.org/.

Fallside, D. C. and P. Walmsley (2004). W3C Recommendation: XML Schema Part 0: Primer Second Edition, W3C.from http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/.

Fernández, M., A. Malhotra, et al., Eds. (2006). XQuery 1.0 and XPath 2.0 Data Model (XDM), W3C Candidate Recommendation 11 July 2006, W3.

Harold, E. R. and W. S. Means (2002). XML in a Nutshell. Sebastopol, O'Reilly & Associates. 0-596-00292-0.

Harrie L., M. J. (2003). Real-time data generalisation and integration using Java. Geoforum Perspectiv.

Harrop, R. and J. Machacek (2005). Pro Spring, APress. 1-59059-461-4.

Holzner, S. (2001). Inside XSLT, Que. 0-7357-1136-4.

ISO/TC211 (2003). ISO 19107, ISO
https://committees.standards.org.au/COMMITTEES/IT-
004/PRIVATE/I0028/ISO%2019107%20.pdf.

JUMP (2006). Whitepaper: JUMP Unified Mapping Platform - Overview, JUMP.from http://www.jump-project.org/project.php?PID=JUMP&SID=OVER.

Köbben, B. (2005). [jts-devel] CW-CCW rule in JTS Normalize() function. J.-d. m. list.

Lake, R. (2004). GML, Geography Markup Language. Foundation for the geoweb, Wiley. 0-470-87154-7.

Lehto, L. and L. T. Sarjakoski (2005). "Real-Time generalization of XML-encoded spatial data for the Web and mobile devices." International Journal of Geographical Information Science **19**(8-9): 957-973.

Leung, T. W. (2004). Professional XML Development with Apache Tools, Wiley Publishing. 0-7645-4355-5.

Log4j Community. "Log4j Website and documentation." from http://logging.apache.org/log4j/docs/.

Marcos, E. (2005). "Software engineering research versus software development." ACM SIGSOFT Software Engineering Notes **30**(4): 1-7.

MySQL. "Introduction to MySQL Spatial Support." Retrieved July 1, 2006, from http://dev.mysql.com/doc/refman/5.0/en/gis-introduction.html.

Nah, F. F. H. (2004). "A study on tolerable waiting time: how long are Web users willing to wait?" Behaviour and Information Technology **23**: 153-163.

Nielsen, J. (1994). 5.5 Feedback. Usability Engineering. San Francisco, Morgan Kaufmann.

Novatchev, D. and S. Tyszko. (2006). "Two-stage recursive algorithms in XSLT." Retrieved September, 2006, from http://topxml.com/xsl/articles/recurse/.

OpenGeoSpatial (2002). GML 2.1.2 Schemas, from http://schemas.opengis.net/gml/2.1.2/.

Oracle (2005). Whitepaper: Oracle Locator: Location-Enabling Every Oracle Database., Oracle.http://www.oracle.com/technology/products/spatial/pdf/10gr2_collateral/locator_twp_10gr2.pdf.

Owens, B. (2006, August 26 2006). "What is Geoserver." Retrieved November 1, 2006, from http://docs.codehaus.org/display/GEOS/What+is+Geoserver.

Provost, W. "Whitepaper: Intetgrating Web Sevices with XSLT." Retrieved July, 2006, from http://www.xml.com/pub/a/ws/2003/09/30/integrating.html.

Refractions Research Chapter 4. Using PostGIS; GIS Objects. PostGIS Manual for version 1.1.3. P. R. R. f. d. a. packaging), Refractions Research.

Refractions Research. (2006). "UDig Website." Retrieved August 10, 2006, from http://udig.refractions.net/confluence/display/UDIG/Home.

Ryden, K., Ed. (2005). OpenGIS® Implementation Specification for Geographic information - Simple feature access - Part 1:Common architecture.

Ryden, K., Ed. (2005). OpenGIS® Implementation Specification for Geographic information - Simple feature access - Part 2: SQL option, Open Geospatial Consortium Inc.

Schulz, R. (2006). "GeoTools." Retrieved July 1, 2006, from http://docs.codehaus.org/display/GEOTOOLS/History.

Sedgewick, R. (2002). Divide and Conquer. Algorithms in Java, Addison-Wesley Profiessional.

Spring Community. (2006). "Spring Framework Community website." from http://www.springframework.org/.

Steve Wilson, J. K. (2000). Java Platform Performance: Strategies and Tactics, Addison-Wesley. 0201709694.

Stolze, K. (2003). SQL/MM Spatial: The Standard to Manage Spatial Data in Relational Database Systems. 10th Conference on Database Systems for Business, Technology and Web.

Sun (2004). addShutdownHook(java.lang.Thread). JavaTM 2 Platform Standard Ed. 5.0 Javadocs, Sun.from http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Runtime.html#addShutdownHook(java.lang.Thread).

Sun (2004). Object.hashCode(). JavaTM 2 Platform Standard Ed. 5.0, Javadoc, from http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Object.html#hashCode().

Sun (2006). JavaServer Pages Standard Tag Library, Sun.from http://java.sun.com/products/jsp/jstl/.

Thompson, H. S., D. Beech, et al. (2004). W3C Recommendation: XML Schema Part 1: Structures Second Edition, W3C.from http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/structures.html.

U.S. Census Bureau (2005). TIGER/Line® Technical Documentation, U.S. Census Bureau.

Vatsavai, R. R. (2002). GML-QL: A Spatial Query Language Specification for GML, Department of Computer Science and Engineering , University of Minnesota.

Vretanos, P., Ed. (2005). Web Feature Service (WFS) Implementation Specification, OpenGIS.

Vretanos, P. A., Ed. (2005). OpenGIS® Filter Encoding Implementation Specification, Open Geospatial Consortium Inc.

W3C. "What is XSL?"   Retrieved July, 2006, from http://www.w3.org/Style/XSL/WhatIsXSL.html.

Warnill, C., P. Soon-Young, et al. (2004). An Extension of XQuery for Moving Objects over GML. Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'04) Volume 2 - Volume 2, IEEE Computer Society.