XML DATA BINDING FOR GEOSPATIAL MOBILE APPLICATIONS

PH.D. DISSERTATION

Presented in Partial Fulfillment of the Requirements for the Degree
Doctor of Philosophy in the Institute of New Imaging Technologies of
Universitat Jaume I

By
Alain Tamayo Fong
M.Sc. Geospatial Technologies, M.Sc. Computer Science
Doctoral Programme in Integration of Geospatial Information

Supervisors:
Carlos Granell, Phd
Joaquín Huerta, Phd
Laura Díaz, Phd

This work has been partially supported by the “España Virtual” project (ref. CENIT 2008-1030) through the Instituto Geográfico Nacional (IGN).

Castellón de la Plana, 2011

To my parents

Enlace de Datos XML en Aplicaciones Geospaciales para Móviles

Tesis Doctoral
Alain Tamayo Fong
Directores: Dr. Carlos Granell Canut, Dr. Joaquín Huerta Guijarro,
Dr. Laura Díaz Sánchez
Castellón de la Plana, 2011

Las aplicaciones geoespaciales se han vuelto muy comunes tanto en entornos de escritorio, servidor, web, como en entornos móviles. El crecimiento de estas aplicaciones, así como del número de proveedores de datos, hace necesario la utilización de formatos y protocolos estándares para intercambiar información geospacial. En este sentido los estándares definidos por el *Open Geospatial Consortium* (OGC) constituyen un método efectivo para mejorar la interoperabilidad, dado que permiten la integración de datos procedentes de distintas fuentes. Estos estándares incluyen un conjunto de especificaciones de implementación de *interfaces de servicios web* y de *codificaciones de datos* para estandarizar la forma en que se comunican los componentes de un sistema distribuido. En particular, las interfaces de servicios web son conocidas como *Servicios Web OGC* (OWS)[1]. El gran número de instancias de servicios OGC disponibles en Internet dan fe del éxito alcanzado por estos estándares [FMLPNI10].

La irrupción de las aplicaciones geoespaciales en el mundo de la computación móvil ha sido posible gracias al aumento de la disponibilidad y capacidad de procesamiento de dispositivos de este tipo durante la última década. Esta tendencia también ha sido estimulada por la creciente demanda de los usuarios para ejecutar estas aplicaciones en sus teléfonos móviles [CCL09, Ant11]. Mientras que aplicaciones geoespaciales, tales como los sistemas de navegación, servicios web de mapas y

[1]OGC Web Services

globos virtuales son ya comunes en dispositivos móviles, el número de aplicaciones de este tipo basadas en estándares de OGC son muy escasas en estos dispositivos. En nuestra opinión, la complejidad de los protocolos definidos por los estándares dificulta enormemente el desarrollo de estas aplicaciones. Estos protocolos se basan en complejas estructuras de datos intercambiadas en formato XML, cuyo procesamiento eficiente en los dispositivos móviles es un problema importante debido a las limitaciones de hardware presentes en estos dispositivos [KLT07, WTS07].

La solución más común para hacer frente a la complejidad de los datos intercambiados en el ámbito de los servicios web, es el uso de herramientas tales como los generadores de código [VEG02, Her03, ZMCO04, BCG+05]. Estos generadores alivian el trabajo de los desarrolladores al producir automáticamente código de comunicación y procesamiento de XML, tanto para el lado cliente como para el lado servidor. Esta solución no se puede aplicar fácilmente a los *Servicios Web OGC* debido al gran tamaño y complejidad de los esquemas asociados a sus especificaciones. Aunque es posible que el código generado satisfaga las necesidades de un sistema sin grandes limitaciones de capacidad de procesamiento, memoria o tamaño del código generado, seguramente no satisfará las necesidades de dispositivos con restricciones de hardware como las que encontramos en la mayoría de los dispositivos móviles.

Objeto y Objetivos de la Investigación

La importancia del código de procesamiento de XML y su impacto en el rendimiento y el tamaño de las aplicaciones han sido tradicionalmente subestimadas. Muchos diseñadores trabajan sobre la hipótesis de que independientemente de cuán complejos sean los esquemas construidos, es posible generar fácilmente código para su procesamiento utilizando algún tipo de herramienta. Aunque esto es cierto para algunos casos, como aplicaciones que se ejecutan en un entorno servidor asumiendo que una gran cantidad de recursos de hardware está disponible, no ocurre lo mismo para otros entornos con otras limitaciones. Este es precisamente el caso de las aplicaciones geospaciales móviles basadas en estándares de OGC.

El objetivo fundamental de este trabajo es la definición de una solución que permita la generación automática de código de procesamiento de XML personalizado para aplicaciones móviles basadas en estándares de OGC. La solución propuesta se basa en la observación de que las implementaciones reales de estos estándares suelen utilizar solo un subconjunto de sus esquemas y permite la extracción automática de este subconjunto a

partir de un grupo de documentos XML que deben ser procesados por una aplicación en particular. La utilización de este subconjunto, unido a otra información relacionada con la utilización de los esquemas, posibilita la generación de un código más compacto que se adapta mejor a las restricciones de los dispositivos móviles actuales. La efectividad de esta solución se demuestra a través de la implementación de aplicaciones de ejemplo entre las que se incluye un cliente para la especificación de *Servicio de Observación de Sensores* (SOS) dirigido a la plataforma Android.

Planteamiento y Metodología

El primer paso de nuestra investigación ha consistido en demostrar una de nuestras hipótesis fundamentales, que es que la mayoría de las versiones actuales de esquemas asociados a los estándares OGC son grandes y complejos, lo que provoca que los generadores de código disponibles para plataformas móviles produzcan un código binario que no satisface algunas de las limitaciones presentes en estas plataformas. En concreto, el tamaño del código generado no es adecuado para la ejecución en un dispositivo con recursos limitados. Para satisfacer este objetivo se ha utilizado un conjunto de métricas de software para medir la complejidad de los esquemas asociados a los estándares ([LKR05, MSY05, BM09b]), mostrando que esta complejidad no solo es bastante alta sino que también continúa creciendo. La mayoría de estas métricas se han extraído de la bibliografía existente en el tema, aunque ha sido necesario la definición de nuevas métricas para medir la influencia que tienen los mecanismos de subtipos presentes en los esquemas XML en la complejidad final de dichos esquemas.

El proceso mismo de medición de la complejidad de los esquemas ha sugerido diferentes soluciones para tratar con esta complejidad. Algunas de estas soluciones son expuestas en este documento y sus beneficios son medidos utilizando métricas de software concretas. Partiendo de estas posibles soluciones se ha desarrollado en detalle aquella que consideramos como la más pragmática, que permite la ejecución de código de procesamiento de XML para clientes móviles sin necesidad de modificar la infraestructura de servidores basados en estándares OGC existente.

La solución que se presenta en este documento para lidiar con la complejidad de los esquemas se denomina *generación de código de enlace de datos XML basada en instancias (Instance-based XML data binding)*. Como se ha mencionado anteriormente, esta solución se basa en la observación de que las implementaciones reales de sistemas basados en los

estándares utilizan solamente un subconjunto de sus esquemas. Esto permite el desarrollo de aplicaciones personalizadas que presentan un mejor rendimiento que las basadas en los requisitos genéricos. Para demostrar la suposición de que las aplicaciones reales solo utilizan una parte de los estándares se incluye en este documento un estudio que intenta medir el porcentaje de los esquemas relacionados con la especificación SOS que es utilizado por un conjunto de servidores disponible en Internet. Se ha elegido SOS porque sus esquemas están entre los más complejos de todas las especificaciones OGC.

La generación de código de enlace de datos XML basada en instancias esta compuesta de dos pasos. El primer paso consiste en la extracción automática del subconjunto de los esquemas de una especificación que es utilizado por una aplicación específica. Este subconjunto se calcula basándose en la suposición de que se dispone de un grupo representativo de documentos XML que deben ser manipulados por la aplicación. El segundo paso consiste en utilizar toda la información extraída en el paso anterior para generar el código optimizado tanto como sea posible para una plataforma de dispositivos móviles específica. Se ha dividido el proceso en dos etapas, porque de esta forma los resultados de la primera etapa se puede utilizar de manera independiente de la plataforma, lo que significa que el subconjunto de esquemas extraídos se pueden utilizar para generar código con cualquier generador disponible para cualquier sistema operativo o lenguaje de programación. El segundo paso genera código para una plataforma y lenguaje de programación específicos, Android y Java, en nuestro caso. La elección de la plataforma móvil y el lenguaje de programación se ha realizado sobre la base de la disponibilidad de herramientas maduras para implementar un prototipo que se utilizará para construir dos aplicaciones de ejemplo.

Aportaciones Originales

Las aportaciones de este trabajo se incluyen principalmente en el tema de procesamiento de XML para dispositivos móviles, aplicadas en particular al tema de los servicios web estándares para el intercambio de información geospacial. También se presentan algunas aportaciones de menor relevancia al tema de métricas de complejidad para esquemas XML. Estas aportaciones se pueden resumir de la siguiente manera:

1. Un estudio de la complejidad de los esquemas asociados a los estándares de OGC. El estudio utiliza métricas nuevas o existentes

anteriormente para medir diferentes aspectos de la complejidad de los esquemas.

2. Un proceso de generación de código de procesamiento de XML para aplicaciones basadas en estándares de OGC dirigidas a dispositivos móviles. El proceso se compone en primer lugar de un algoritmo que permite la simplificación de los esquemas según las necesidades de aplicaciones específicas. Se incluye además, la implementación de un generador de código de enlace de datos XML que utiliza el algoritmo mencionado anteriormente para producir código optimizado para la plataforma Android.

3. También se presenta una evaluación de desempeño del código generado con nuestra herramienta con datos reales de sensores extraídos de servidores SOS.

Conclusiones

En este trabajo se ha realizado un estudio de la complejidad de los esquemas de los estándares de OGC. Los resultados han demostrado que al menos la mitad de los estándares analizados pueden ser considerados como grandes y complejos de acuerdo con las métricas incluidas en el estudio. Adicionalmente, se han propuesto un conjunto de nuevas métricas para mostrar diferentes puntos de vista de los efectos del uso de los mecanismos de subtipos de los esquemas XML en su complejidad. También se han presentado casos de uso en los que estas métricas pudieran ser aplicadas, por ejemplo para evaluar el impacto de decisiones de diseño, evaluación de la eficacia de diferentes soluciones para hacer frente al problema de la complejidad de esquemas, para detectar posibles problemas de diseño, etc.

También se ha presentado una solución que permiten la generación personalizada de código de procesamiento de XML para aplicaciones basadas en OGC dirigidas a dispositivos móviles. La solución está compuesta de un algoritmo que permite la simplificación de esquemas de datos según las necesidades de aplicaciones específicas, y de un generador de código dirigido a la plataforma Android. Los resultados de la aplicación del algoritmo a un escenario de uso real de casos han mostrado que permite una reducción sustancial de los esquemas originales de alrededor del 90% de su tamaño. Esta enorme reducción en el tamaño de los esquemas se traduce en una reducción del código binario de más del 80%

en el caso de uso considerado. La transformación realizada por este algoritmo se realiza a nivel de esquema y sin hacer ninguna suposición sobre la plataforma de destino, en consecuencia, los esquemas de salida se puede utilizar en combinación con cualquier generador de código de enlace de datos XML. El generador de código implementado utiliza información acerca de cómo los documentos XML hacen uso de sus esquemas asociados para optimizar el tamaño del código generado. Esta herramienta ofrece un grupo de funciones que permiten la producción de un código muy compacto, y su eficacia ha sido demostrada a través de una serie de experimentos y la construcción de dos aplicaciones de ejemplo. Los experimentos han demostrado que el código generado por nuestra herramienta es sustancialmente menor que el código generado por otras herramientas.

Futuras Líneas de Investigación

Actualmente se están siguiendo varias líneas de investigación derivadas de los resultados presentados en este documento. En el tema de las implementaciones de OWS para dispositivos móviles se trabaja en la exploración de formatos alternativos para el intercambio de información entre clientes y servidores. El uso de XML para codificar la información añade una sobrecarga que puede no ser soportada por algunos dispositivos con recursos limitados. Otra línea de investigación está relacionada con la generalización de los resultados presentados para SOS al resto de las especificaciones de OGC. Las aplicaciones de ejemplo construidas como demostración tienen mucho en común, presentan retos similares y cuestiones sin resolver en las que se podría profundizar aún más en el contexto de otras especificaciones. La resolución de algunos problemas pendientes, tales como el uso de las técnicas de generación de código para lógica de negocio e interfaz de usuario de este tipo de aplicaciones puede reducir la complejidad del proceso de construcción de las mismas. Por último, el tema de la complejidad de los esquemas también puede ser explorado con mayor profundidad. El conjunto de características de los esquemas que puede medirse a través de métricas de software que permitan una mejor comprensión de estos esquemas y que a la vez faciliten la implementación de aplicaciones basadas en los mismos es potencialmente grande.

Abstract

Geospatial applications such as navigation systems, web mapping services, and even virtual globes, are more common everyday on mobile devices. Nevertheless, applications based on OGC standards, in these devices are still a few. In our opinion, the complexity of the protocols defined by the standards makes the task of writing reliable and efficient mobile applications based on them a difficult task.

The large size of the XML schemas associated to OGC specifications causes that they cannot be easily used to produce XML processing code targeted to mobile devices. The task of writing this code manually is recognised to be difficult and error-prone. On the other hand, the use of a code generator in the presence of such large schemas may produce results that do not meet all the requirements of mobile applications, which have severe performance limitations when compared to desktop systems.

In this context, we present in this dissertation an approach that allows the automatic generation of XML processing code for mobile applications based on OGC standards. Using this approach it is possible to generate customised code with a small size that can be easily accommodated in a resource-constrained device. Our solution, named *Instance-based XML data binding* extracts useful information from a subset of XML documents that must be manipulated by the application that allows the optimisation of the generated code. The approach has been implemented in a code generator targeted to Android mobile devices. Its usefulness is demonstrated by building a set of sample applications, including a full-fledged mobile client for the Sensor Observation Service (SOS) specification. We also present a complexity study for the specification schemas. This study provides a quantitative way to analyse and measure the complexity of these schemas.

Acknowledgements

It is a pleasure to thank those who made this thesis possible.

First and foremost I would like to thank my advisors for the support and guidance they provided me over the years. I would like to show my gratitude to my principal supervisor, Carlos Granell, for his good advice, patience, swift and accurate comments, and words of encouragement. I would also like to thank to my co-advisor Joaquín Huerta for always being able to find a solution to all of the problems I encountered along the way. Additionally, I thank Laura Díaz for her constructive comments and valuable exchange of ideas pertaining to this research.

Thank you to the experts in my thesis committee: Luis Eduardo Leyva, Pedro Muro Medrano, and Theodor Foerster. Their valuable comments have definitely improved the quality of this dissertation.

I am indebted to many of my colleagues for providing a stimulating and fun environment. Thanks to all of the former and current members of the Geospatial Technologies Research Group for their support and friendship. I would like to especially thank Pablo Viciano for helping me with the implementation of experiments and sample applications included in this document.

Last, I would like to thank my family and friends for their support in the new life I started four years ago. Thanks to Karen for always being at my side all these years. Thanks to Carlos Abargues for his unconditional friendship and support. Thanks to Hebert for being patient in correcting my English writing.

Thanks to my friends from the Master degree in Geospatial Technologies: Rania, Ledjo, Luc, Tanmoy, and the others. Thanks to my compatriots sharing this new life with me; they made me feel at home: Arturo Canler, Guillermo Matos, Luis Enrique Rodríquez, Arturo Quintana, and all the rest.

Table of Contents

List of Figures xix

List of Tables xxiii

Part I Introduction

1 Introduction 3
1.1 Motivation 3
1.2 Research Methodology 4
1.3 Contributions 6
1.4 Structure of the Dissertation 7

Part II Background

2 XML, XML Schema and Web Services 11
2.1 XML 11
2.2 XML Schema 13
2.2.1 Type Derivation 14
2.2.2 Document Composition 15
2.2.3 Criticism 16
2.3 XML Processing 17
2.3.1 Vocabulary-independent Data Access Interfaces 18
2.3.2 XML Data Binding 18
2.4 Web Services 21
2.4.1 Approaches to Web Service Development 22
2.5 Concluding Remarks 23

3 OGC Web Services 25
3.1 OWS Architectural Principles 26

3.2 OWS Specifications Overview 27
3.3 OWS Implementations 29
3.3.1 Server Side 30
3.3.2 Client Side 30
3.4 Sensor Observation Services 32
3.5 Concluding Remarks 36

4 Mobile Computing 37
4.1 Mobile Hardware 37
4.2 Mobile Software 38
4.2.1 Android 39
4.3 Web Services for Mobile Devices 40
4.4 XML Processing for Mobile Devices 41
4.5 OWS Implementations for Mobile Devices 42
4.6 Concluding Remarks 44

Part III XML Processing for Geospatial Mobile Applications

5 Complexity of OWS Schemas 47
5.1 Related Work 49
5.2 Metrics 50
5.2.1 C(XSD) Metric Definition 52
5.2.2 Subtyping-related Metrics 54
5.2.3 Measurement Process Description 58
5.3 Results 58
5.3.1 XML-Agnostic Metrics 58
5.3.2 XSD-Aware Metrics 60
5.3.3 C(XSD) 63
5.3.4 Subtyping Metrics 65
5.3.5 Discussion 68
5.4 Practical Use of Metrics 69
5.4.1 Use Case Scenario: Evaluating Design Decisions 69
5.4.2 Use Case Scenario: Studying Specifications Evolution 71
5.4.3 Other Possible Scenarios 72
5.5 Pragmatic Solutions to Complexity 73
5.5.1 XML Data Binding Code Generators 73
5.5.2 Profiles 75
5.5.3 Using the Linked Data Style 76
5.6 Concluding Remarks 77

6 Instance-based Schema Simplification **79**
6.1 Instance-based XML Data Binding 80
6.1.1 Instance-based Schema Simplification 81
6.2 Notation 83
6.3 Simplification Algorithm 87
6.3.1 Helper Functions 87
6.3.2 Algorithm 88
6.4 Experimentation 90
6.4.1 Gathering Input Instance Files 91
6.4.2 Generating the Output Subset 92
6.4.3 Generating Binary Code 92
6.5 Concluding Remarks 94

7 XML Data Binding for Mobile Devices **97**
7.1 XML Data Binding Code Generator 98
7.1.1 Supported Features 98
7.2 Basic Mapping of Schema Components 100
7.2.1 Mapping Complex Types 101
7.2.2 Mapping Simple Types 105
7.2.3 Mapping Global Elements 106
7.3 Supported Features Explained 107
7.3.1 Support for Instance-based Code Generation 108
7.3.2 Source Code Based on Simple Code Patterns 113
7.3.3 Tolerate Common Validation Errors 113
7.3.4 Collapse Elements Containing Single Elements 114
7.3.5 Disabling Parsing/Serialization Operations 115
7.3.6 Ignoring Sections of XML Documents 116
7.4 Experimentation 116
7.4.1 DBMobileGen 116
7.4.2 Experiment Description 117
7.4.3 Results 117
7.5 Sample Applications 120
7.5.1 WPS Basic Client 121
7.5.2 SOS Mobile Client 124
7.5.3 Challenges and Open Issues 127
7.6 Concluding Remarks 129

Part IV Experiments

8 Empirical Study of SOS Server Instances **133**

8.1 SOS Server Instances 134
8.2 Limitations of the Study 134
8.3 Dataset Description 135
8.4 Results 135
8.4.1 Capabilities Files 136
8.4.2 Procedure Description Files 142
8.4.3 Observation Files 147
8.5 Subset of XML Schemas Used 149
8.5.1 GML 149
8.5.2 SOS 151
8.6 Discussion 152
8.7 Concluding Remarks 155

9 Performance Evaluation 157
9.1 Performance Considerations for Java Programs 157
9.1.1 Startup Performance 159
9.1.2 Steady-State Performance 159
9.2 Experimental Setup 160
9.2.1 Test Datasets 161
9.2.2 Hardware and Software 162
9.3 Results 163
9.3.1 CAPS Dataset 163
9.4 Discussion 167
9.5 Concluding Remarks 169

Part V Conclusions and Future Work

10 Conclusions 173
10.1 Contributions 173
10.2 Future Work 175

Bibliography 177

Part VI Appendices

A List of SOS server instances 195

B Performance Results 197
B.1 Sensor Descriptions Dataset 197

B.1.1 Mobile Configuration 198
B.1.2 Mac OS X Laptop 198
B.1.3 Windows PC 199
B.2 Observations Dataset 199
B.2.1 Mac OS X Laptop 200
B.2.2 Windows PC 202
B.3 Measurements Dataset 203
B.3.1 Mobile Configuration 203
B.3.2 Mac OS X Laptop 204
B.3.3 Windows PC 204

C Parsers Comparison 205
C.1 Mobile Configuration 206
C.2 Mac OS X Laptop 207
C.3 Windows PC 208

List of Figures

2.1 Graph of dependencies between schema components 16
2.2 XML data binding code generation process 19
3.1 Dependencies between OGC specifications 29
3.2 CartoCiudad web portal 32
3.3 Integrated ocean observing system web portal 33
3.4 Interaction with the SOS server 35
4.1 gvSIG Mini screenshots. To the left some of the options available are shown over a WMS layer. To the right the current location is shown using Open Street Map as base map layer 43
5.1 Graph of relations between schema component for schema fragment in Listing 2.3 57
5.2 Values for the LOC metric 60
5.3 Distribution of the number of schema components in the specification schemas 62
5.4 C(XSD) values for R=2 65
5.5 XSD-aware simple metrics values for WMS 1.3.0 and its merging with OWS and GML 70
5.6 Following GML evolution through metrics 72
5.7 Comparing size of code (KBs) for different code generators 74
5.8 Comparing effectiveness of different approaches to deal with the complexity of schemas 76
6.1 Instance-based XML data binding code generation process 81
6.2 Relations between information items in XML documents (right) and schemas components defining its structure (left) 82
6.3 Graph of relations in schema fragment in Listing 2.3 85
6.4 Location of air pollution control stations in the Valencian Community 91
6.5 Simple schema metrics for original and simplified schemas 94
7.1 Flow diagram for the code generation process 99
7.2 Elements in the substitution group of *gml:_Feature* 110

7.3 Elements in the substitution group of *gml:_Feature* for the a subset of the SOS schemas 110
7.4 Type definition hierarchy for GML 3.1.1 112
7.5 Size of generated code for full schemas 119
7.6 Size of generated code for simplified schemas 120
7.7 WPS Google Maps-based client 122
7.8 WPS basic client architecture. 122
7.9 Selecting or adding servers 126
7.10 Showing sensors stations in the map 126
7.11 Specifying Filters 127
7.12 Tables and charts 128
8.1 Support of SOS operations in actual server instances 138
8.2 Number of servers classified by the number of offerings, procedures and observed properties 142
8.3 Observation offerings in North America 143
8.4 Hierarchy of observation types 148
8.5 Dependencies of SOS from other specifications 150
8.6 Overall number of schema components vs actually used components in SOS 152
9.1 Typical execution time pattern followed by a Java program 158
9.2 Execution times of XBinder and DBMG for CAPS-S dataset (Mobile Scenario) 165
9.3 Execution times of XBinder and DBMG for CAPS-L dataset (Mobile Scenario) 166
9.4 Execution times for CAPS-S dataset (PC Scenario 1 - Mac OS X Laptop) 167
9.5 Execution times for CAPS-L dataset (PC Scenario 1 - Mac OS X Laptop) 168
9.6 Execution times for CAPS-S dataset (PC Scenario 2 - Windows PC) 169
9.7 Execution times for CAPS-L dataset (PC Scenario 2 - Windows PC) 170
B.1 Execution times for SD dataset (Mobile Scenario) 198
B.2 Execution times for SD dataset (PC Scenario 1 - Mac OS X Laptop) 198
B.3 Execution times for SD dataset (PC Scenario 2 - Windows PC) 199
B.4 Execution times for OBS-S dataset (PC Scenario 1 - Mac OS X Laptop) 200

B.5 Execution times for OBS-L dataset (PC Scenario 1 - Mac OS X Laptop) 200
B.6 Execution times for OBS-S dataset (PC Scenario 2 - Windows PC) 202
B.7 Execution times for OBS-L dataset (PC Scenario 2 - Windows PC) 202
B.8 Execution times for MEA dataset (Mobile Scenario) 203
B.9 Execution times for MEA dataset (PC Scenario 1 - Mac OS X) 204
B.10 Execution times for MEA dataset (PC Scenario 1 - Mac OS X) 204
C.1 Execution times for CAPS-S dataset (Mobile Scenario) 206
C.2 Execution times for CAPS-L dataset (Mobile Scenario) 206
C.3 Execution times for CAPS-S dataset (PC Scenario 1 - Mac OS X Laptop) 207
C.4 Execution times for CAPS-L dataset (PC Scenario 1 - Mac OS X Laptop) 207
C.5 Execution times for CAPS-S dataset (PC Scenario 2 - Windows PC) 208
C.6 Execution times for CAPS-L dataset (PC Scenario 2 - Windows PC) 208

List of Tables

3.1 Geospatial web service interfaces and data encodings 28
3.2 SOS operation profiles 34
5.1 Lines of code (LOC) and number of files (#F) 59
5.2 Number of complex types (#CT) 61
5.3 Main XML features metrics (except #CT) 62
5.4 Use of wildcards 63
5.5 C(XSD) values for OWS specifications 64
5.6 C(XSD) values for encoding specifications 64
5.7 Use of subtyping mechanisms 65
5.8 DPR values for the OWS specifications 66
5.9 DPF values for the OWS specification schemas 67
5.10 SRR values for the OWS specification schemas 67
6.1 Comparing original and simplified schema sets 93
6.2 Comparing size of code (KBs) for original and simplified schema sets 93
7.1 Comparing size of code (KBs) for original and simplified schema sets 118
7.2 Comparison between the number of complex types in the schemas related to the WPS Basic Client before and after applying the instance-based simplification algorithm 123
8.1 Dataset description 135
8.2 Most frequent validation errors for capabilities files 137
8.3 Operations supported by the server instances 138
8.4 Support of filters 140
8.5 Formats supported to represent observation information 141
8.6 Most frequent validation errors for sensor description files 144
8.7 Most frequent validation errors for observation files 147
8.8 Comparison between overall number of components in GML and number of components actually used 150
8.9 Distribution of SOS schema files by specification 151

8.10 Comparison between the overall number of components in SOS and number of components actually used 152
9.1 Datasets description 161
9.2 Generated parsers for each dataset 162

Part I

Introduction

CHAPTER 1

Introduction

Geospatial applications have become commonplace in desktop, server, web and mobile environments. As nowadays the number of data providers and clients all around the world is continuously growing, standard ways of exchanged data are needed. In this regard, standards defined by the *Open Geospatial Consortium* (OGC) are a vehicle for *interoperability* as they allow the integration of data coming from different sources. OGC has defined a set of *web service interfaces* and *data encondings* to exchange geospatial information in a standard way. The success of these standards is witnessed by the large number of service instances available online [FMLPNI10].

The irruption of geospatial applications in the mobile computing world has been possible because the availability of mobile devices and its capabilities has increased rapidly over the last decade. This trend has also been stimulated by the increasing demand from users to run such applications on their mobile phones [CCL09, Ant11].

1.1 Motivation

While geospatial applications such as navigation systems, web mapping services and even virtual globes are more common everyday on mobile devices, applications based on OGC standards in these devices are still a few. In our opinion, the complexity of the protocols defined by the standards makes the task of writing reliable and efficient mobile applications based on them an arduous task. These protocols are based on

complex data structures exchanged in XML format, which reputed as being too verbose [Bar11], causes that they cannot be efficiently processed in mobile devices because of their hardware limitations [KLT07, WTS07].

The common solution to deal with all this complexity in the web service realm has been the use of tools such as code generators [VEG02, Her03, ZMCO04, BCG+05]. Using generators, developers are relieved from the burden of producing communication and/or XML processing code manually for web service end-points. While the interaction between clients and servers in *OGC Web Services*, if analysed in the context of individual specifications, is not really complex in terms of communication (low average number of operations per specification, few restrictions regarding the order of operations, etc.), the opposite happens in terms of the complexity of the exchanged data. An evidence of this statement is the ever growing size of the XML schemas associated to these specifications, used to describe the structure of the exchanged data.

The importance of XML processing code and its impact in performance and size of applications have been traditionally underestimated [NJ03, AIM+04]. Many designers work with the assumption that no matter how complex schemas are, code to process XML documents based on them can be easily produced by using some sort of generator. Although, this is true in some scenarios, there are still others where finding a code generator that produce code meeting all the requirements of an application (performance, size, scalability, etc.) may be very difficult or even impossible. This is precisely the case of building web services endpoints based on OGC standards for mobile devices, which have severe performance limitations when compared to a desktop system.

In this context, we are trying to find better ways to build mobile applications based on OGC standards. The first step is to define an approach, and a set of supporting tools, that allows the automatic generation of XML processing code for mobile applications based on the aforementioned standards. These tools must cope with the problems related with the processing of large schema files, which often prevent existing generators from producing code that meet the users' expectation.

1.2 Research Methodology

Finding ways to overcome the complexity of the information exchange protocols to make them suitable for mobile devices has been our main research goal. We have started by trying to prove our main assumption, which is that *most of the current versions of schemas associated*

to OGC Web Service specifications are big and complex, which provokes that code generators produce a binary code that does not meet application requirements for mobile devices. Specifically, generators produce code with large binary sizes that is not adequate for execution in a device with constrained resources. To achieve our main goal we have used a set of software metrics to measure the complexity of the specification schemas to show that this complexity is not only high but it is also growing [LKR05, MSY05, BM09b]. When metrics to measure some aspects of interest did not exist previously we have introduced new metrics, e.g., metrics to measure the influence of the subtyping mechanisms of XML Schema in complexity. We also measure the impact of the complexity of the schemas in the final code of actual systems, to reinforce the idea that the complexity we are facing is indeed very real.

The process of measuring complexity has itself suggested different solutions to deal with it. We outline some solutions and measure its benefits using concrete software metrics. From these solutions, we have developed further the one we consider the most pragmatic, which will allow the implementation of XML processing code for mobile clients without modifying the existing infrastructure of available standard-based servers.

The solution proposed here is based on the observation that *actual system implementations do not use all of the capabilities included in the specifications schemas.* This allows the development of customised applications that will present better performance than those based on generic requirements. Although, the assumption above could seem obvious to some extent, we present a study about how a set of *Sensor Observation Service* (SOS) server instances uses the schemas associated to this specification. This not only to prove that the assumption is true, but also to try to quantify how much of the schemas is used in real implementations. We have chosen SOS because of our previous experience with the specification [Tam09, THG$^+$09], and also because the SOS schemas are among the most complex schemas of the OGC specifications.

Our solution, named *Instance-based XML data binding code generation* consists in the automatic extraction of the subsets of the specification schemas used on a given application, and the use of such subset to generate XML processing code adapted to the constraints of mobile devices. The term *XML Data Binding* is used to refer to the process of mapping XML data to application objects [McL02, LM07]. This process is often accomplished by using code generators that offers better productivity and improved type-checking of XML data. As mobile platforms present several hardware constraints, we restrict our main objective to the reduction

of the size of code generated from the schemas, allowing it to be easily accommodated on a mobile application. Secondary objectives are to allow the successful processing of as many XML documents as possible even if they do not follow correctly the structure defined by the specifications, and to ensure that the previous goals are met without sacrificing execution speed.

Instance-based XML data binding code generation, is a two-step process. The first step extracts the subset of the specification schemas that is used by the application, based on the assumption that a representative subset of the XML instances that must be manipulated by the application is available. The second step consists in using all of the information extracted in the previous step to generate XML processing code as optimised as possible for the target platform. We have divided the process in two steps because this way the results of the first step can be used in a platform-neutral way, meaning this that the extracted schemas subset can be used to generate code using any other code generator available for any operating system or programming language. The second step produces code for a specific platform and programming language, Android and Java, respectively. The choice of mobile platform and programming language has been made based on the availability of mature tools to implement a prototype, which will be used to build a full-fledged sample application. This application will be an SOS client for the Android platform.

1.3 Contributions

The contributions of this work are mainly included in the topic of XML processing for mobile devices, with a particular application to the subject of OGC Web Services and Encodings. We also present some minor contributions to the topic of complexity metrics for XML Schema. These contributions are summarised as follows:

1. A comprehensive complexity study for the OGC specification schemas is presented. The study uses existing and new metrics to measure different aspects of the complexity of XML schemas.

2. A two-step process to generate XML processing code for OGC-based applications targeted to mobile devices. The process is composed first, by an algorithm that allow the simplification of specification schemas according to the needs of particular applications. It also

includes the implementation of a XML data binding code generator that uses the algorithm mentioned before to produce source code for OGC-based clients targeted to the Android Platform.

3. A performance evaluation of the XML processing code generated with our tool in the context of the SOS specification.

1.4 Structure of the Dissertation

This dissertation follows the structure described next. *Part I*, which contains the present chapter, presents an introduction to the work presented here, explaining the motivation, research methodology and the main contributions. In *Part II*, we include the necessary background on the topics our work is related to, with the aim to provide a context to the reader for the rest of the dissertation. Part II is composed of three chapters. *Chapter 2* presents an introduction to topics related to XML, XML Schema and Web services; *Chapter 3* presents briefly web service interfaces and encodings defined by OGC; and *Chapter 4* introduces mobile computing.

Part III includes the core of our work. *Chapter 5* presents a comprehensive complexity study of the specification schemas. It also analyses briefly some possible solutions to the problem of the complexity of schemas. *Chapter 6* presents the algorithm used to calculate the subset of the specification schemas that is used by a given application. Next, *Chapter 7* exposes details of the process of generating XML processing code for Android mobile applications.

Part IV presents experiments designed to prove some of our assumptions and to prove that the generated code meets its initial requirements. *Chapter 8* presents an empirical study of existing SOS server instances. Although this study was originally conceived to determine an approximation of how much of the schemas was used in actual implementations it was extended to consider other aspects related to how the SOS specification is used in practice. *Chapter 9* presents a comprehensive performance study to show that our goals are met without sacrificing execution speed. Last, Part V presents in *Chapter 10* the conclusions of our work and different future lines of research that can be followed in this topic.

Part II

Background

CHAPTER 2

XML, XML Schema and Web Services

The *eXtensible Markup Language* (XML) is a textual data format that has reached a great success in the Internet era. XML documents are similar to HTML documents, but do not restrict users to a single vocabulary, which offers them a great deal of flexibility to represent information. To define the structure of documents within a certain vocabulary, schema languages such as *Document Type Definition* (DTD) or *XML Schema*, can be used. XML has become the main exchange format for Web services.

In this chapter, we present an introduction to the concepts and technologies related to *XML*, *XML Schema* and *Web services* used in the remainder of this dissertation.

2.1 XML

XML is a text format originally designed to meet the challenges of large-scale electronic publishing [W3C08]. It is derived from the *Standard Generalized Markup Language* (SGML) [ISO86] and it defines a set of rules to encode documents in a machine-readable form. XML documents use *tags* to structure the information, but these tags are not restricted to a specific vocabulary. Instead, documents creators can define and use their own vocabulary. XML documents must only follow a few syntactic rules to be considered *well-formed.* The information contained in these documents follows a *tree structure*, specifically an *unranked tree*,a finite labelled tree where nodes can have an arbitrary number of children [Nev02].

Listing 2.1: Simple XML document

```
<?xml version="1.0"?>
<Container>
  <item>
    <baseElement>String Value 1</baseElement>
  </item>
  <item>
    <baseElement>String Value 2</baseElement>
  </item>
</Container>
```

XML instances[1] are made up of *information items*, which can be *elements* or *attributes*. Elements are delimited by *start* and *end* tags, and they are identified by a *name*. Elements may contain other elements, referred to as their *children*, or just a *value* within its start and end tags[2]. Attributes are name/value pairs that are contained inside element start tags. The information items of an XML document form its *Information Set* or *Infoset*, which is specified in [W3C04b]. The Infoset specification is also used as information model for other XML-related technologies such as XPath[3][W3C99b] or XML Schema (see Section 2.2).

Listing 2.1 shows an example of a simple XML document. It defines a *root element* called *Container* that has a set of *items* inside. *Items* also include other child elements. XML documents can only contain a single root element. Listing 2.2 presents a second example that shows the use of attributes. In this case, *item* has an attribute called *xsi:type* with the value *"Child"*[4].

XML has gained a lot of popularity, being used extensively as exchange format in the Web. It has been adopted as the most common form of encoding information exchanged by Web services [Kay03, WG08, Wil03] (see Section 2.4). [Kay03] attributes this success to two reasons. The first one is that the XML specification is accessible to everyone and it is reasonably simple to read and understand. The second one is that

[1]The terms *document* and *instance* are used interchangeably to denote data encoded in XML.

[2]It is possible to mix child elements and values if the *mixed content model* is used, but for the sake of simplicity we are ignoring this possibility here.

[3] XPath defines a syntax to address parts of an XML document.

[4]The purpose of this attribute is introduced in Section 2.2.1

Listing 2.2: XML document illustrating the use of the *xsi:type* attribute

```
<?xml version="1.0"?>
<Container>
  <item xsi:type="Child">
    <baseElement>Base String Value</baseElement>
    <chdElement>Child String Value</chdElement>
  </item>
</Container>
```

several tools for processing XML are readily available. We add to these reasons that as XML is *vocabulary-agnostic*, it can be used to represent data in basically any domain. On the other hand, XML is reputed as being too verbose, e.g. [Bar11] compared the representations of a fairly large dataset in XML and a simple application-specific format. The results show that data encoded in XML was 180 times larger than the application specific format, which also had a significant impact on performance.

2.2 XML Schema

The *XML Schema definition language* is used to define the structure of information contained in XML documents [W3C04c, W3C04d]. The language itself is defined using XML. Each XML schema file has a root element named *schema* that contains the definition of the structure of a set of XML documents. This structure is expressed through schema components such as *complex types, simple types, elements, attributes, and element and attribute groups.*

An XML document conforming to the structure defined in some schema is said to be *valid* against the schema. We denote the set of all valid files against a schema S as *I(S).* If schemas are available when an XML document is parsed, its infoset can be annotated with type information from the schemas. This *extended infoset* is known as *Post Schema Validation Infoset* (PSVI).

XML Schema, in addition to a set of predefined types defined by the language, allows users to define their own types. Types are used to define the structure of *elements.* Element declarations in schemas are matched to elements in XML documents. Elements in schemas can be either *local* or *global.* An element is local when it is defined inside a type or element group definition. It is global when it is defined as a child element of

Listing 2.3: XML Schema fragment

```
<complexType  name="Base">
  <sequence>
    <element name="baseElement"  type="string"/>
    <element ref="baseElement2"  minOccurs="0">
  </sequence>
</complexType>

<complexType name="Child">
  <complexContent>
    <extension base="Base">
      <sequence>
        <element name="chdElement"  type="string"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>

<complexType name="ContainerType">
  <sequence>
    <element name="item" type="Base" maxOcurrs="unbounded"/>
  </sequence>
</complexType>

<element name="Container" type="ContainerType" />
<element name="baseElem2" type="string" />
```

the root element of a schema file. Global elements are useful to reuse element declarations on the schemas by defining them in a single place and referencing them latter from different parts of the schemas. They also define which elements can be used as root elements of XML documents.

Listing 2.3 shows a fragment of an XML schema file. This fragment contains the declaration of three global complex types and two global elements. XML instances presented in Listings 2.1 and 2.2 are *valid* against this schema, meaning that they satisfy the constraints defined on it. For the sake of simplicity we have omitted the *schema* root element.

2.2.1 Type Derivation

XML Schema provides a derivation mechanism to express subtyping relationships. The mechanism allows types to be defined as subtypes of existing types, either by extending their content model in the case of *derivation by extension* (*Child* in the schema fragment above); or by restricting it, in the case of *derivation by restriction*. Type derivation in

XML Schema is similar to inheritance in the context of *Object-oriented programming* but limited only to data (fields).

What is interesting about type derivation is that wherever we find in the schemas an element of type A, the actual type of the element in an XML document can be either A or any type derived from it. This is why in Listing 2.2 an element of type *Base* has been substituted by an element of type *Child*, although it requires its real type to be specified using the *xsi:type* attribute.

Apart from type derivation, a second subtyping mechanism is provided through *substitution groups*. This feature allows global elements to be substituted by other elements in XML documents. A global element E, referred to as *head element*, can be substituted by any other global element that is defined to belong to the E's substitution group.

Both mechanisms provoke a polymorphic situation where the *real* or *dynamic type* of an element found in an XML instance may differ from its *declared type* in the schemas. In the remainder of this document we use the term *Data Polymorphism (DP)* to refer to the fact that nodes in XML documents may have a *dynamic type* that differs from its *declared type*. A side-effect of this situation is that dependencies between elements and types in the schemas are not always explicit, as an element of a certain type in an XML instance, may contain elements of types not mentioned explicitly in its declaration. We refer to these non-explicit dependencies as *hidden dependencies*. This is the case in the schema in Listing 2.3 of the relationship between *ContainerType* and *Child* (Figure 2.1). In an XML document an element of type *ContainerType* may contain elements of type *Child*, although no direct relationship between both types can be inferred by looking only at the definition of *ContainerType*.

2.2.2 Document Composition

Schema components defined in a file can be reutilised in other files through the use of *include* and *import* tags. Before explaining how they work, the concept of *namespace* must be introduced. XML namespaces is a mechanism that assigns expanded names to elements and attributes in order to avoid clashes between names from different markup vocabularies [W3C09].

Components defined in the same namespace, but in a different file, can be accessed by using the *include* tag that specifies in the *schemaLocation* attribute where the external schema is located. Similarly, components defined in a different namespace may be accessed by *importing*

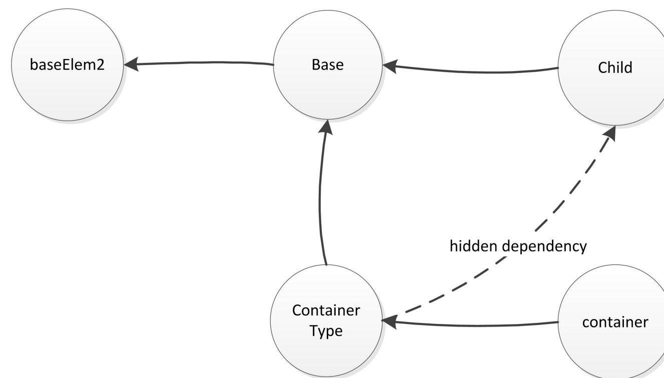

Figure 2.1: Graph of dependencies between schema components

the namespace and optionally specifying where the external schema is located.

2.2.3 Criticism

XML Schema has been widely adopted by the industry and academia, but has been frequently criticised for being an overly complex language. For example, [MNSB06] state that the complexity of the XML Schema specification and the difficulty of understanding the effect of constraints on typing and validation of schemas might be the cause why in practice the extra expressiveness of this language over its predecessor, DTD, is only used to a very limited extent.

[MS06] present a list of limitations of XML Schema arguing that one important factor of its complexity is the type system, as the notion of types adds a layer of complexity. Information items in XML instances must be matched to an element declaration, and then to a type definition with the constraints for its content, differing from DTD, where the information item can be matched directly to its associated constraints. These authors also consider the inclusion of both subtyping mechanisms introduced in Section 2.2.1 as the presence of conflicting design approaches in the *W3C XML Schema Working Group*, resulting in an unnecessarily complex specification. [Hos10] attributes the complexity of the schema language to the attempt of mixing two completely different notions: regular expressions and object orientation. [Kay03] qualifies the XML Schema specification as *"impenetrable to mere mortals"*.

2.3 XML Processing

The term *XML processing* includes the application programming interfaces (APIs) and techniques used to manipulate information in XML format. The main operations supported by these APIs are *parsing*[5] and *serialization.* Parsing is the action of reading the information contained in an XML document into the application. Serialization is the reverse process, i.e., saving application data into an XML file.

According to [WKNS05], XML processing can be implemented using a *vocabulary-independent data access interface* such as those provided by SAX [6][SAX], DOM[7][W3C04a] or StAX[8][Jav04]; or using a *vocabulary-dependent data access interface*, where XML data is mapped into application-specific concepts. The first option is recognised to be difficult and error-prone and produces code that is hard to modify and maintain. In words of Tim Bray, one of the co-creators of XML, he found the task of writing code to process arbitrary XML as *"irritating, time-consuming, and error-prone"* [Bra03].

The second option, also known as *XML Data Binding*, is favoured as developers can focus on the semantics of the data they are manipulating. An abstraction layer is added over the raw XML processing code, where XML information is mapped to data structures in a given application data model. XML data binding code is often produced by using code generators.

The importance of XML processing code and its impact in performance and size of applications have been traditionally underestimated. Analyses of the impact of XML processing code in the context of web servers and databases have been presented by [NJ03] and [AIM$^+$04]. [NJ03] present a set of database applications where XML processing is the performance bottleneck. [AIM$^+$04] show that in a commercial server under study with a real workload, 37 % of the time was spent in XML processing code.

[5] Also referred as deserialization, unmarshalling, etc.

[6] Simple API for XML

[7] Document Object Model

[8] Streaming API for XML

2.3.1 Vocabulary-independent Data Access Interfaces

Vocabulary-independent DAIs can be subdivided in *tree APIs*, like DOM, and *streaming (event-based) APIs*, like SAX. The main difference lies in that *tree APIs* create a hierarchical representation of the information, and the information cannot be accessed until the whole document has been parsed. *Streaming APIs*, however, do not need to parse the entire document before it can be processed, at the expense of not allowing random access to the document content. In the case of tree APIs, reading the whole document before accessing the information implies that much more memory must be used to accommodate this information than in the case of streaming APIs.

Streaming APIs can be further subdivided in *push-style* APIs (e.g. SAX), where the parser reads the documents and executes callback methods in the application to process the information; and *pull-style* APIs (e.g. XMLPull API [HS], StAX), where the application reads the document as a sequence of events and processes them as they are reached. In [LDL08], it is stated that the use of streaming APIs is preferred for networking applications, as information can be processed as it is received and memory is used much more efficiently.

2.3.2 XML Data Binding

A large number of code generators for XML data binding code exists for many platforms and target programming languages. These generators offer the possibility to generate source code to read and write XML documents based on a given set of XML schema files. Figure 2.2 shows the common process followed when a generator is used to produce XML processing code. A set of schema files is used as input to the generator that produces source code in a target programming language or binary code for a given platform. The generation process may be controlled by a configuration file defining different parameters to customise the transformation.

The transformation of schema components into programming language components can be defined in a pseudo-formal way as $T(x, p) = c$. In the formula, T is the transformation function, $x \in S$, where S is the set of all schema components in the input files; $p \in P$, where P is the set of all possible configuration parameters, and $c \in C$, where C is the set of data constructs in the target programming language. The Cartesian product of S and P, $(S * P)$, is the function domain and C is the function co-domain. Each code generator defines its own rules to accomplish

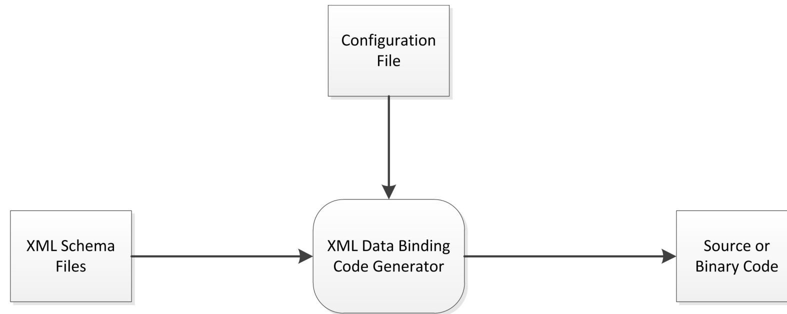

Figure 2.2: XML data binding code generation process

this transformation. Hence, in theory, we could uniquely describe each of them by defining how T is implemented.

Code generators provide an attractive approach, potentially giving benefits such as increased productivity, consistent quality throughout all the generated code, higher levels of abstraction as we usually work with an abstract model of the system; and the potential to support different programming languages, frameworks and platforms [Her03]. Nevertheless, current implementations present some limitations such as:

- Most generators make a straightforward mapping of components in the schemas to components into the target programming language. This is a problem when large schemas are used because the generated code can be very large to be executed on a resource-constrained device.

- In case code generators provide more advanced mapping capabilities between XML Schema and the target programming language, the parametrisation process can be complex. It will usually involve very low level manipulation of schema components requiring a deep understanding of their interrelationships.

The straightforward mapping of schema components to programming language constructs in the presence of large schemas is not a big problem in the context of desktop or web server applications, but it is critical for a mobile device. For example, the code generated from the SOS schemas using XBinder, presented next, and targeted to the Android mobile platform has a size of almost 4 MB, which is rather large to be included in

a mobile application[9]. The parametrisation problem is not as critical as the former problem, but it can be a cause of delay during development when schemas with a large size and containing complex relationships are used.

In the remainder of this dissertation we will use the following XML data binding tools to demonstrate the large size of the code generated from geospatial schemas, and as reference points against which our data binding implementation will be compared:

- *JAXB-RI*[JAX]: Reference implementation of the *Java Architecture for XML Binding*(JAXB) API [Jav06]. This API allows Java developers to map information in XML instances to Java objects inside their applications. JAXB-RI maps types in schema files to types of the Java language in a straightforward way, making every complex type defined in the schemas a class declaration in the programming language.

- *XMLBeans* [Apa]: XMLBeans is a widely used code generator for the Java language. It offers similar features than JAXB but performs a more sophisticated mapping of schema components to Java classes and interfaces that result in a larger and more complex code.

- *XBinder*[Obj]: XBinder produces code for several programming languages (C, C++, Java, C#). It also allows the generation of code targeted to different mobile platforms such as Android and CLDC[10][Oraa]. The structure of the generated source code resembles the structure of the XML Schema definitions.

The availability of a large number of XML data binding code generators has caused that the task of producing XML processing code has been taken for granted by schema designers, which frequently assume that independently of the length of schemas, a working implementation can be built with little effort. Although this is true in some occasions, with the growth in size of schemas in some domains it might not be the case. For example, we face this problem when code generated from large

[9] An experiment measuring the size of generated binary code for SOS schemas is detailed in Chapter 6

[10] The *Connected Limited Device Configuration* (CLDC) defines the base set of application programming interfaces and a virtual machine for resource-constrained devices using the Java language

schemas must be accommodated in a device with memory or processing limitations such as mobile devices.

2.4 Web Services

A comprehensive definition of Web services is provided by [Pap08]. This book states that *"A Web service is a platform-independent, loosely-coupled, self-contained, programmable Web-enabled application that can be described, published, discovered, coordinated, and configured using XML artifacts (open standards) for the purpose of developing distributed interoperable applications"*, besides, *"Web services expose their feature programmatically over the Internet (or Intranet) using standard Internet languages (based on XML) and standard protocols, and can be implemented via a self-describing interface based on open Internet standards."*. From the previous definition we can infer that there is a tight relation between Web services and XML, hence, all of the issues related with XML processing has a direct impact on web service development.

Summarizing, the more relevant points about Web services are:

- *Web services are loosely coupled software modules*: A web service exposes its functionality through an interface. This interface is defined in a platform-independent way, which allows the interaction of services built on different operating systems and programming languages.
- *Web services semantically encapsulate discrete functionality*: They are software modules that perform well-defined tasks.
- *Web services can be accessed programmatically*: They are not targeted to a human user, but operate to the code level.
- *Web services are distributed over the Internet*: They make use of widely-adopted internet protocols such as HTTP [IET99].

According to [Kra07] Web services offer several technical advantages and organizational benefits:

- *Maintenance of legacy systems*: Services can be added on top of existing systems without affecting internal processes.
- *Independence of programming languages*: Web service end-points may be developed in different programming languages (and deployed in different platforms), being still able to interoperate.

- *Reduction of data management issues*: Clients only access the data they need at a specific point of time. They are not deeply concerned with bandwidth, data and storage management issues.
- *Rapid application development and integration*: As the number of services provided worldwide through web service interfaces is continuously growing, new sophisticated applications can be created. This results in applications with no explicit data embedding and a multitude of data sources for integration.

Web services are the more common technology used to implement a *Service Oriented Architecture* (SOA), defined in [Bea09] as ''*...a combination of consumers and services that collaborate, is supported by a managed set of capabilities, is guided by principles, and is governed by supporting standards*".

2.4.1 Approaches to Web Service Development

Currently, there are two main competing architectures for web service development: *RPC-style* (or *"Big" web services*) and *RESTful resource-oriented* [RR07, PZL08]. In addition, [RR07] identify a third one, which is a combination of both: *REST-RPC hybrid*.

Remote Procedure Calls (RPC) in computer science refers to the execution of a procedure located in a remote computer across a network. The term RPC applied to Web services consists of exchanging information in the form of self-contained documents (messages) over the HTTP protocol. Messages are encoded usually, but not necessarily using SOAP, an XML language defining a message architecture and formats [W3C07]. SOAP simply acts as an *envelope* for the content or *payload* that is annotated with some information included in *headers*. Web services end-points are commonly described using the *Web Services Description Language* (WSDL), an XML-based language designed for this purpose [W3C01]. In our opinion, the main characteristic identifying the RPC-style is that functionality is accessed using the notion of *procedure*. A procedure is just a processing entity that receives some input and produces some output. The RPC-style is the most widespread because a more mature stack of technologies, with extensive tool support, has been constructed on top of this paradigm. Functionality related to transaction support, reliability or message-level security can be integrated into SOAP headers. The main disadvantage of this style is its complexity, although much of it can be overcome with the use of appropriate tools [PZL08].

The RESTful resource-oriented style was originally defined by Roy Fielding for building large-scale distributed hypermedia systems [Fie00]. The style contains a set of constraints applied to elements within the architecture, such as *client-server interaction*, *statelessness*, *cacheability*, or *use of an uniform interface*. The term *RESTful web service*, not included in the original Fielding's work, is the application of these constraints to the Web service realm. Here, services are built as collections of resources (anything that can be named) that are accessed through the uniform interface provided by HTTP. This way, services can be built with no more infrastructure that the one provided by the Web itself, and consequently they are perceived to be very simple. A more detailed description of this architectural style can be found in [Fie00].

2.5 Concluding Remarks

In this chapter we have introduced the main concepts and technologies related to XML, XML Schema and Web services. Understanding these concepts is a requirement to understand the context in which our work has been developed. Web services are the abstraction used to define standard web service interfaces to exchange geospatial information. The structure of the messages exchanged between client and servers, in XML format, is described using the XML Schema language. With the purpose of building a working implementation based on the standards, XML processing code must be produced either by using a manual or automatic approach. The task of producing code manually is tedious and error-prone, but the alternative, using an XML data binding code generators do not always produce code satisfying mobile application requirements, as we will see in the following chapters.

CHAPTER 3

OGC Web Services

The *Open Geospatial Consortium* (OGC) is a non-profit, international organisation that is leading the development of standards for geospatial and location based services [OGCa]. This organisation has produced and promoted open standards to enable *interoperability* between geospatial information producers and consumers. The aim is to build the *Geospatial Web*, defined in [LF07] as *"...an integrated, discoverable collection of geographically related Web services and data that spans multiple jurisdictions and geographic regions. In a broad sense, the Geospatial Web refers to the global collection of general services and data that support the use of of geographic data in a range of domain applications"*.

The use of these standards and the participation in its definition offers several benefits to content providers and consumers. For example, content providers position themselves early to influence definition of new standards, reduce costs through cooperative standard development with other OGC members, shorten time to market by using OGC standards rather than custom interfaces, and have a convenient forum for discussion of industry issues and solve shared problems. They can also form customer relationships and business partnerships, deliver solutions more quickly and at lower costs; they can mobilize a range of products across open interfaces, rather than performing resource intensive custom integration [Per10].

On the other side, technology and information consumers can voice their interoperability needs directly to a broad global industry, academic and government community; they can be assured that reusability of software is achieved, or work with other users in the OGC process to demonstrate the need for and potential market appeal of new requirements.

They can consider OGC programs as a form of technology risk reduction, mobilize new technologies solutions quickly, and adapt easily to the rapidly changing information technology world, policy changes, and new emerging requirements [Per10].

3.1 OWS Architectural Principles

OGC defines a set of web service interface specifications and data encodings to exchange geospatial data. These services are known as a whole as *OGC Web Services* (OWS) and they are built following the fundamental principles summarised next [Per10]:

- *Components implementing services are organised into multiple tiers*: These services are available to clients or other software components. Services are loosely arranged in four tiers from Clients to Application Services to Processing Services to Information Management Services. Services can use other services within the same tier. Servers can operate on data stored in that server and/or on data retrieved from another server.

- *Collaboration of services produces user-specific results*: All services are self-describing, supporting dynamic connection binding of services supporting *publish-find-bind*[1]. Services can be chained with other services. Services are provided to facilitate defining and executing chains of services.

- *Service communication uses open Internet standards*: Communication between components uses standards World Wide Web protocols such as HTTP GET, HTTP POST, and SOAP. Specific server operations are addressed using *Uniform Resource Locators* (URLs). *Multi-purpose Internet Mail Extensions*(MIME) types are used to identify data transfer formats. Data transferred is often encoded using XML, with the contents and format specified using XML schemas.

- *Service interfaces use open standards and are relatively simple*: OGC web services interfaces are coarse-grained, providing only a few

[1] Deployment pattern where services are *published* to a register, where service consumers *find* (discover) service instances and then *bind* (invoke) to these services.

static operations per service. Service operations are stateless, not requiring servers to retain information about past interactions. One server can implement multiple service interfaces whenever useful. Standard XML-based data encoding languages are specified for use in data transfers.

- *Server and client implementations are not constrained*: Services are implemented by software executing on general purpose computers connected to the Internet. The architecture is hardware and software vendor neutral. The same cooperating services can be implemented by servers that are owned and operated by independent organisations. Many services are implemented by standard-based Commercial Off-The Shelf (COTS) software.

In the next section we introduce the main OWS specifications. Further information about the OWS topic can be found in [Kra07, LF07] and [Per10] or the OGC website[2].

3.2 OWS Specifications Overview

Some of the most widely known specifications used in this dissertation are shown in Table 3.1. As mentioned before, OGC specifications include web services interfaces, as well as data encodings used to exchange data between OWS servers and clients. Content providers publish their data through a well-defined interface that clients may access through standard Internet protocols such as HTTP. The preferred method to encode web service operations in OWS has been so far the use of *HTTP bindings*. These bindings use HTTP GET with *Key-Value Pair* (KVP) encodings and/or HTTP POST with an attached XML document containing the request parameters. Newer specifications also include SOAP bindings [OGC08d].

An important requirement in the design of OGC specifications design is reusability. They are built on the foundation provided by other specifications. The reutilisation of existing components simplifies the specification design task, but it also brings the complexity of the reutilised specifications into the other specifications as well. Figure 3.1 shows the

[2] `http://www.opengeospatial.org/`

Table 3.1: Geospatial web service interfaces and data encodings

Name	Description
Web Map Service (WMS)	WMS provides a simple HTTP interface for requesting geo-registered map images from one or more distributed geospatial databases [OGC06c]
Web Feature Service (WFS)	It allows a client to retrieve and update geospatial data encoded in GML format [OGC10c]
Web Coverage Service (WCS)	It provides access to rich sets of spatial information, in forms useful for client-side rendering, multi-valued coverages, and input into scientific models [OGC10a]
Sensor Observation Service (SOS)	It provides an API retrieving sensor and observation data [OGC07h]
Web Processing Service (WPS)	It defines a standardised interface to publish geospatial processes [OGC07g]
Sensor Planning Service (SPS)	It defines interfaces for queries that provide information about the capabilities of a sensor and how to task the sensor[OGC07d]
Geography Markup Language (GML)	GML is a grammar for expressing geographical features. It serves as a modelling language systems as well as an interchange format [OGC04]
Sensor Model Language (SensorML)	SensorML is a language that specifies models and encodings that provide a framework within which characteristics of sensors and sensor systems can be defined [OGC07c]
Observation and Measurements (O&M)	O&M defines an abstract model and Sensor Model Language schema encoding for observations [OGC07a]
KML	KML is a language focused on geographic visualisation, including annotation of maps and images [OGC08a]

dependencies that exists between some of the OGC specifications introduced in Table 3.1[3]. Boxes in darker colours represent service specifications and the others represent mostly data encoding specifications. These dependencies have been extracted from the schemas included with the specifications themselves. For example, *WCS 2.0* schemas depend for its definition on *Web Services Common Standard* (OWS 2.0 in the figure) [OGC10b], GML 3.2.1 and the *SWE Common Data Model Encoding Standard* (*sweCommon 2.0*) [OGC11]. *OWS 2.0* defines a set of aspects that are common to all of the OGC web services implementations, and *sweCommon 2.0* defines low-level data models for exchanging sensor related data. In the figure we can observe that not all the specifications have the same number of dependencies, which suggests that the level of complexity varies between specifications. The dependencies do not always have the same nature either, for example, there are circular dependencies

[3]Table 3.1 only includes the specifications that are used later on this document, as a consequence some of the specifications in the figure are not listed.

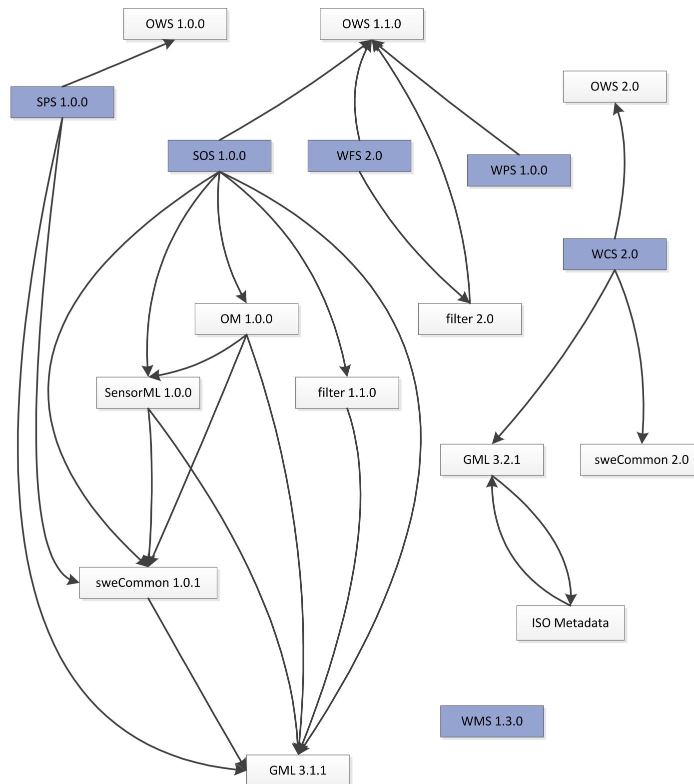

Figure 3.1: Dependencies between OGC specifications

and also specifications that depends on others through different paths on the graph.

In the remainder of this document when referring to schemas in a specification we differentiate between *main schemas*, those defined completely in the specification, and *external schemas*, those included or imported from the main specification schemas.

3.3 OWS Implementations

The number of OWS implementations keeps growing every year. The number of registered products in the OGC website implementing the specifications are counted by hundreds [OGCb]. Regarding service instances available online the study presented in [LPBHF^{+}10] found 6,544 service

instances in Europe of which more than 50% were WMS servers. In this section, we show some existing products in the server and client-side. We also highlight the fact that few implementations exist for mobile platforms.

3.3.1 Server Side

In the server side a group of well-known products exists. Some of them implements several OWS specifications such as GeoServer [geo], MapServer [map] or Deegree [dee]. Other family of server products is produced by 52° *North - Initiative for Geospatial Open Source Software GmbH*[4].

GeoServer is an open source software server that implements several OGC web service specifications such as WMS, WFS and WCS. This product is the OGC's reference implementation for these specifications. *Deegree* is a geospatial software package that provides server-side implementations of several OGC web services such as WMS, WFS, WCS, *Catalogue Service Web-Profile*(CSW) and in more recent versions WPS and SOS. Similarly, MapServer, provides implementations for WMS, WFS, WCS, SOS, etc., and also includes modules for web client-side applications.

The 52° North product family includes implementations for WPS, and sensor-related specifications such as SOS, *Sensor Alert Service* (SAS) [OGC06b], *Sensor Event Service* (SES)[OGC08c], SPS and *Web Notification Service* (WNS)[OGC07f]. SAS enables real-time alerting for sensor related information. SES is an enhanced version of SAS, and WNS provides support for asynchronous notification of sensor events. None of these last specifications has reached the status of *standard*.

3.3.2 Client Side

For the client side a large number of implementations has also been developed. We can roughly classify these products as: *generic clients*, *centralised portals*, and *middleware or supporting libraries*. The classification is not meant to be exhaustive and different products may be included in more than one category at the same time.

In the category of *generic clients* we include products that allow users to select the server they want to connect to, and they may request information from this server. Examples of these clients are uDig [Ref],

[4] `http://www.52north.org/`

gvSIG [gvsa] and OPENJump [ope]. As a general rule, these clients allow users to connect explicitly to servers providing data through different standards.

The second category, *centralised portals*, refers to web portals where users can visualise information that is provided by one or several organisations. In this case, the users usually do not select the servers they want to connect to, but can select from a list of available data which part of it they want to visualise and/or process.

One example in this category is *CartoCiudad*[5], a web portal containing information about Spain, regarding territorial division, urban background, address, transportation network, etc. It allows the visualisation of this information as well as it provides functions to search for places or to calculate routes (Figure 3.2). Another example in this category is the *Integrated Ocean Observing System* (IOOS)[6]. According to its website IOOS is a federal, regional, and private-sector partnership working to enhance the ability to collect, deliver, and use ocean information. The website provides a viewer that allows users to visualise information gathered by sensors and served through SOS instances (Figure 3.3).

The last category is that of *middleware or supporting libraries* which allows developers to access OGC web services through an API. This is the case of Geotools[7], a Java open source library to manage geospatial data. The library provides support for different OGC web services such as WMS and WFS. In addition it supports different vector and raster data formats such as GML, ESRI Shapefiles [ESR98], GeoTIFF [NR00], etc. It also allows developers to interact with spatial databases in PostGIS[8], Oracle Spatial[9] or MySQL[10].

[5] `http://www.cartociudad.es`

[6] `http://www.ioos.gov`

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

[8] `http://postgis.refractions.net/`

[9] `http://www.oracle.com/technetwork/database/options/spatial/index.html`

[10] `http://www.mysql.com`

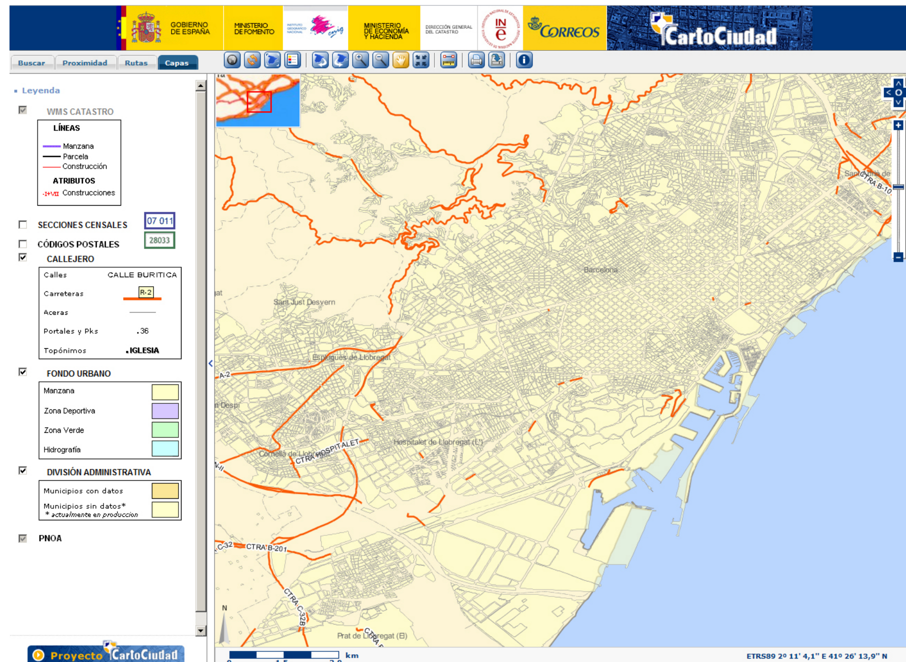

Figure 3.2: CartoCiudad web portal

3.4 Sensor Observation Services

The SOS specification will be used frequently in the remainder of this document. The main reasons for this are first, the large experience we have with the specification [Tam09, THG+09, TGD+11, TVGH11, TGH11b]. Second, the specification is one of the more complex in nature from all of the OWS specifications, as it provides larger functionality and depends on a larger number of other specifications (see Figure 3.1). As it will be the main focus of our experiments we consider pertinent to provide a wider introduction to SOS.

SOS has been developed in the context of the *Sensor Web Enablement* (SWE) initiative, a framework that specifies interfaces and metadata encodings to enable real-time integration of heterogeneous sensor networks. It provides services and encodings to enable the creation of web-accessible sensor assets [OGC08b]. SWE is an attempt to define the foundations for the *Sensor Web* vision, a worldwide system where sensor networks of any kind can be connected [LCT05, vZSM08]. It includes specifications

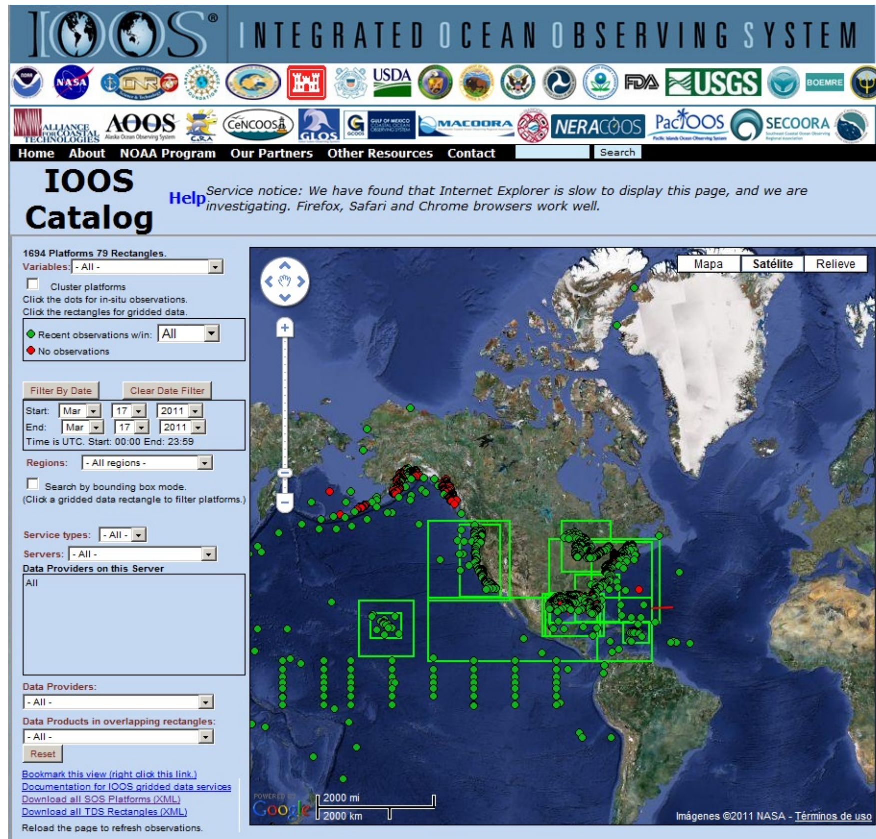

Figure 3.3: Integrated ocean observing system web portal

for service interfaces such as: *Sensor Observation Service* (SOS); and *Sensor Planning Service* (SPS); and encodings such as: *Observation and Measurement* (O&M) and the *Sensor Model Language* (SensorML). All of these web service interfaces and encodings have been described in Table 3.1. SWE also comprises other OGC discussion papers such as SAS, SES and WNS. A comprehensive review of SWE can be found in [BEJ$^+$11].

The SOS implementation specification provides access to observations gathered by sensors in a standard way that is consistent for all sensor systems, including remote, in-situ, fixed and mobile sensors [OGC07h]. The information exchanged between SOS clients and servers, as a general rule, will follow the *O&M* specification for observations and the *SensorML* specification for sensors or system of sensors descriptions. Besides, the

Table 3.2: SOS operation profiles

Profile	Operations
Core (Mandatory)	*GetCapabilities, DescribeSensor, GetObservation*
Transactional	*RegisterSensor, InsertObservation*
Enhanced	*GetResult, GetFeatureOfInterest, GetFeature-OfInterestTime, DescribeFeatureOfInterest, De-scribeObservationType, DescribeResultModel*

specification is open to extension by using observation types defined by information providers. The main goal of SOS is to provide access to observations, which are grouped into *observation offerings*. An observation offering is a set of observations related by some criteria. Unfortunately, the specification does not provide a more precise definition or any clear guidance about how to do this grouping. The only clue provided is that classifiers (sensor systems, observed property, location, etc.) must be factored into offerings in such a way that in response to a *GetObservation* request the likelihood of getting an empty response for a valid query should be minimised. The offerings are constrained by the following parameters [OGC07h]:

- Specific sensor systems that report the observations,
- Time period(s) for which observations may be requested (supports historical data),
- Phenomena that are being sensed,
- Geographical region that contains the sensors, and
- Geographical region that contains the features that are the subject of the sensor observations (may differ from the sensor region for remote sensors)

The SOS implementation specification defines three operation profiles: *core profile* (mandatory), *transactional profile* (optional) and *enhanced profile* (optional). These profiles are shown in Table 3.2 and explained next.

The *core profile* contains three operations: *GetCapabilities*, *DescribeSensor*, and *GetObservation*. These are the basic operations needed

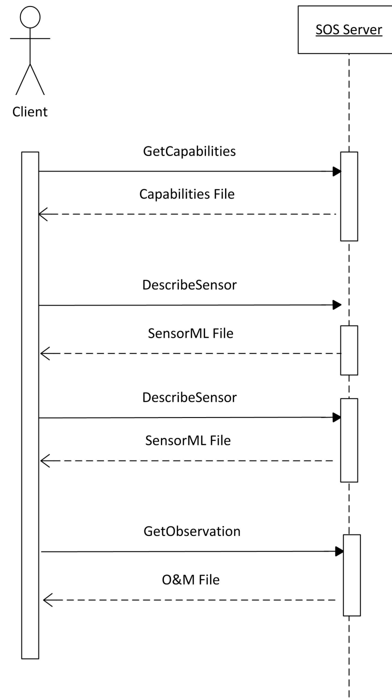

Figure 3.4: Interaction with the SOS server

for any data consumer to access sensor observations stored in an SOS server. *GetCapabilities* is an operation that is common for all the OGC web services, and as such is defined in [OGC07b]. This operation allows clients to access metadata about the capabilities provided by the server. The *DescribeSensor* operation allows SOS clients to retrieve SensorML or *Transducer Markup Language* (TML) [OGC07e] descriptions of a given sensor specified as parameter of the operation. The *GetObservation* operation is used to retrieve observation data from the server. Several parameters for filtering the observations must be supplied. The common flow of interactions of a client with a SOS server is shown in Figure 3.4.

The *transactional profile* offers support for data producers. Using the supplied operations *RegisterSensor* and *InsertObservation*, a data producer can register its sensor systems and insert the observations produced

by them into the server. Later, clients can read this information using the core profile operations.

The third and last profile is the *enhanced profile*, which provides clients with a richer interface for interacting with the server. The operations are *GetResult*, that allows clients to obtain sensor data repeatedly without having to send and receive requests and responses that largely contain the same data except for a new timestamp; *GetFeatureOfInterest* returns the description of a feature advertised in some observation offerings of the SOS capabilities document; *GetFeatureOfInterestTime* returns the time periods for which the SOS will return data for a given advertised feature of interest; *DescribeFeatureOfInterest* returns the XML schema for a given feature; *DescribeObservationType* returns the XML schema that describes the Observation type that is returned for a particular phenomenon; and *DescribeResultModel* returns the schema for the result element that will be returned when the client asks for the given result model by the given *ResultName*.

3.5 Concluding Remarks

Standard specifications to exchange geospatial data are defined using web service interfaces. The use of these standards and the participation in its definition offers important benefits to content providers and consumers. The specifications includes a set of operations which rely in a set of encodings describing the structure of the exchanged data. The encodings and the structure of operation requests are defined using XML Schema, as the messages are encoded using XML. An important number of applications implementing OWS specifications exists for the server and client side, which is a proof of their ample acceptance in academic circles and industry.

CHAPTER 4

Mobile Computing

In the last decade, we have witnessed an outstanding growth in the number of mobile devices such as hand-held PCs, PDAs and smartphones. These devices frequently equipped with wireless networking capabilities are changing the way people interact with information in the Internet, shaping what has been called the *mobile Web* [CCL09]. In the mobile Web users access Web content anytime, anywhere, and through any class of device. This revolution has been possible by the rapid evolution of mobile hardware and software, and the widespread use of wireless communications. In this chapter we present a brief introduction to the realm of mobile computing, focusing mostly on *mobile phones*, as they are the mobile devices our work is targeted to.

4.1 Mobile Hardware

In the hardware aspect, *Advanced RISC Machine* (ARM) has been the predominant architecture for mobile processors. Currently, it is easy to find 1 GHz processors in smartphones, such as the *Qualcomm Snapdragon* family of processors [Qua] or the *Apple A4* processor [Mac10]. These are *System-on-a-chip* processors including at least graphic processing capabilities in the same unit.

Regarding networking, mobile connectivity is based on wireless technologies, which has evolved very quickly with networks such as GPRS [3GP], the UMTS/3GSM family [1] with data rates of 384 Kbit/s to 14.4

[1] http://www.umtsworld.com/technology/technology.htm

Mbit/s, or more recently LTE[2] offering download speed above 100 Mbit/s. It has also become common that mobile phones be equipped with *Wireless LAN* (WLAN) transmission [IEE07]. These options allow mobile users to choose depending on their location and the available services which technology better suits their needs for a certain application.

Storage capacity has also grown in the last years, and today it is possible to incorporate *Secure Digital* (SD) and *SD High-Capacity* (SDHC) cards to mobile phones. SD cards have a capacity of up to 4 GB and SDHC cards of up to 32 GB, although its support on different mobile platforms varies. The transfer speed of these cards ranges between 2 MB/s to 10 MB/s. SDHC can also support the Ultra High Speed mode I (UHS-I) with a trasfer rate of up to 104 MB/s. A more recent technology in the market is SDXC (*SD eXtended-Capacity*), to our best knowledge still not available for mobile phones. It offers capacities of more than 32 GB up to 2 TB[3].

Despite all these advances, a key point regarding mobile hardware is that they are *battery-powered*, so even when applications with more demand for processing capabilities, data bandwith, etc. may be executed on mobile devices, saving energy by reducing both processing and data transmission is still of paramount importance [KLT07, WTS07, vB09]. As such, the use of advanced features such WLAN networking or GPS support must be balanced with the large impact they have in battery life. In [PS05], it is shown that as disk capacity, CPU speed, RAM and wireless transfer speed have been increasing exponentially since 1990, battery energy density has had a much more modest growth.

4.2 Mobile Software

In the software aspect, highly influenced by hardware advances, a group of strong software platforms have been developed such as Symbian OS [Nok], Android [Goo], Blackberry OS [RIM], iOS [App], and Windows Phone [Mic]. These platforms have also a large number of applications that can be bought or downloaded for free from different software market stores such as *Apple App Store*, *Android Market*, *Ovi Store* or *Windows Marketplace for Mobile*.

Symbian OS is the leader of the operating systems for the smartphones

[2] 3GPP Long Term Evolution: `http://www.3gpp.org/article/lte`

[3] `http://www.sdcard.org`

market with around 37% of the market share [Gar11]. The design of Symbian OS is highly modular, and most of its components expose a C++ API that is available to third-party application developers [Hay09]. Symbiam applications can also be developed using the Java Platform, Micro Edition (Java ME)[Orab], a Java platform designed for embedded systems (mobile phones, set-top boxes, digital TVs, vending machines, etc.).

4.2.1 Android

Android, developed by Google, is currently the platform with the fastest growth rate in the smartphones market, having a market share of 22.7% as of February 2011 [Gar11].

Android is a Linux-based operating system targeted to smartphones. It provides a Java API similar to that offered by Java ME. One difference of the Android API with Java ME is that Android is not targeted to multiple device configurations, which makes the Android API simpler as it is targeted to a single device model. Another difference is that Java Virtual Machine included in Android (called Dalvik) is more optimised and responsive than other JVM [HKM+10]. The Dalvik VM is specially optimised for slow CPU and little RAM. It is has also been highly optimized to keep power consumption as low as possible.

Dalvik uses it own byte-code format called *Dalvik Executable format* (DEX), which is more compact than compressed .jar files [PK10]. Last versions of Dalvik have added *Just-In-Time (JIT) compilation* to speed up the execution of Java code. Android also implements the SAX, DOM and XMLPull APIs for XML processing.

We have chosen Android as our development platform because, apart from its growing popularity, the Android SDK includes an useful set of development tools available for free. These tools allows its integration with the *Eclipse Integrated Development Environment*[4]. The Eclipse plug-in, named *Android Development Tools*(ADT) includes, in addition to the possibility to develop and execute applications directly in an Android device, a platform emulator where applications can be developed, debugged and tested.

[4] `http://www.eclipse.org/`

4.3 Web Services for Mobile Devices

The progress of mobile hardware and software in the last few year has made possible to access services available in the Web from mobile terminals. The adoption of Web services in mobile devices is a hot topic in research [TVN+03, FC05, SNMRRP07, KL09, CCL09, ZDL09]. [ZDL09] presents a convenient categorisation of the work in the subject:

- *Adaptation of standard web service technology*: Standard technologies such as WSDL and SOAP are applied directly to mobile systems. Smaller and more restricted devices might omit some advanced capabilities such as dynamic discovery and transactions.

- *Alternative protocols*: The use of low bandwidth consuming protocols enhances communication between clients and providers. Examples are the use of compression techniques or alternative exchange formats to deal with the overhead of XML in service descriptions and formats.

- *Use of mediator components*: Proxy components can be placed between mobile devices and service providers. These proxy components can adapt the service responses to the limitations of mobile devices.

Web service clients in the first category, may interact with existing services available online without these being modified. In the second category, services must support alternative protocols to deliver compressed data. The last category is a trade-off, where services remain unmodified by the addition of an extra component that deals with needed data transformation.

Another interesting category of Web services is that of *Mobile-aware web services*. These services have the capability of adapting their responses to the need of mobile clients, such as screen resolution limitations, bandwidth limitations, etc. The adaptation of the content to the user device provides an improved user experience and avoids occupying network bandwidth with information that will not be shown to the final user [CCL09, OdP09]. These services might belong to the first category above, if still using standard web service formats; or to the second category if using alternative, specialised formats.

4.4 XML Processing for Mobile Devices

When considering XML processing in the context of mobile devices there are two main competing requirements: *compactness* and *processing efficiency* [KLT07]. The smaller the messages transmitted, the less resources are spent in data transmission, but this may require the use of more processing power if the data must be compressed and decompressed. Processing efficiency refers both to time and memory spent during processing.

To reduce message size several compression techniques or alternative protocols have been used. For example, the use of more compact XML-based formats such as WAP Binary XML [W3C99a], EXI [W3C11] or Xebu [KTL05]; or even the use of general purpose compression techniques such as gzip [Deu96]. Some of these techniques, such as EXI[5] and Xebu, reduce the size of data but do not require this data to be decompressed by the message receiver, while others must spend processing time performing compression/decompression tasks (gzip). The choice of using compression or not must be carefully considered because it has been proven that wireless communication can be much more expensive than computation in terms of energy consumption [BA03]. This may cause that in some scenarios transmitting uncompressed large datasets may be more expensive that the option of compressing-transmitting-decompressing the same dataset. In all these cases there is a drawback, which is that the use of these *alternative protocols* must be supported on the server side, or at least on a mediator component handling the communication with the real server.

About processing efficiency, not much work has been done in the mobile devices area. XML processing code has been produced following best practices such as the use of streaming APIs to avoid unnecessary memory consumption when large documents are handled. A prominent exception in this topic is the work presented in [KLT07], [KTL05] and [LK08]. These articles are all related to the implementation of a middleware platform for mobile devices: the *Fuego mobility middleware* [TKLR06], where XML processing has a large impact. The proposed *XML stack* provides a general-purpose XML processing API called *XAS* [KLT07], an XML binary format called *Xebu* [KTL05], already mentioned before, and others APIs such as *Trees-with-references* (RefTrees) and *Random Access XML Store* (RAXS)[LK08]. XAS is implemented on top of kXML [Hau], an

[5]EXI can also be combined with compression, but we are considering here the case where this feature is not used.

XMLPull parser implementation for mobile devices, adding support for *typed content*. Xebu is a binary XML format based on the event model provided by XAS. Last, *RefTrees* and *RAXS*, built also on top of XAS, allow manipulating XML documents using a tree model, and provides document management operations, such as change transactions, versioning, and synchronisation.

Regarding XML data binding there are also several tools available for generating XML data binding code for mobile devices such as XBinder and CodeSysnthesis XSD/e [Cod], or for building complete web service communication end-points for resource constrained environments, such as gSOAP [VEG02]. All of these tools map XML Schema structures to programming languages construct in a straightforward way, which is not adequate when large schema sets are used.

4.5 OWS Implementations for Mobile Devices

The integration of OGC web services in mobile devices has been slow. The main reason for this is that geospatial data usually occupies lots of space and requires substantial processing capabilities, which are not always available in mobile platforms. Most of the communication and data exchange in OWS is encoded in XML format. The performance problems associated with XML processing in mobiles is well-known, as the effort to parse and serialise XML messages from files (or communication channels) to memory and vice versa, consumes a lot of resources. [KLT07, DKDF09, ZXjC+10]

Unfortunately, the approaches to XML processing presented in the previous section are not easily or effectively applied to mobile OWS-based applications. For example, the use of compression techniques would require to spend processing time performing compression operations and to modify the existing infrastructure of OWS-based servers to support the compressed formats. On the other hand, XML data binding code generators tend to map types in schema files to types in the target language in a straightforward way, which cause that large schema files produces large binary compiled files. Last, SOAP-based generators cannot applied for some specifications because they do not provide support for SOAP bindings.

We have been able to find only a few actual OWS clients for mobile devices, most of them for the WMS specification. The first one is J2ME

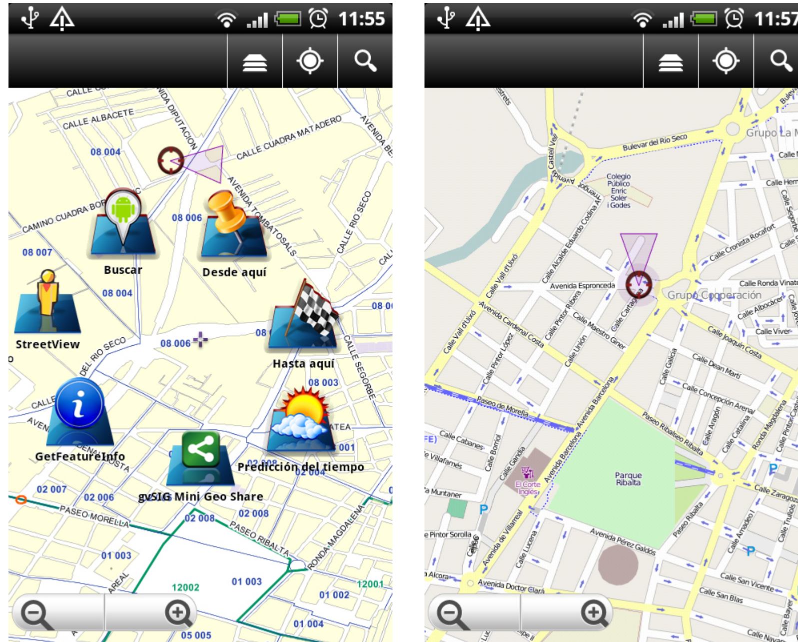

Figure 4.1: gvSIG Mini screenshots. To the left some of the options available are shown over a WMS layer. To the right the current location is shown using Open Street Map as base map layer

OGC WMS Client by Skylab[6], which is a Java ME client able to get and display maps from a WMS server. The second and third one are the two versions of gvSIG for mobile devices, gvSIG Mobile [gvsb] and gvSIG Mini [Pro].

gvSIG Mobile is a GIS that allows to access the most common spatial formats and a wide range of GIS and GPS tools. It allows to connect to WMS servers. *gvSIG Mini* is a free viewer of publicly available maps based on tiles, which also acts as a WMS client. It also allows address and *Point Of Interest* (POI) searches, route calculations, etc. Figure 4.1 shows some screenshots of this application. It runs on Android and Java ME CLDC compatible platforms.

Last, in [DKDF09], a WMS client implementation for mobile devices

[6] http://www.skylab-mobilesystems.com/en/products/j2me_wms_client.html

is presented. The WMS client is divided in two parts: the *Mobile Client* (MC) and the *WMS Connectivity Layer* (WCL). WCL, running in a desktop computer, acts as a proxy for MC, providing caching and tiling functionality to reduce data transfer with the mobile client.

4.6 Concluding Remarks

With this chapter we close the presentation of the context in which our work has been developed. As mobile devices include today tasks that a few years ago were only feasible for server and desktop computers, the adoption of web service technologies was just a matter of time. Mobile web service clients interact with existing web services that may or may not be aware of the nature of the devices they are interacting with. Knowing that they are interacting with a mobile device allows them to adapt their content to existing device limitations, although this requires the existing infrastructure of services to be modified. Despite the rapid evolution of mobile phones, battery life is still a serious limitation, as such processing power and data transmission capabilities must be used judiciously. For this reason, the role of XML processing in the mobile scenario is very important and it must avoid consuming scarce resources unnecessarily.

The verbosity of XML, and the large size and complexity of OWS specifications and schemas provoke that existing technologies such as XML data binding code generators, could not easily produce code meeting mobile application requirements. This fact may be one the causes of the low number of OWS applications available for mobile devices.

Part III

XML Processing for Geospatial Mobile Applications

CHAPTER 5

Complexity of OWS Schemas

The number and size of OGC standards have been growing in the last few years. The complexity of the standards is well-known, but rarely mentioned in research literature. Besides, little effort has been made to measure it properly using well-defined software metrics. Understanding how and why the specifications have grown and finding solutions to deal with this complexity can be hardly accomplished without the use of appropriate metrics to control and assess the evolution of the specifications.

Complexity in geospatial web services comes mostly from the complexity of the geospatial domain. As stated in [GYC07], *"geographic representation has become more complex through time as researchers have added new concepts, leading to apparently endless proliferation and creating a need for simplification"*. The authors also question if such complexity is really necessary. Although we cannot deny the inherent complexity of the

A short version of the content of this chapter has been published with the title *"Analysing complexity of XML schemas in geospatial web services"* in Proceedings of the *2nd International Conference on Computing for Geospatial Research & Applications (COM.Geo 2011)*, May 23-25, Washington, DC 2011. DOI: 10.1145/1999320.1999337

A modified version with the title *"Measuring Complexity in OGC Web Services XML Schemas: Pragmatic Use and Solutions"* has been accepted to the *International Journal of Geographical Information Science*, Taylor & Francis.

problem domain, during the modelling process of geospatial standards, some extra complexity is introduced due to design decisions. Besides, as the standards are implementation specifications, the path is not complete until concrete implementations are built, hence, other implementation-related aspects, such as poor tool support, may increase the perceived complexity of the specifications.

In our opinion, the complexity of the schemas associated to OWS specifications reflects the complexity of the whole web service specifications. This is because, on the one hand, these web service interfaces do not contain a large number of operations requiring users to have a complex flow of interactions with the servers. This coincides with the results presented by [YLRY07], where a comparison of web services with object-oriented software components was performed. The reasons presented there to explain this observation were that web service interfaces prefer to have less operations with larger parameters, due to the stateless nature of web services and to minimise the number of messages exchanged on the network.

On the other hand, the amount of increasingly structured data exchanged between clients and servers has reached levels that represent a serious challenge when building a real system. The influence of data complexity in web services interfaces has been highlighted in [BM09a], where a metric to calculate complexity of web services based on exchanged data was developed. The results were contrasted with other metrics that do not take data complexity into account, showing that their metric provides a more accurate measure for complexity.

For all of the reasons mentioned above, we present in this chapter an attempt to measure the complexity of XML schemas included in the OWS specifications. The complexity is measured using a set of metrics extracted from the literature, as well as a set of metrics defined by ourselves. Our metrics are targeted to measure the influence that the use of the subtyping mechanism of XML Schema has in complexity. The typing mechanism of XML Schema, and the language itself, have been seriously criticised for been very complex [MNSB06, MS06, Hos10]. We also present examples of the pragmatic use that can be given to these metrics either during the design or implementation phase, as well as pragmatic solutions to the schemas complexity problem.

The remainder of this chapter is structured as follows. Section 5.1 presents related work in the subject of XML Schema complexity metrics. After this, the metrics used and the results obtained of calculating their values are exposed in sections 5.2 and 5.3. Sections 5.4 and 5.5 deals

with pragmatic aspects related to the metrics such as practical use and solutions to deal with complexity.

5.1 Related Work

Literature about measuring XML schemas complexity has increased in the last few years. They are based mainly on adapting metrics for assessing complexity on software systems or on XML documents [BMV05, McC76, QS05]. To our best knowledge the most relevant attempt in this topic is presented in [LKR05]. The authors conducted a comprehensive study of a sample of XML schemas and proposed a categorisation of schemas according to its size. Most of the metrics considered in this study are limited to counting distinct XML Schema features, although more complex metrics such as the Mcabbe's ciclomatic complexity were adapted to measure schema complexity. Another relevant study is [MSY05] which defines eleven metrics to measure the quality and complexity of XML schemas. In addition to several simple metrics, two composite indices were defined to measure these aspects.

In [BM09b], the authors present a more sophisticated metric that takes into account, not only the number of main schema components like the previous mentioned works, but also the internal structure of these components. A weight is assigned to every schema component based on the weight of its inner components. These metrics also recognise recursive structures as a feature that increases complexity of schemas.

In a similar way, [Vis06] proposes more advanced schema metrics, arguing that previous work in the topic only measures size as an approximation for complexity. The authors present a set of metrics to measure other structural properties of the schemas but do not provide any evidence of why these metrics are a better approximation to complexity than others.

Last about schemas complexity, we have the work in [PSH10] where a set of schema metrics are presented in the context of schema mapping. Based on a set of metrics that consider schemas size, use of different schema features and naming strategies, a combined metric is defined. The combined metric is evaluated in the context of business document standards.

Similar studies in the geospatial domain are scarce, though, an interesting discussion of complexity in the context of the *Geographic Markup*

Language (GML) can be found in Ron Lake's blog[1]. This discussion tries to identify the origin of GML complexity and use some of the metrics in [LKR05] to categorise its schemas. Our research attempts to extend this discussion to the OWS specifications, but focusing more on the complexity of the schemas themselves.

5.2 Metrics

In this section we present the metrics used in the complexity analysis for the schemas in the specifications listed in Table 3.1. We consider a wide set of metrics ranging from simple metrics that simply count schema features to more complex metrics that attempt to give an overall value for complexity or measure specific aspects of the schemas. The main criterion to select the metrics is that they could be *intuitively* understood, i.e., that we easily understand what we are measuring. In contrast to the use of *'esoteric'* metrics, which end up being of little interest to the industry [FN99].

The first group of metrics attempts to measure the raw size of the schemas and do not consider any XML-related information. For this reason, they are classified by [LKR05] as *XML-agnostic*:

- *Lines of Code (LOC)*: Total number of lines of code on the specifications' schemas.
- *Number of files (#F)*: Total number of files related to the specification. Here, we consider recursively all of the files referenced through *include* and *import* XML schema statements.

The second group of metrics are *XSD-aware metrics*, which are concerned with schema information. The metrics have been taken from [LKR05], [MSY05] and [BM09b]:

- *Number of complex types (#CT)*: All complex types, including global ($\#CT_G$) and anonymous ($\#CT_A$) complex types.
- *Number of simple types (#ST)*: All simple types, including global ($\#ST_G$) and anonymous ($\#ST_A$) simple types.
- *Number of global elements (#EL)*: All global element declarations.

[1] `http://www.galdosinc.com/archives/140`, `http://www.galdosinc.com/archives/186`

- *Number of global model groups (#MG)*: All global model groups definitions.
- *Number of global attributes (#AT)*: All global attribute declarations.
- *Number of global attribute groups (#AG)*: All global attribute groups definitions.
- *Number of global items (#GLOBAL)*: Number of global schema components (types, elements, model groups, attributes and attribute groups):

$$\#GLOBAL = \#CT_G + \#ST_G + \#EL + \#AT + \#MG + \#AG$$

- *Wildcards*: Number of times wildcards are used.
- *C(XSD)*: This metric calculates a complexity weight taking into account the internal structure of schema components. It calculates approximately the number of primitive (or atomic) information items that must be considered to fully understand a set of schemas, as well as the number of recursive branches contained in the schemas. The complete definition of the metric can be found in [BM09b].

The last group contains those metrics that attempt to measure the influence in complexity of the subtyping mechanism of XML Schema. First, we just count the XML Schema features related to the subtyping mechanism [LKR05]. After this, we define a set of new metrics to measure the influence of the use of subtyping in schemas complexity. These metrics were first introduced in [TGH11a], and are developed in greater depth in the following subsections.

- *Use of subtyping features of XML schema*: Here, we consider first basic counting metrics such as the number of substitution groups, uses of derivation by restrictions and uses of derivation by extension.
- *Data Polymorphism Rate* (DPR): This is a measure of how much polymorphism is contained in the schemas. It measures the fraction of the schema elements that are polymorphic.
- *Data Polymorphism Factor* (DPF): This metric attempts to measure how much the polymorphic elements affect the overall complexity. Depending on the number of elements or types a given element

can be substituted for, the perception of complexity can be higher or lower.

- *Schemas Reachability Rate* (SRR): This metric measures the fraction of the schemas that are "hidden" for the schema users. By hidden we mean that main schemas have dependencies on them, but these dependencies are not explicit.

For some of the metrics mentioned in this section we present three different values. These values try to separate the complexity that is located in *main schemas* from that one located on *external schemas*(see Section 3.2):

- C_O, an *overall value* that includes all of the components in the main and external schemas.
- C_L, *local value* that includes only the components in the main schemas
- C_E, a *external value* that includes only the components in the external schemas

The *overall value* (C_O) gives us an idea of the complexity of the whole schema set related to a given specification. Support for this schema set must be implemented when building a single OGC web service from scratch. *Local values* (C_L) are useful to measure complexity of a specification without considering the external schemas. This is the case when the support for external schemas is already implemented somewhere (e.g. in a library). External values (C_E) give us a measure of how much complexity we are importing into a given schema. Which of the three values is the most important depends on the problem at hand.

5.2.1 C(XSD) Metric Definition

In our study we include the metric presented in [BM09b] that measures the complexity of schemas based on its internal structure, opposed to the metrics presented so far that limit themselves to just counting schema components or features. It pays special attention to the use of recursive structures as a source of complexity to schema users. A complexity value, or *weight*, is calculated for each schema component as an aggregation of the weights of the components it contains. The overall value of the metric is calculated with the following formula:

$$C(XSD) = \sum_{i=1}^{N} C(E_{gi}) + \sum_{j=1}^{M} C(A_{gj}) + \sum_{t=1}^{K} C(EG_{gt}) + \sum_{r=1}^{P} C(AG_{gr}) + \sum_{s=1}^{R} C(CT_{gs}) + \sum_{q=1}^{Q} C(ST_{gq}) \quad (5.2.1)$$

where, the first two terms are the summation of weights of global element and attribute declarations respectively. The remaining terms are the summation of weights of global unreferenced model groups, attribute groups, complex and simple types that are declared/defined in the main specification schemas. The values N, M, K, P, R and Q are the number of global elements, attributes, unreferenced element groups, unreferenced attributes groups, unreferenced complex types and unreferenced simple types respectively. In the second group of terms only unreferenced components are considered to avoid counting them several times as they are used in the declaration of global elements and attributes.

In [BM09b] formulae are provided to calculate the weight of the different types of schema components. For example, to calculate the weight of a complex type we use the following formula:

$$w_{type} = w_{baseType} \pm [\sum_{i=1}^{N} C(E_{gi}) + \sum_{j=1}^{M} C(A_{gj}) + \sum_{t=1}^{K} C(EG_{gt}) + \sum_{r=1}^{P} C(AG_{gr})] + NRC * R \quad (5.2.2)$$

where, $w_{baseType}$ [2] is the weight of the base type. If derivation is not explicit (*anyType* is the base type) the weight of the base type is 1. If we are in the case of derivation by extension, *N, M, K, P,* are the number of non-inherited local or referenced elements, attributes, and referenced element and attribute groups, that are not related to any element containing recursion. The sum of all these values is added to the weight of the complex type. If the complex type is derived by restriction, *N, M, K, P*, are the corresponding number of schema components not inherited

[2] The notation $w_{typename}$ is equivalent to $C(CT_{typename})$

from the base type, and its weight is subtracted from the weight of the base type. In both cases *NRC* is the number of child elements that contains recursion; and R is an integer value greater or equal than 1 that can be understood as the weight given to recursion in a schema set.

5.2.2 Subtyping-related Metrics

To measure the influence of the subtyping mechanisms in complexity we introduce three new metrics: *Data Polymorphism Rate(DPR)*, *Data Polymorphism Factor* (DPF) and *Schema Reachability Rate* (SRR).

The term *Data Polymorphism (DP)* was introduced in Chapter 2 to describe the fact where XML nodes have a type (*dynamic type*) that differs from the type of its corresponding element declaration (*declared type*).

Data Polymorphism Rate

The *Data Polymorphism Rate (DPR)* is a measure of how much polymorphism is contained in the schemas. It calculates the fraction of elements contained in complex type declarations that are *polymorphic*, i.e, its dynamic type may differ from its declared type. It is expressed by the formula:

$$DPR = \frac{\sum_{i=1}^{N} PE_{CTi}}{\sum_{j=1}^{N} E_{CTj}} \qquad (5.2.3)$$

In the formula, N is the total number of complex types, PE_{CTi} is the number of elements in the declaration of the complex type CT_i that are polymorphic. E_{CTj} is the overall number of elements in the type declaration of type CT_j. For every type, a reference to a global element and an inner element declaration are considered as equals and count as 1. As a consequence, the numerator is the total number of polymorphic elements in the schemas. Similarly, the denominator is the total number of elements contained in all complex types in the schemas. The result value is in the interval [0, 1], indicating the fraction of the elements that are polymorphic. This metric is a variation of the *Polymorphic Factor* (POF) metric used in the OOP context [AM96].

Data Polymorphism Factor

The previous metric gives an idea of the number of polymorphic elements in schemas, but does not measure their influence in the overall complexity of the schemas. For instance, a polymorphic element that can be replaced by two other elements does not have the same effect in complexity than an element that can be substituted by twenty different elements. In this regard, we define the *Data Polymorphism Factor (DPF)* as follows:

$$DPF = \frac{\sum_{i=1}^{N} OE_{CTi}}{\sum_{j=1}^{N} E_{CTj}} \tag{5.2.4}$$

In this case, OE_{CTi} is the number of possible different elements that could be contained in a complex type. It is the summation of the number of elements declared in CT_i, the number of elements in the substitution groups of those elements, and the number of possible dynamic types that can have any element in CT_i different from its declared type. The denominator is the same as in the previous metric.

In the formula $OE_{CTi} >= E_{CTi}$ for all $i \in \mathbb{N}$ in the interval $[1, N]$. As a consequence the values of DPF are always equal or greater than 1, representing the factor by which the number of elements to be considered might grow when polymorphic elements are taken into account.

Let us consider, for example, the schema fragment in Listing 2.3. In this code *ContainerType* has an element declaration of declared type *Base*, but it may also have a different dynamic type: *Child*. As there are no substitution groups in the fragment the value of $OE_{ContainerType} = 1 + 1 = 2$. The value of OE_{CTi} for the other two types is straightforward to calculate as they do not contain polymorphic elements. The metric value for the whole fragment is calculated as follows: $DPF = (2 + 1 + 2)/(2 + 1 + 1) = 1.25$

Schema Reachability Rate

The last metric proposed in this chapter, *Schema Reachability Rate (SRR)*, attempts to measure the fraction of *imported schema components* that are hidden (not referenced explicitly) by the subtyping mechanisms. As mentioned in Section 3.2, the schemas of OWS specifications reutilise

other schemas by *importing* them from the main schemas. An imported component may be referenced directly if it is explicitly mentioned in the declaration of another schema component, or indirectly, if it is in the substitution group of a referenced element, or its derived from a type that is referenced directly. For example, it is not clear for everybody that if we are using GML 3.1.1 in our schemas and we define an inner element to be of type *gml:AbstractFeatureType*[3], this element may have 13 different dynamic types (considering only GML types) in XML documents based on these schemas.

To calculate SRR, we define first G_S, G_{SH} and $V_{Rm}(G)$ as follows:

Definition 1: *We define G_S for the schemas in a specification S as the directed graph $G_S = (V_S, E_S)$, where vertices in V_S, are all of the global schema components declared in all of the schemas related to S (main and external schemas). E_S are directed edges between these vertices. An edge from v_i to v_j exists if v_j is used somehow in the declaration of v_i.*

Definition 2: *We define G_{SH} for the schemas in a specification S as the directed graph $G_{SH} = (V_{SH}, E_{SH})$, where $V_{SH} = V_S$. E_{SH} extends E_S by including also non-explicit dependencies, i.e. an edge from v_i to v_j exists if v_j is used somehow in the declaration of v_i, or if v_j, is in the substitution group of an element referenced from v_i, or v_j is a type derived from a type used in the declaration of v_i.*

Definition 3: *We define, $V_{Rm}(G)$ for a directed graph $G = (V, E)$ and V_m, a subset of V, as the subset containing all of the vertices in V that are reachable from V_m*

Based on these definitions, if we consider that $V_m(G)$ is the subset of $V(G)$ containing the schema components included in the main schemas, $V_{Rm}(G)$ would contain any schema component that is reachable from the main schemas. In the case of G_S, this will be components reachable through explicit dependencies, and in the case of G_{SH}, these are reacheable components through explicit and non-explicit dependencies. Using these vertex sets the SRR metric is calculated as follows:

[3]The prefix *gml* refers here and in the remainder of this document to the namespace *http://www.opengis.net/gml*

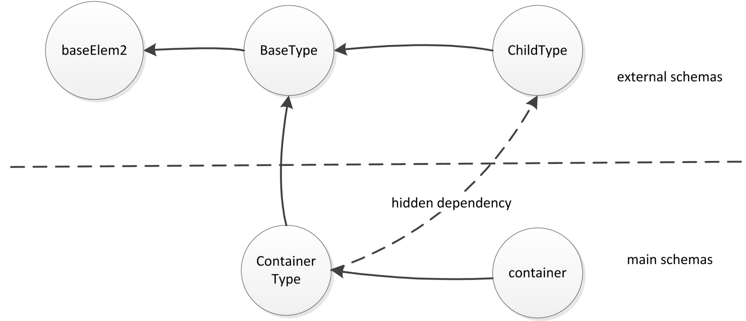

Figure 5.1: Graph of relations between schema component for schema fragment in Listing 2.3

$$SRR = \frac{|V_{Rm}(G_{SH})| - |V_{Rm}(G_S)|}{|V_S|} \tag{5.2.5}$$

The metric measures the fraction of schema components a specification depends from, but that are not explicitly referenced from any component in the main schemas, or any component reachable through explicit dependencies from the main schemas.

Figure 5.1 shows the graph of component relations for the schema fragment in Listing 2.3. If we ignore the hidden dependency between *ContainerType* and *Child* we obtain G_S, otherwise we obtain G_{SH}. If we consider, for example, that the declaration of element *container* and type *ContainerType* are located in the main schemas, and *Base* and *Child* and *baseElem2* declarations are located in external schemas, we can calculate the value of DPRF for the main schemas: $V_m = \{container, ContainerType\}$, $V_{Rm}(G_S) = \{container, ContainerType, Base, baseElem2\}$, $V_{Rm}(G_{SH}) = \{container, ContainerType, Base, Child, baseElem2\}$, so:

$$SRR = \frac{5-4}{5} = 0.2$$

This value means that a fifth of the schema components are referenced through non-explicit dependencies.

5.2.3 Measurement Process Description

The general process used to calculate the values of the metrics for a specification P (except for C(XSD) and SRR) is described by the following algorithm:

1. *Input:* $I = \{x \mid x$ *schema files that belongs to main schemas of* $P\}$
2. $S = I, C_O = 0$ $C_L = 0$, $C_E = 0$
3. *While there are unprocessed files in S*
   (a) *Get next unprocessed file x from S*
   (b) *Calculate C(x), value of the metric for x*
   (c) *If x belongs to main schema files* $C_L = C_L + C(x)$
   (d) *Else If x belongs to external schema files* $C_E = C_E + C(x)$
   (e) $C_O = C_O + C(x)$
   (f) *Calculate* $W = \{y \mid y$ *file imported or included in* $x\}$
   (g) $S = S \cup W$
4. *Output:* C_O, C_L, C_E

The most important step is the calculation of *C(x)* that is the value of the metric for the corresponding schema file being analysed. For *C(XSD)* a more complicated recursive algorithm must be applied to calculate the metric values. Similarly, to calculate SRR, a modified version of the *breadth-first search* algorithm presented in [CLRS01] is used.

The measurement process considers all of the external files included or imported from the main schemas directly or indirectly. In this regard, it must be noticed that a particular implementation might not need all of these schemas. As a consequence, the value of the metrics must be taken as a worst-case estimation of the size or complexity of the schemas.

5.3 Results

The results of applying the metrics mentioned before to the specification schemas listed in Table 3.1 are shown in the following subsections.

5.3.1 XML-Agnostic Metrics

In this section we calculate the values of *XML-agnostic metrics*, which are those that do not consider XML-related information. The total amount of lines of code (LOC) and the number of files (#F) give us a raw idea of the size of a given schema set. Table 5.1 shows the value

of the metrics for the considered OWS specifications. According to the categorisation for LOC values presented in [LKR05], a schema set with between 10,000 and 100,000 LOC is considered large. Values between 1,000 and 10,000 correspond to medium-sized schemas. And values between 100 and 1,000 correspond to small-sized schemas. There are also categories for mini schemas, below 100 LOC, and huge schemas, above 100,000 LOC

Table 5.1: Lines of code (LOC) and number of files (#F)

	SOS 1.0	WFS 2.0	WCS 2.0	SPS 1.0	WPS 1.0	WMS 1.3.0	GML 3.2.1	GML 3.1.1	SML 1.0.1	O&M 1.0	KML 2.2.0
LOC_O	17,877	3,631	15,416	14,439	3,412	774	11,349	10,110	14,242	14,458	3,388
LOC_L	1,006	781	407	1,234	1,426	652	7,349	9,598	1,416	216	1,708
LOC_E	16,871	2,850	15,009	13,205	1,986	122	4,000	512	12,826	14,242	1,680
$\#F_O$	87	23	87	71	29	3	59	33	50	52	3
$\#F_L$	16	1	5	20	13	2	29	32	5	2	2
$\#F_E$	71	22	82	51	16	1	30	1	45	50	1

Considering the overall values, 7 out of 11 of the specifications are considered large, three of them are considered medium and only one is considered small. The specifications related with sensors (SOS, SPS, SensorML, O&M), as well as WCS, exhibit the higher values for the metrics. It is not a coincidence that they are the ones with higher number of dependencies from other specifications. On the other hand, WMS turns out to be the simplest (which is maybe why it is the most widespread) of the specifications, being in terms of lines of code, about 20 times smaller than SOS. WMS does not depend on any major external schema for its definition. It might be a little bit surprising that WFS presented such small figures when compared to other web service interface specifications. The reason for this is that WFS schemas do not reference directly the schemas for the data exchanged between clients and servers. An actual implementation of WFS should include some version of GML; hence if we use for example GML 3.2.1 the overall LOC would be similar to the one of SPS. The same applies for WPS, which is designed to be complemented with application-specific schemas, so its final metric value will depend from the specific implementation.

The proportion of *local v. external values* for both metrics follows well-defined patterns (Figure 5.2). Most of the size of web service interfaces is imported from the encodings defining the data formats. Encodings can

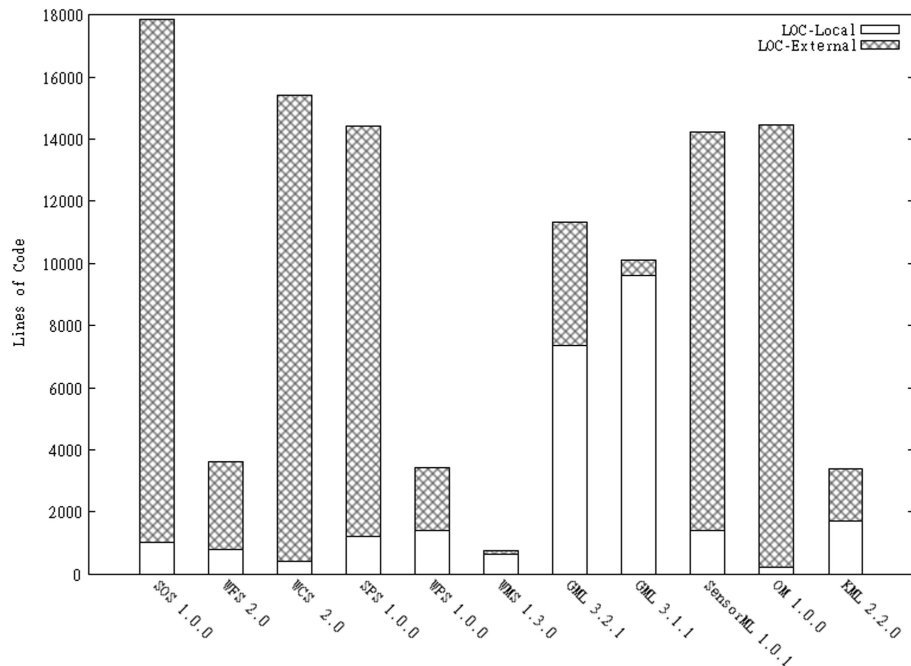

Figure 5.2: Values for the LOC metric

have lower or larger local values depending whether they have more or less dependencies from other encoding specifications. In the cases of SOS, WCS, SPS, SensorML and O&M, a larger portion of the metric values is due to their dependencies to some of the GML versions. In all of these cases, the corresponding GML value of the metric accounts for more than 50% of the overall value.

5.3.2 XSD-Aware Metrics

In this section we calculate the values of *XSD-aware metrics*, which are concerned with schema information. The metrics are divided into those that simply count main schema features, and C(XSD), which takes the internal structure of components into account to assign a weight value to each component. These weights are aggregated later to calculate an overall complexity value for a schema set.

XSD-Aware Simple Metrics

We present first the results of applying *XSD-aware metrics* that count the number of main schema components. We start with the number of complex types (#CT), which is considered paramount for measuring complexity because it measures the number of structured concepts modelled by the schemas [LKR05] . Moreover, types are the fundamental concept when schemas are used to write (or generate) XML data binding code. The #CT metric includes global complex types, as well as anonymous complex types. The number of complex types by specification is shown in Table 5.2.

Table 5.2: Number of complex types (#CT)

	SOS 1.0	WFS 2.0	WCS 2.0	SPS 1.0	WPS 1.0	WMS 1.3.0	GML 3.2.1	GML 3.1.1	SML 1.0.1	O&M 1.0	KML 2.2.0
$\#CT_O$	740	163	797	585	99	33	654	394	610	615	153
$\#CT_L$	37	61	16	54	53	33	366	394	109	5	69
$\#CT_E$	703	102	781	531	46	0	288	0	501	610	84

Schemas with #CT in the range 256-1,000 are considered *large*, in the range 100-256 are considered *medium* and *small* with #CT between 32 and 100 [LKR05]. As complex types model concepts, we can state that higher values of the metric imply higher conceptual complexity. The values of the overall metrics for all of the 11 specifications belong to these three ranges, 7 of them are large, two of them is medium-sized and the other two are small schemas. Again, WCS, SOS and SPS are among the most complex schemas and WMS is the simplest. We have to take into account for WFS and WPS, as in the previous group of metrics, that they are meant to be combined with other encoding specifications.

A categorisation of the schemas based on the number of other schema components is not provided in the literature. Nevertheless, they can give us some idea of size of schemas and how often these features are used in the specifications. Table 5.3 shows the overall values of the metrics for these schema components, which are also included in Figure 5.3. These values reinforce the idea of having a clear differentiation between a first group containing large specifications (SOS, SPS, WCS, both GML versions, SensorML and O&M), a second group containing medium-sized specifications (WFS, KML) and a last group with small specifications

Table 5.3: Main XML features metrics (except #CT)

	SOS 1.0	WFS 2.0	WCS 2.0	SPS 1.0	WPS 1.0	WMS 1.3.0	GML 3.2.1	GML 3.1.1	SML 1.0.1	O&M 1.0	KML 2.2.0
#ST	118	46	74	103	15	5	70	64	100	100	42
#EL	727	156	754	593	64	60	653	485	586	588	292
#MG	28	3	14	19	7	0	7	12	26	26	0
#AT	23	15	20	15	16	9	17	15	33	33	0
#AG	40	12	17	37	9	7	39	35	39	39	2
#ALL	1498	354	1,625	1,150	174	80	1,414	986	1,249	1,256	441

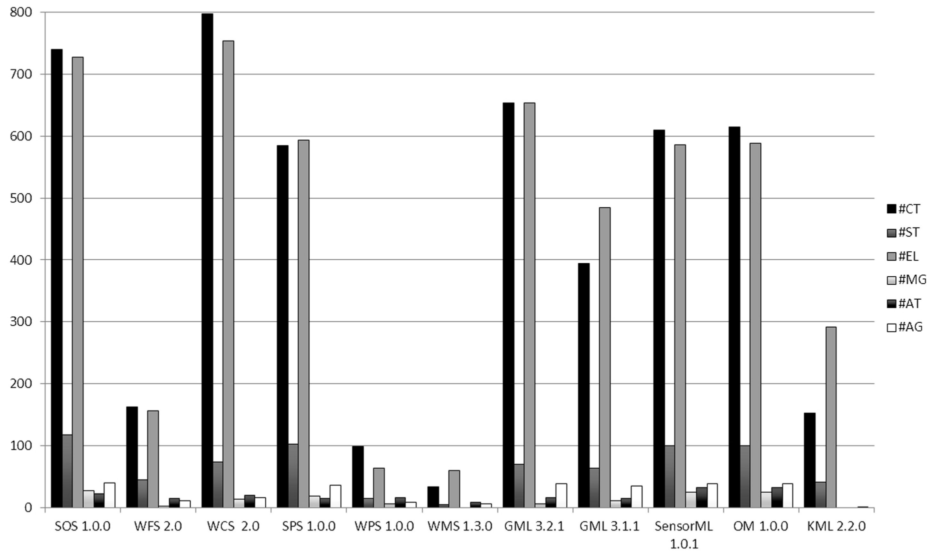

Figure 5.3: Distribution of the number of schema components in the specification schemas

(WPS and WMS). In Figure 5.3 we can observe the correlation that exists between the values of the metrics. This observation suggests that the coding style used in the schemas is consistent through all of the specifications.

Last, we count wildcards, which allow schema designers to specify extensibility points using *any* tags. By using these tags in a complex type definition, we indicate that any global element can occupy that place in

an instance document. The scope of the valid elements the wildcard is replaced by can be constrained by their namespace. The use of wildcards is widespread with the purpose of keeping schemas extensible, but they greatly complicate the processing of instance files.

A discussion about why the use of wildcards should be avoided when designing web service interfaces can be found in [Pas06]. We just want to highlight the fact that when parsing an XML instance we cannot be sure of what we will find in the place of the wildcards, so we must be ready to find almost anything. This obviously makes the source code for processing the instances files more complicated. If instead of writing the code manually we use a code generator, the presence of wildcards limits their possibilities of performing optimisations. Table 5.4 shows how wildcards are used in the specifications. In this case, only the specifications already labelled as large and medium-sized make use of this feature. It is worth noticing the case of KML where 23 wildcards have been found. 21 of these wildcards are located in an external schema file imported by KML containing the definition of the *eXtensible Address Language* (xAL) defined by OASIS (Organization for the Advancement of Structured Information Standards)[4]. The external schema is referenced in a single place in the KML schemas.

Table 5.4: Use of wildcards

	SOS 1.0	WFS 2.0	WCS 2.0	SPS 1.0	WPS 1.0	WMS 1.3.0	GML 3.2.1	GML 3.1.1	SML 1.0.1	O&M 1.0	KML 2.2.0
Wildcards	13	13	5	9	0	0	9	7	10	11	23

5.3.3 C(XSD)

C(XSD) is introduced in [BM09b] and it calculates a weight for each schema component taking the internal structure of the component into account. For each schema component a complexity value or weight is calculated. These values are then aggregated to calculate an overall complexity value for the schemas. This result is an approximation of the number of primitive (or atomic) information items that must be considered to fully understand a set of schemas. The metric also considers the

[4] `http://www.oasis-open.org/home/index.php`

influence in complexity of recursive branches. Tables 5.5 and 5.6 show the value of the metric for the web service and encoding specifications considered here. Figure 5.4 shows the value of the metric for the specifications, assuming a value of R = 2 (weight given to recursion).

Table 5.5: C(XSD) values for OWS specifications

	SOS 1.0	WFS 2.0	WCS 2.0	SPS 1.0	WPS 1.0	WMS 1.3.0
C_{XSD}	261,238 + 2,381R	1,960 + 16R	209,997 + 1,171R	96,451 + 885R	1,578 + 2R	707 + 3R

Table 5.6: C(XSD) values for encoding specifications

	GML 3.2.1	GML 3.1.1	SML 1.0.1	O&M 1.0	KML 2.2.0
C_{XSD}	150,094 + 839R	74,609 + 611R	244,827 + 2,267R	233,194 + 2,137R	74,940 + 614R

The results of applying this metric to the specifications are much in the same course of the previous metrics. A couple of interesting facts deserve more attention, though. For example, the structural complexity of elements in SensorML and O&M is higher than in WCS even when the latter contains more complex types in its definition. SensorML and O&M contain the most complex schema components if analysed individually. The schema component with the higher value for the metric is *Component* with 23016 + 219R. Coincidentally, this element contains the highest number of recursive branches in its definition. Similarly, the structural complexity of KML is similar to the one of GML 3.1.1 even when it contains less than half of its complex types and one third of its LOC values.

These results reinforce the idea presented in [BM09b] that simpler metrics such as #CT do not give us all of the information we need about the complexity of a given schema, as the internal structure of a type can be very complex or very simple. Hence, they should not have the same weight in the overall complexity.

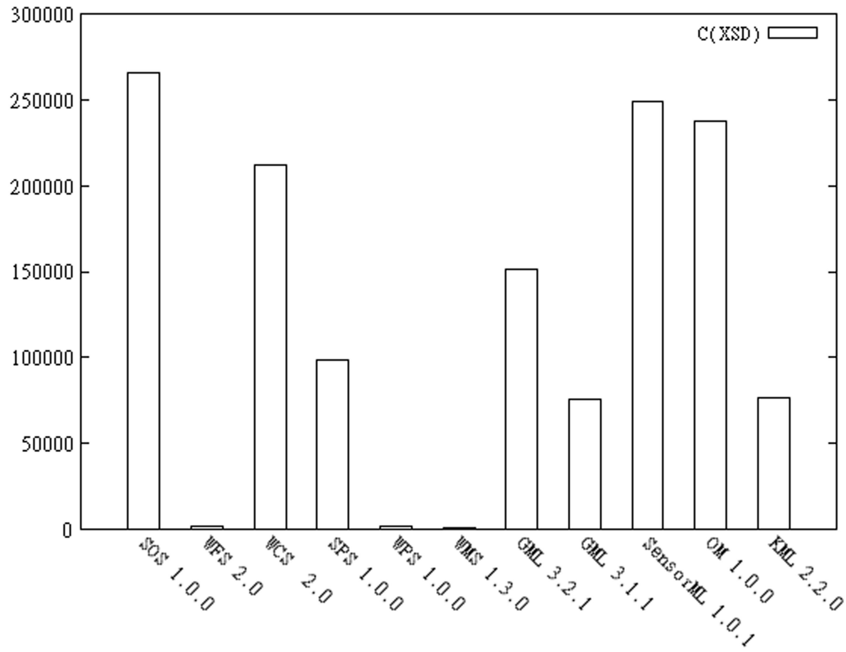

Figure 5.4: C(XSD) values for R=2

5.3.4 Subtyping Metrics

XML Schema subtyping mechanisms were introduced in Section 2.2.1. In this category we count first number of abstract elements or types (#AET), number of substitution groups (#SG) and number of complex types derived by restriction or extension (#TE and #TR). The values of these metrics are shown in Table 5.7.

Table 5.7: Use of subtyping mechanisms

	SOS 1.0	WFS 2.0	WCS 2.0	SPS 1.0	WPS 1.0	WMS 1.3.0	GML 3.2.1	GML 3.1.1	SML 1.0.1	O&M 1.0	KML 2.2.0
#AET	61	15	74	52	2	2	63	47	53	53	124
#SG	83	11	123	72	4	0	112	60	72	72	10
#TE	291	55	356	243	30	4	300	182	241	243	55
#TR	59	1	15	54	1	0	13	53	57	57	0

The results show that subtyping mechanisms are widely used in the specifications leading to an elevated number of non-explicit dependencies between schema components. This may lead to inadvertently overlook important details when analysing dependencies. An important detail about type derivation by restriction is that it cannot be mapped *"smoothly"* to an object-oriented programming language [Oba95, Das01]. Hence, specifications that make a large use of this feature may suffer from difficulties when schemas are mapped to these languages, presumably using a code generator. High values of #TR could also be interpreted as a potentially flawed design, because if a large number of subtypes requires their content to be redefined there is a good chance that their ancestors has not been selected/defined in a correct way.

From the values of #AET in the table the one corresponding to KML stands out from the rest. KML makes a wide use of abstract elements and types. On the other hand, it has a relatively low use of substitution groups and it does not use derivation by restriction at all. These values contrast with the one for GML schemas, which has lower values for #AET but have higher values for the rest of the metrics in Table 5.7.

Data Polymorphism Rate

The values for the DPR metric are presented in Table 5.8. From these results we can observe that simpler specifications contain zero or a low degree of polymorphism. The rest of the specifications have a similar degree ranging from 12 to 15%. Saying if these values are too high or not is not a trivial task; however in [AM96], analysing polymorphism in the context of OOP, it is stated that a values of POF above 10% are expected to reduce the benefits obtained with an appropriate use of polymorphism. This is because highly polymorphic hierarchies will be harder to understand, debug and maintain.

Table 5.8: DPR values for the OWS specifications

	SOS 1.0	WFS 2.0	WCS 2.0	SPS 1.0	WPS 1.0	WMS 1.3.0	GML 3.2.1	GML 3.1.1	SML 1.0.1	O&M 1.0	KML 2.2.0
DPR	0.13	0.12	0.15	0.13	0.05	0	0.15	0.14	0.13	0.13	0.05

Data Polymorphism Factor

Table 5.9 shows the values of DPF for the schemas of the different specifications. These results show that the effect of polymorphic elements on SOS and SPS is higher than in WFS and WCS. Presumably this is caused by the larger number of dependencies of the sensor-related specifications. SOS and SPS have a lot of common dependencies, that is why DPR and DPF values are basically the same for both specifications. As the simplest specifications barely contain polymorphic elements, the values of DPF for them are equal or close to the minimal value, 1.

Table 5.9: DPF values for the OWS specification schemas

	SOS 1.0	WFS 2.0	WCS 2.0	SPS 1.0	WPS 1.0	WMS 1.3.0	GML 3.2.1	GML 3.1.1	SML 1.0.1	O&M 1.0	KML 2.2.0
DPF	2.20	1.47	1.48	2.20	1.05	1	1.40	2.17	2.15	2.16	1.13

Schema Reachability Rate

The different values used to calculate the SRR metric, as well as the actual value of these metrics are shown in Table 5.10. The results show that for SOS, WCS, and SPS more than 60% of the schema components that could be used in XML documents are not referenced explicitly from the schema component in the main schemas, or any component that is referenced from them. This high rate suggests that the effect of the subtyping mechanism in schemas complexity is enormous. For the rest of the specification the effect goes from moderate (WFS, WPS) to non-existent (WMS).

Table 5.10: SRR values for the OWS specification schemas

	SOS 1.0	WFS 2.0	WCS 2.0	SPS 1.0	WPS 1.0	WMS 1.3.0	GML 3.2.1	GML 3.1.1	SML 1.0.1	O&M 1.0	KML 2.2.0
$\lvert V_{Rm}(G_{SH})\rvert$	1,277	321	1,349	1,058	146	71	1,334	975	1,070	1,076	398
$\lvert V_{Rm}(G_S)\rvert$	319	245	220	203	126	71	1,033	975	325	180	398
$\lvert V_S\rvert$	1,498	353	1,625	1,266	179	80	1,414	986	1,249	1,256	399
SRR	0.64	0.22	0.69	0.68	0.11	0	0.21	0	0.59	0.71	0

Specifications with higher values of SRR are those having a larger number of dependencies and higher values of DPR and DPF. The consequence of having a large number of polymorphic elements, which can be replaced by a large number of elements or types, is that many of the schema components dependencies are not explicit. This, on the other hand, can be the cause of errors or unexpected situations to schema designers and users. Let us consider an example for GML 3.1.1: *gml:AbstractFeatureType* references the global element *gml:name*, which is the head element of a substitution group containing other global elements such as *gml:srsName*, *gml:csName*, *gml:ellipsoidName*, etc. This reference to *gml:name* is inherited by a large set of types which derives, directly or indirectly, from *gml:AbstractFeatureType* such as *gml:PolygonType*. At this point, a polygon instance whose *gml:name* element is substituted by an *gml:srsName* or *gml:ellipsoidName* element is valid against the schemas, although this substitution may not make sense at all.

In an experiment presented in Chapter 8, it is reported that only 29.5% of the components of the SOS schemas are used in of a group of 56 server instances. The cause of this low usage may be that the schemas are perhaps more complex than needed, but may also be that server developers do not fully comprehend all of the relations between schema components mostly because these relations are not explicit, nor easy to discover.

5.3.5 Discussion

The results of the analysis show that at least half of the presented specifications can be considered as large and/or complex according to all of the metrics that provide some sort of categorisation. Most of the metrics coincide in finding a clear differentiation between a first group containing large specifications (SOS, SPS and WCS), a second group containing medium sized specifications (WPS and WFS), and a third group of simple specifications (WMS). More complex specifications, as a general rule, are those that present a larger number of dependencies from other specifications. The results show that the subtyping mechanisms, which is by itself very complex, is extensively used in the schemas, being an important source of complexity for schema users.

The results also suggest that for obtaining a general view of the complexity of a specification more than one metric must be used because none of them provides an integral measure that solely could order the specifications by its complexity or size. This multidimensional view of complexity

has been mentioned in the research literature, e.g. [KMB04, MD01]. For example, a metric such as C(XSD), which at first sight looks like an integral complexity metric, do not measure the effect of wildcards or even the effect of the subtyping mechanisms. In the same manner, a metric such as DPF may be used to compare schemas with approximate sizes, because if the size difference is too big these values may be irrelevant.

5.4 Practical Use of Metrics

The low penetration of software metrics in the software industry has been frequently highlighted [FN99, MD01, KMB04]. This has been attributed to several factors, such as much academic research is irrelevant to industrial needs, mainly because academic models often rely in parameters which cannot be measured precisely in practice or they mostly focus on detailed code metrics and not on relevant metrics for process improvement. Another factor is the fact that sometimes it is hard to prove that a metric actually measures the attribute it claims to measure, specially if these attributes are qualitative and subjective in nature, as is frequently the case for software attributes such as quality, maintainability or reliability. For these reasons we consider pertinent to provide examples of how the metrics presented in this chapter could be used in practice: specifically we illustrate how metrics aid the evaluation of design decisions and how they can be used to follow up the evolution of different versions of the schemas.

These use case scenarios provide examples of the use of software metrics in the two different ways introduced in [MD01]: *predictive*, before evolution occurs; and *retrospective*, after evolution occurs. The use case scenarios are presented in a succinct way due to space limitations.

5.4.1 Use Case Scenario: Evaluating Design Decisions

Using metrics in a predictive way, they can be helpful to *decision making* during the specification design phase. For example, a typical decision that must be made is *redefine vs. reuse*, when we must choose between reusing components in existing specification schemas or redefining them. Using appropriate metrics we can have an idea of the effect of one decision or the other in the final size and complexity of the schemas.

For example lets us analyse how WMS 1.3.0 could be affected if we make it compatible with OWS Common Specification 1.1.0 [OGC07b] and GML 3.1.1. With a few simple transformations to WMS 1.3.0 schemas,

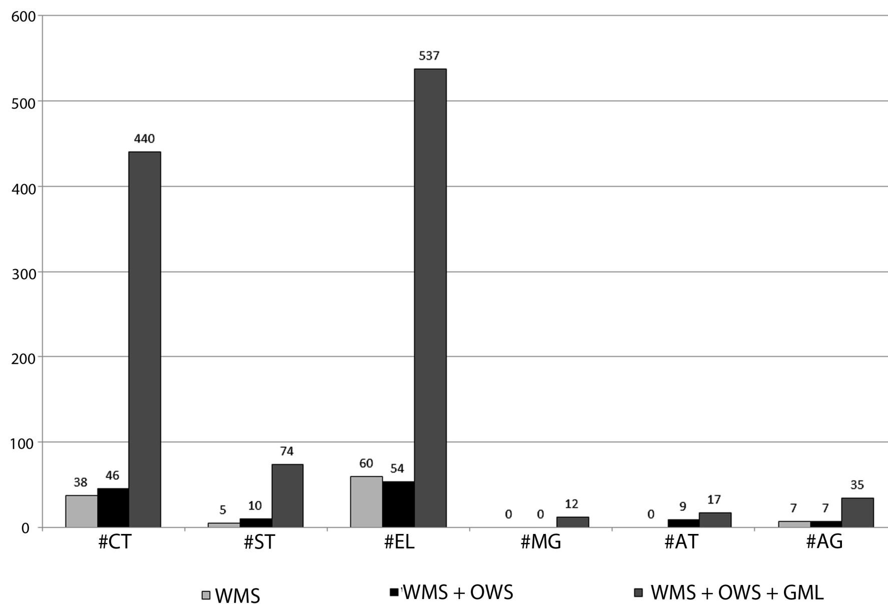

Figure 5.5: XSD-aware simple metrics values for WMS 1.3.0 and its merging with OWS and GML

we could change the *Capabilities* file of this specification to follow the structure defined in OWS common. From Figure 5.5 we can see that using OWS Common does not make WMS much bigger, with the added value that support for OWS common can be reutilised from other specifications if it has been already implemented.

In a second step we could try to reuse components from GML 3.1.1 to define WMS layers. In its broadest sense, a *feature* is defined as an abstraction of a real world phenomenon, and a *geographic feature* is a feature associated to a location relative to the Earth. According to this definition we might consider a WMS layer as a geographic feature and as such, we could define it as a subtype of *gml:AbtractFeatureType*, defined on the GML schemas. After modifying the WMS schemas to point to *gml.xsd* and calculating the values of the metrics shown in Figure 5.5, we can see that adding GML is not very helpful, as complexity and size of the specification is dramatically raised.

5.4.2 Use Case Scenario: Studying Specifications Evolution

A valid use of metrics in a retrospective way could be the analysis of schema evolution for a given specification. Evolution of the schemas may be useful for understanding how and why schemas have grown (or shrunk), to help choosing which version is more appropriate for a given application, or to estimate development effort when related web services specifications are implemented. They can also give us some hints about what to expect in the future for a certain specification.

For example let us consider the evolution of GML. Figure 5.6 a and b show the values of the metrics #F, #CT, #EL, #ST, #AT, #AG, and #MG[5]. In the figure we can observe the growing trend followed by the size of GML schemas since its first version. We can observe that the number of files has been almost multiplied by 20 since the first to the last version of the specification. Until the advent of GML 3.2.1, the trend was that numbers inside versions corresponding to the same major revision (1.x, 2.x) were similar, but in version 3.x the number of complex types and the number of files have grown more than 90% from version 3.0.0 to version 3.2.1.

The values of the metrics can help to estimate the effort to upgrade from one version of GML to another. It can be very useful to figure out that upgrading a system from using GML 3.1.1 to version 3.2.1 could not be as straightforward as someone might expect, based on the thought that the latter is supposed to be a *not-so-large* revision of the former. If we take a closer look to local and external values of these metrics we can see that this "unexpected" growth in size is because of the dependencies of GML 3.2.1 from the ISO 19139 schemas[6].

Apart from the metrics introduced in previous sections, we could use other metrics to explore and understand the changes from a GML version to the other. For example, we can consider the number of types kept and not kept in the GML namespace from one version to the other. If we consider that two types are the same if they have the same name, we can say that 305 types were kept, 153 were erased and 113 new types were introduced in the GML namespace in version 3.2.1 if compared with

[5]GML 1.0.0 is not defined using XML Schema. The metric values shown for that version of the specification are the result of converting from DTD to XML Schema using the XML editor <oXygen/>: `http://www.oxygenxml.com`

[6]ISO/TS 19139:2007. Geographic information – Metadata – XML schema implementation (`http://www.iso.org/iso/catalogue_detail.htm?csnumber=32557`)

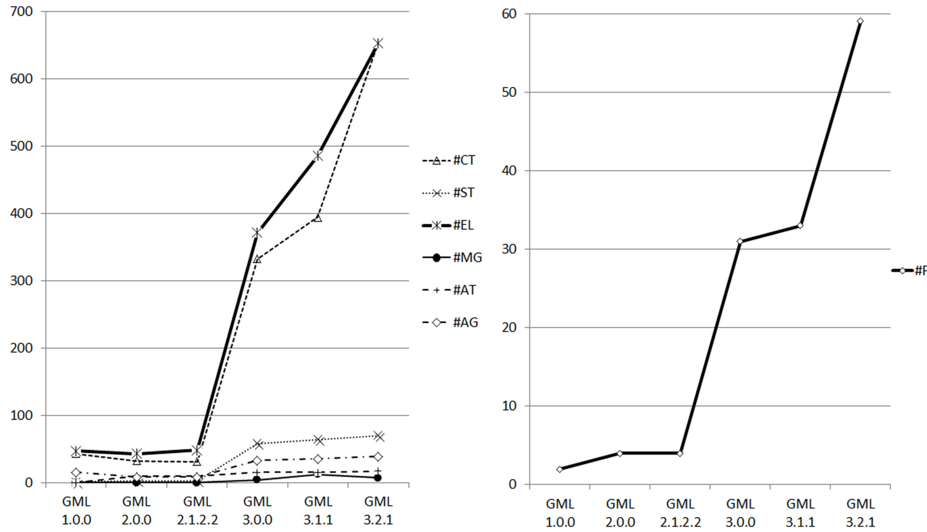

Figure 5.6: Following GML evolution through metrics

version 3.1.1. We can refine the type equality statement to consider in addition to the type name, its location or other more sophisticated criteria if we need more precision in the comparison.

5.4.3 Other Possible Scenarios

Metrics can also help us detecting potential problematic issues related to implementation. For example, as mentioned in Section 5.3.4, derivation by restriction tend to cause problems when mapped to object-oriented languages. Another potential problematic implementation issue is the use of similar names that could cause name collisions when using generators. For example, GML 3.1.1 contains several element names differentiated only by casing or by the use of underscores or similar symbols as a prefix: *gml:FeatureStyle* and *gml:featureStyle*, *gml:TopologyStyle* and *gml:topologyStyle*, *gml:Surface* and *gml:_Surface*, etc. Last, metrics can also be used to asses the effectiveness of different solutions to the problem of complexity of schemas as will be presented in more detail in the next section.

5.5 Pragmatic Solutions to Complexity

In this section we expose several possible solutions to manage the complexity of the OWS specification schemas. Some of the metrics presented before are used to illustrate the effectiveness of each solution.

5.5.1 XML Data Binding Code Generators

The use of automatic tools for code generation may reduce considerably the work of developers. In fact, it seems that many schema designers take for granted that code generators will be used to generate XML processing code for the specification schemas. Let us consider here, a scenario where XML data binding code for the SOS schemas must be produced. To produce this code we will use the code generators presented in Section 2.3.2: JAXB, XMLBeans and XBinder. In the case of XBinder the code will be targeted to Android. All of these generators need, apart from the generated code, a set of supporting libraries that must be included in our applications at execution time.

Figure 5.7 shows a comparison of the code generated with these tools regarding the size of the compiled generated code. In addition to this value, the size of the supporting library and the combined size of generated code and libraries are presented.

The results show that in all cases the overall size is rather large, if we look at them taken into account the inherent constraints of mobile devices. The values for JAXB stands out among the rest, being significantly smaller. Still, these values are too large to be adequate for execution in a mobile device, in addition to the fact that JAXB is not supported in J2ME nor Android Java API. In the opposite direction, the figures for XMLBeans are also significant, the generated code, as well as the library code, are much larger than the other generators. The main reason seems to be that it was never meant to be used with such large schemas, as such it uses a more complex mapping from schema components to programming language objects.

The differences in code distribution between JAXB and XBinder seem very interesting. XBinder, when used to generate code for mobile devices, uses very light supporting libraries, but at the expense of moving most of XML processing to the generated code. On the other hand, JAXB has all of the XML processing in the supporting libraries and generates clean and compact code[7].

[7] In some stage of the investigation we use the XSD Schema Definition Tool for

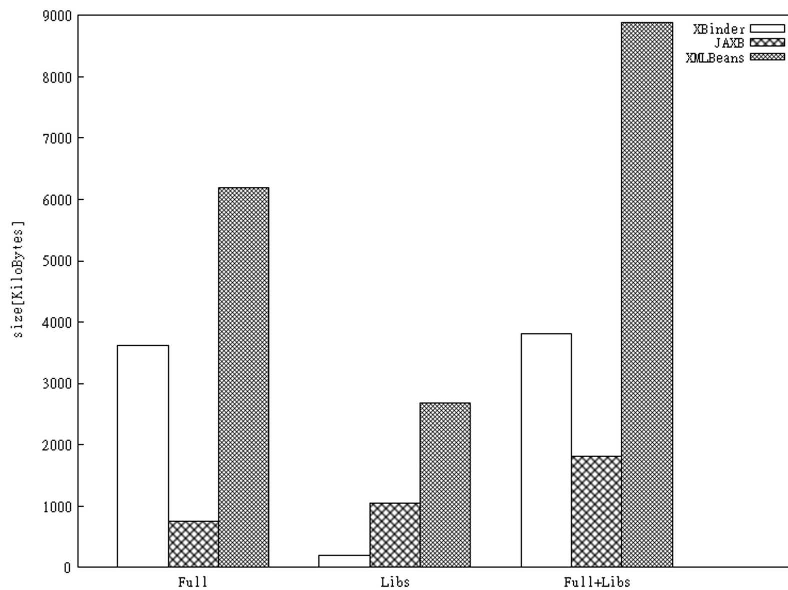

Figure 5.7: Comparing size of code (KBs) for different code generators

This comparison between code generated using different tools shows that the size and complexity of the schemas has a direct impact on the code generated from them. In many desktop or server environments the effects of this impact, although should not be taken lightly, can be reduced by the use of powerful hardware. Unfortunately, this is not the case for mobile devices, where limitations regarding memory, processing power, and its close relation to battery life do not have a solution yet.

Last, we would like to point out that for none of the generators the binary code could be directly produced without needing some kind of adjustments. The adjustments could be related to modifying code by hand to avoid compilation problems related to the use of derivation by restriction, or changing configuration parameters to avoid name clashes,

.NET platform `http://msdn.microsoft.com/en-us/library/x6c1kb0s(v=VS.100).aspx`. The code generated by this tool was similar to that of JAXB, very clean and compact, because all of the XML processing was located in the supporting libraries. In .NET Framework 4.0, the size of the supporting library for XML processing is 2156 KB.

or the failure of the generator to follow the intricate dependencies between schema components.

5.5.2 Profiles

The use of *profiles*, used here in its broadest sense as *subset of the schemas*, is a well-known solution to the problem of complexity of schemas. Even, a subsetting tool is included with GML 3, to extract subsets of the GML schemas. In addition, a set of standard profiles for GML has been defined such as the *Simple Feature Profile* (SFP)[OGC06a], *Common CRS* [OGC05a] or *CRS Support Profile* [OGC05b].

A similar solution to the use of profiles is what we call *"selective importing"*, where only the used schemas of a given specification are imported instead of the whole specification. For example, if we need support for GML features in our schemas, we could import *feature.xsd* directly instead of *gml.xsd*[8].

Following the steps of the GML subsetting tool, [TGH11b] presented an algorithm to extract customised schemas depending on specific application needs. This algorithm uses a set of instance files that must be processed by an application to identify which parts of the schemas are really necessary. Although the example presented there is related to mobile applications based on the SOS specification the algorithm can be applied to other specification schemas as well. This algorithm will be presented in detail in Chapter 6, as it is the core of the approach we are proposing to deal with schemas complexity.

Let us suppose that the portion of SOS schemas that belongs to GML 3.1.1 used in the application in [TGH11b] are all included in SFP and the set of schemas that starts in *feature.xsd*. In this case, we could compare the following approaches according the some of the metrics presented before:

- Use the whole GML schemas (by importing *gml.xsd*)
- Use the Simple Feature Profile
- Use selective importing (by importing *feature.xsd*)
- Use the GML subset extracted using the algorithm in [TGH11b]

[8] This is not always possible such as in the case of GML 3.2.1. This version is designed in a way that schemas using GML schemas can only be validated correctly if they import *gml.xsd*

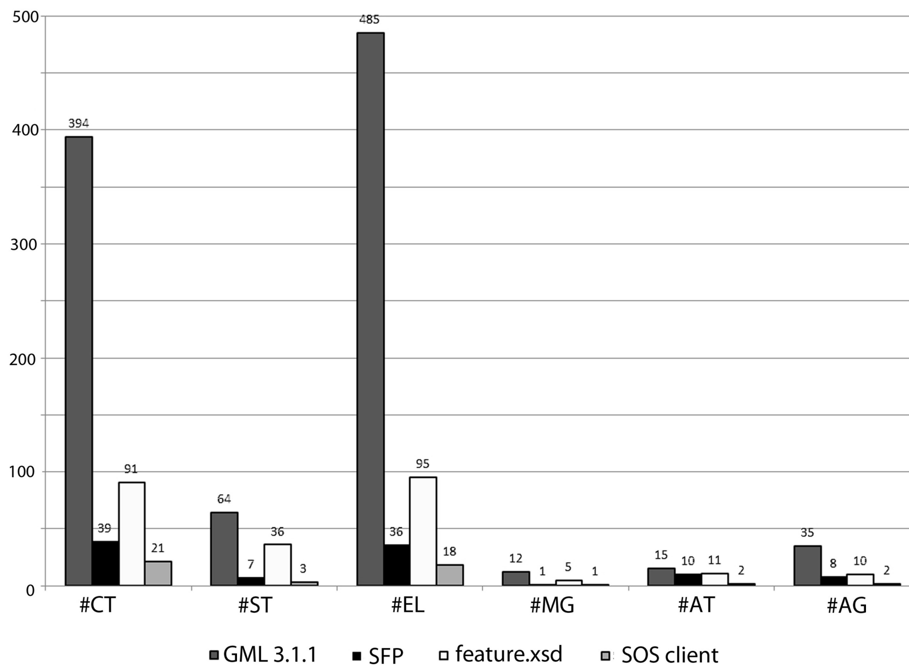

Figure 5.8: Comparing effectiveness of different approaches to deal with the complexity of schemas

The result of counting the main schema components in each case is presented in Figure 5.8. It is obvious from the figure that any of the approaches that try to avoid the use of the whole schemas could reduce considerably the overall schemas complexity. We could also conclude that the use of more specific solutions instead of generic ones could greatly simplify the implementation of real systems.

5.5.3 Using the Linked Data Style

A second pragmatic solution could be the use of the *"linked-data style"*. Linked data is a paradigm that advocates for, similarly to the web of hypertext, building the *Web of Data* by linking the information contained in documents in the web with other documents containing related information. This can be accomplished by following for basic principles: Use URIs as names for things, use HTTP URIs so that people can look up

those names, when someone looks up a URI provide useful information using the standards (RDF, SPARQL), and include links to other URIs [BL06].

As pointed out in [SGD10], since its inception GML has provided a mechanism to implement this paradigm of distributed data model. The GML specifications allow users to either use embedded objects or use references to external objects. What would be the effect on schema complexity if we force the use references instead of embedding data directly? Let us analyse this in the context of GML 3.1.1. Next, we compare the value of the metric C(XSD) for the original schemas against a modified version of the schemas in which only the *"linked style"* is supported. It must be noticed that forbidding the embedded style does not change the number of global components in the specifications, for this reason we have not used any of the metrics based on counting these items. The overall value of C(XSD) for the full schemas is 74,609 + 611R. The overall value for the schemas following the linked-style *(GML Linked Profile)* has been reduced to 18,731 + 13R, which considering that R=2 ,signifies a reduction of more than 75% of the complexity of the original schemas.

The scenario presented here is an extreme one, as all the embedded data has been removed from the schemas. In a real implementation other aspects apart from schemas complexity must be considered. In the case above, removing the chance of embedding data will increase the number of messages exchanged on the network as each piece of information must be requested separately by following the links provided by other documents. Nevertheless, this does not necessarily imply that we will have a higher network traffic, as we could follow these links only if needed, and we will not be obliged to read all of the information as if were embedded in other documents. In any case, in a real implementation, different trade-offs could be made about what to link and what to embed to satisfy all of the application requirements.

5.6 Concluding Remarks

In this chapter we have presented a quantitative way to analyse and measure the complexity of OWS schemas. The use of adequate metrics allows us to quantify the complexity and other properties of the schemas. The results of the analysis have shown that at least half of the presented specifications can be considered as large and complex according to all of the metrics included in our study. The metrics also allowed to group specifications according to their complexity.

The new metrics introduced here (DPR, DPF and SRR) have shown from different views the effect of the use of subtyping mechanism on complexity. For example, DPR has shown the fraction of polymorphic elements, frequently high, included in the schemas. DPF has considered how the possible polymorphic situations for these elements increase the effort needed to fully understand the schema components definitions. Last, SRR has shown that more than 60% of the schema components included in large specifications are referenced in ways that cannot be seen explicitly, augmenting the risk of making mistakes while working with the schemas.

We have also presented use case scenarios where metrics could be applied. Nevertheless, the metric set presented here should not be seen as a closed set, many other metrics can be useful in many different scenarios. Some of the potential uses of metrics are to evaluate the impact of design decisions, assessing the effectiveness of different solutions to deal with schemas complexity, and to detect potential design problems such as components with too many information items, or excessively deep subtyping hierarchies.

CHAPTER 6

Instance-based Schema Simplification

In the previous chapter we have seen that most of the OWS specifications schemas are large and this has a direct impact on the XML processing code generated using XML data binding tools. While developing an application, if the generated code does not meet its requirements, it must be written manually, which could be a daunting task judging for the size of the schemas. Although the application may not need all of the information contained in the schemas, due to their length and complex dependencies that exist between schema components, it is very hard to separate the parts we need from those we do not. This implies in many cases that developers partially or totally ignore the specification schemas, wasting all of the potential they offer to automate the production of XML processing code.

In our opinion, the impossibility of writing XML processing code for mobile devices in an easy and reliable way is probably one of the main causes of the low adoption of OWS clients in mobile environments. In this spite, we present in this dissertation a solution called *Instance-based XML Data Binding*. This solution consists in the automatic extraction

---

The algorithm presented in this chapter has been published with the title *"Dealing with large schema sets in mobile SOS-based applications"* in Proceedings of the *2nd International Conference on Computing for Geospatial Research & Applications (COM.Geo 2011)*, May 23-25, Washington, DC 2011. DOI: 10.1145/1999320.1999336

of the subsets of the specifications schemas used in a given application, and the use of such subset to generate XML processing code adapted to mobile device restrictions. Our main optimisation metric will be the size of code generated from the schemas, as it is one of the main obstacles to accommodate this code on OWS-based mobile applications. We also want the solution to generate code that is able to process as many XML documents as possible even if they contain typical validation errors. The main objectives must be accomplished without incurring in severe performance penalties regarding mostly execution time.

In this chapter we present an introduction to *Instance-based XML Data Binding*. After this, the remainder of the chapter focuses in the first step of the solution, which is concerned with extracting the subset of the schemas that are used on a given application. All the details regarding code generation for a specific platform and programming language will be presented in the next chapter.

6.1 Instance-based XML Data Binding

Instance-based XML data binding code generation, is a two-step process. The first step, *Instance-based schema simplification*, extracts the subset of the specification schemas that is used by the application, based on the assumption that a representative subset of XML instances that must be manipulated by the application is available. The second step, *Code generation*, consists on using all of the information extracted on the previous step to generate XML processing code as optimised as possible for a target platform.

We have divided the process in two steps because this way the results of the first step can be used in a platform-neutral way, meaning that the extracted schemas subset can be used to generate code using any other code generator available for any operating system or programming language. The second step produces code for a specific platform and programming language, Android and Java, respectively.

The whole process is shown in Figure 6.1, the inputs to the first step are the schemas for a given specification and a set of XML documents conforming to them that must be processed by the application. The outputs will be a subset of the schemas and other information about the use of certain features of the schemas that can be used to optimise the code in the following step. The outputs of the first step are the inputs of the code generation step. In this chapter we will focus only in *Instance-based Schema Simplification*.

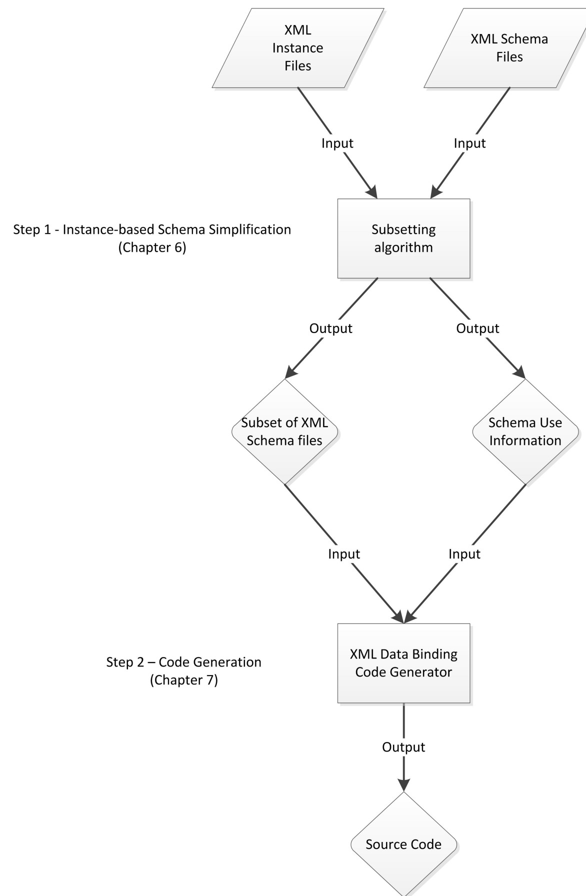

Figure 6.1: Instance-based XML data binding code generation process

6.1.1 Instance-based Schema Simplification

The aim of the *Instance-based Schema Simplification* step is to extract the subset of the schemas used on a set of XML documents. This information can be used later to produce more compact XML processing code (see Chapter 7). The idea behind this approach, which is more formally defined in subsequent sections is depicted graphically in Figure 6.2. Starting from a set of XML instances and the schema files defining their structure, we calculate which schema components are used and which are not. After this, we suppress all of the components that are no longer necessary.

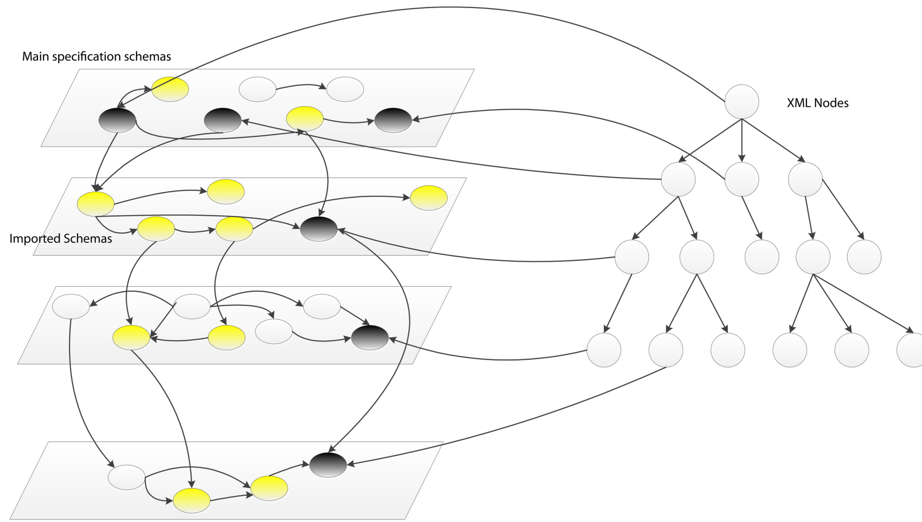

Figure 6.2: Relations between information items in XML documents (right) and schemas components defining its structure (left)

The figure shows to the left the graph of relationships between schema components. The different planes represent different specification schemas, with the main specification schemas at the top. Links between schema components represent dependencies between them. To the right we have the tree of information items (XML nodes) contained in XML documents. For the sake of simplicity we show only the tree of nodes corresponding to a single instance. An edge between an XML node and a schema component represents that the component describes the structure of the node. To simplify the figure we have only shown a few edges, although and edge for every XML node must exist.

In the following sections we present in detail how the simplification algorithm works, focusing in how the subset of the schemas is determined. The calculation of the second output, *Schema Use Information*, is not detailed here, it will be just mentioned as needed in the following chapter as this information is more closely related to code generation issues. The next section presents the necessary notation and concepts used later on. After this, Section 6.3 presents the algorithm in detail. Last, Section 6.4 presents experimental results of applying the algorithm to a real case study.

6.2 Notation

To refer to nodes contained in instance files, we will use *XPath* notation [W3C99b]. XPath expressions are shown in bold. The following examples referring to nodes in Listings 2.1 and 2.2 should suffice to understand the notation through the remainder of this chapter:

- **/Container** refers to the root node of both instances.
- **/Container/item** refers to all items contained in the root elements.
- **/Container/item[i]** refers to item in position i inside **/Container**. Positions are counted starting at 1.

To refer to components in schema files, we will use the following notation:

- To refer to global types and elements, we use its name in italics, e.g. *Container*, *ContainerType*, etc.
- To refer to attributes or elements within types, model groups or attribute groups, we add their name and a colon as prefix to the attribute or element name. The whole expression is written in italics. For example: *ContainerType:item*, *Base:baseElem*, *Child:chdElem*, etc.

For the purpose of our discussion we define the concept of schema set in the following way:

Definition 1: *An schema set* $S = (T_S, E_S, A_S, MG_S, AG_S, R_S)$*, where* T_S *is the set of all type definitions,* E_S *is the set of all element declarations,* A_S *is the set of all attribute declarations,* MG_S *is the set of all element group definitions,* AG_S *is the set of all attribute group definitions, and* R_S *is a set of binary relations (described later) between components of* T_S*,* E_S*,* A_S*,* MG_S*, and* AG_S*.*

Components included in sets T_S, MG_S and AG_S, are composed by a set of inner components. In the case of types, inner components can be references to global elements, attributes, model groups and attribute groups, or they can be nested element and attribute declarations. Model groups may contain references to global elements and other model groups, or they may contain nested element declarations. Similarly, attribute groups may contain references to other global attributes and attribute

groups, or they may contain nested attribute declarations. Inner components can be *optional*, meaning that it is legal that they do not appear in all valid instance documents. For example in Listing 2.3, element *baseElem2* in *Base* is optional; as such, items in Listings 2.1 and 2.2 are valid even when they do not contain this element.

The binary relations contained in R_S are:

- ***isOfType(x, t)***: relates an element or attribute x to its corresponding type t. For example: ***isOfType***(*Container*, *ContainerType*), ***isOfType***(*Base:baseElem*, *string*).
- ***reference(x, y)***: relates $x \in T_S \cup MG_S \cup AG_S$ to $y \in E_S \cup A_S \cup MG_S \cup AG_S$ if x references y in its definition using the *ref* attribute in any of its components, e.g. ***reference***(*Base, baseElem2*).
- ***contains(x, y)***: relates $x \in T_S \cup MG_S \cup AG_S$ to $y \in E_S \cup A_S$ if x defines y as an inner attribute or element in its declaration, e.g. ***contains***(*Base,Base:baseElem*), ***contains***(*Child,child:chdElem*), ***contains***(*Container*, *Container:item*).
- ***isDerivedFrom(t, b)***: relates a type t to its base type b, e.g. ***isDerivedFrom***(*Child*, *Base*)
- ***isInSubstitutionGroup(x, y)***: relates an element x to another element y if y is the head element of the x's substitution group.

The schema set S for the schema fragment in Listing 2.3 remains as follows[1]:

$$
\begin{aligned}
S = \{\ & T_S = \{\ \textit{Base, Child, string, ContainerType}\} \\
& E_S = \{\ \textit{Container, baseElem2, Base:baseElem,} \\
& \qquad\quad \textit{Child:chdElem, ContainerType:item}\} \\
& A_S = \emptyset \\
& MG_S = \emptyset \\
& AG_S = \emptyset \\
& R_S = \{\ \textit{isOfType} = \{\ \textit{(Container,ContainerType),} \\
& \qquad\qquad\qquad\qquad \textit{(baseElem2, string),} \\
& \qquad\qquad\qquad\qquad \textit{(Base:baseElem, string),} \\
& \qquad\qquad\qquad\qquad \textit{(Child:chdElem, string),} \\
& \qquad\qquad\qquad\qquad \textit{(ContainerType:item, Base)}\ \},
\end{aligned}
$$

[1] XML Schema *anyType* has been omitted purposely to simplify exposition.

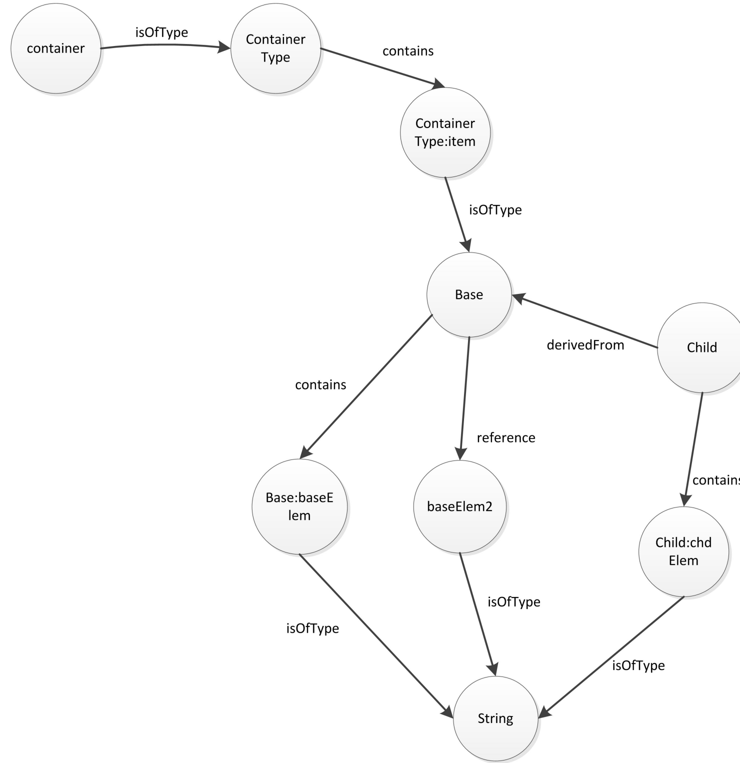

Figure 6.3: Graph of relations in schema fragment in Listing 2.3

$$\begin{aligned}
&\textit{isDerivedFrom} = \{\textit{(Child, Base)}\},\\
&\textit{reference} = \{\textit{(Base: Base:baseElem2)}\},\\
&\textit{contains} = \{\textit{(Base, Base:baseElem)},\\
&\qquad\qquad \textit{(Child, Child:chdElem)},\\
&\qquad\qquad \textit{(ContainerType,}\\
&\qquad\qquad\qquad \textit{ContainerType:item)}\}\\
&\textit{isInSubstitutionGroup} = \emptyset\}
\end{aligned}$$

Figure 6.3 shows a graph with all of these relations. This graph does not reflect hidden dependencies between types or elements. To include them in the graph we had to add an extra edge between *Container-Type:item* and *Child* as the former may be of type *Child* in instance files.

Next, we define the *subset* relation for schemas:

Definition 2: *Let S = (T_S, E_S, A_S, MG_S, AG_S, R_S) and S_1=(T_{S1}, E_{S1}, A_{S1}, MG_{S1}, AG_{S1}, R_{S1}), be two schema sets, we said that S_1 is a subset of S if $T_{S1} \subseteq T_S$, $E_{S1} \subseteq E_S$, $A_{S1} \subseteq A_S$, $MG_{S1} \subseteq MG_S$, $AG_{S1} \subseteq AG_S$, and for every relation R_{iS} in R_S , $R_{iS1} \subseteq R_{iS}$, for example,* $\boldsymbol{isTypeOf}_{RS1} \subseteq \boldsymbol{isTypeOf}_{RS}$

According to this definition a subset of the schema set shown above could be:

S = { T_S = *Base, string, ContainerType*
E_S = {*Container, Base:baseElem, ContainerType:item*}
$A_S = \emptyset$
$MG_S = \emptyset$
$AG_S = \emptyset$
R_S = { *isOfType* = { *(Container,ContainerType), (Base:baseElem, string), (ContainerType:item, Base)* },
isDerivedFrom = $\emptyset$,
reference = $\emptyset$,
contains = {*(Base, Base:baseElem), (ContainerType, ContainerType:item)*}
isInSubstitutionGroup = $\emptyset$}

Last, we define the *union* of two schema sets. This operation will be used in the following sections.

Definition 3: *Let S_1 = (T_{S1}, E_{S1}, A_{S1}, MG_{S1}, AG_{S1}, R_{S1}) and S_2=(T_{S2}, E_{S2}, A_{S2}, MG_{S2}, AG_{S2}, R_{S2}), be two schema sets, we said that S = (T_S, E_S, A_S, MG_S, AG_S, R_S) is the union of S_1 and S_2 if $T_S = T_{S1} \cup T_{S2}$, $E_S = E_{S1} \cup E_{S2}$, $MG_S = MG_{S1} \cup MG_{S2}$, $A_S = A_{S1} \cup A_{S2}$, $AG_S = AG_{S1} \cup AG_{S2}$ and $\forall R_{iS} \in R_s$, $R_{iS} = R_{iS1} \cup R_{iS2}$; e.g.* $isTypeOf_{RS} = isTypeOf_{RS1} \cup isTypeOf_{RS2}$

6.3 Simplification Algorithm

In practical terms our problem of simplifying the schema set related to a given specification Z, denoted as S_Z, to the subset that is used in an actual implementation P, denoted as S_P can be formulated as follows:

***Problem**: Calculate S_P starting from S_Z and X, a set of instance files, knowing that $X \subseteq I(S_P)$, trying to make S_P as small as possible. $I(S_P)$ is the set of all instances valid against S_P*

As the set of valid XML instances for a schema is potentially infinite, the resulting schema set should validate correctly all of the instances in X files, but might validate others as well.

6.3.1 Helper Functions

The algorithm to calculate S_P uses the following helper operations in its definition:

- ***typeOf***(*node*): returns the type of an XML node in an instance file. For example in Listing 2.1, ***typeOf***(**/Container**) = *ContainerType*, ***typeOf***(**/Container/item[1]**) = *Base*. In instance 2 *typeOf*(**/Container/item[1]**) = *Child*.
- ***element***(*node*): returns the element definition matching the content of *node*. For example, ***element***(**/Container/item[1]**) = *ContainerType:item* in both instances in Listings 2.1 and 2.2.
- ***containerOf***(*node*): returns the component containing the definition or reference to ***element***(*node*). For example, ***containerOf***(**/Container/item[1]**) = ***containerOf***(*ContainerType:item*) = *ContainerType* in Listings 2.1 and 2.2.
- ***ancestors***(*type*): returns all of the ancestors of *type*.
- ***leaf***(*node*) : returns *true* is node is a leaf, i.e. *node* does not contain any child element and has a value. Examples of leaf nodes in Listing 2.2 are **/Container /item[1]/baseElem** and **/Container/item[1]/chdElem**.
- ***root***(*instance file*): returns the root node of *instance file*.
- ***addValueToRelation***(*S*, *R(x,y)*): adds *R(x,y)* to the schema set *S*. *R* must be one of the relations defined in Section 6.2.

- ***copyRelations***(S_T, S_S, C): Copy all relation pairs between schema components in C, from the source schema set S_S to the target schema set S_T.

6.3.2 Algorithm

Algorithm 6.1 shows how S_P is calculated.

1: *Input:* $X = \{ x | x$ *input instance file* $\}$
2: *Input: schema set S*
3: *Output: schema subset* S_X *needed to validate instances in X*
4: $S_X = \emptyset$
5: **for** each x in X **do**
6:     *T =* ***SchemaSubsetUsedIn***(***root***(*x*), *S*)
7:     S_X *=* ***union***(S_X , *T*)
8: **end for**

Algorithm 6.1: Schema simplification algorithm

The key of the algorithm is the function ***SchemaSubsetUsedIn***(*node, schema set*) that calculates the subset of the schemas used in an XML file fragment starting at a given node (Algorithm 6.2). The second parameter is the schema set defining the fragment structure. The result of this function is calculated for the root element of all XML instances and then combined through the *union* operation defined in the previous section.

Algorithm 6.2 shows the details of ***SchemaSubsetUsedIn***. For the sake of clarity in the exposition of the algorithm we do not consider attributes and substitution groups. The code considering these cases is similar to processing element and subtypes, respectively.

The algorithm starts by adding the element definition matching the content of the node specified as input to the result (line 5). The type of the node is also added (line 6), as well as pair (***element***(*x*), ***typeOf***(***element***(*x*))) to relation ***isTypeOf*** (line 7). It is very important to notice at this point that ***typeOf***(*x*) and ***typeOf***(***element***(*x*)) are not always the same because the dynamic type of x may be a subtype of the declared type for the element matching its structure.

The next step is to analyse the child nodes in x, in case it has any (line 9). For each child node z, we call recursively the function ***SchemaSubsetUsedIn*** and the schema set returned by this function is combined with

1: *Input: XML node x*
2: *Input: schema set S*
3: *Output: schema subset S_X needed to validate instances in x*
4: $S_x = \emptyset$
5: $E_{Sx} = E_{Sx} + \textbf{element}(x)$
6: $T_{Sx} = T_{Sx} + \textbf{typeOf}(x) + \textbf{ancestors}(\textbf{typeOf}(x))$
7: **addValueToRelation**(S_x,**typeOf** (**element**(x), **typeOf**(**element**(x))))
8: **copyRelations**(S_x, S, **ancestors**(**typeOf**(x)))
9: **if** not **leaf**(x) **then**
10: **for** *each child node z of x* **do**
11: *S_x = union (S_x, **SchemaSubsetUsedIn**(z, S))*
12: *Container =**containerOf**(x)*
13: **if** *z belongs to a model group M* **then**
14: *$MG_{Sx} = MG_{Sx} + x$*
15: ***addValueToRelation**(S_x,**reference**(**containerOf**(M), M))*
16: *Container = M*
17: **end if**
18: **if** *z is reference to global element* **then**
19: **addValueToRelation**(S_x,**reference**(Container, **element**(z)))
20: **else**
21: **addValueToRelation**(S_x, **contains**(Container, **element**(z)))
22: **end if**
23: **end for**
24: **end if**
25: *Result =S_x*

Algorithm 6.2: ***SchemaSubsetUsedIn*** function

the current result using the *union* operation (line 11). After this, a set of relation values are added to maintain the consistency of the model. First, the container of x is calculated (line 12). The container is the type or model group that contains the element matching node x. This container could be ***typeOf**(x)* but could also be any of its ancestors. It also could be any model group referenced by ***typeOf**(x)* or any of its ancestors. For example, let us calculate ***containerOf**(* **/Container/item[1]/baseElem***)*

in Listing 2.2 presented in Chapter 2. Even when ***typeOf(*/Container/item[1]*)*** is type *Child*, this type does not contain the definition of **Container/item[1]/baseElem** because it was inherited from type *Base*.

The relation between ***element(z)*** and its container must be added to the result. The pair (***ContainerOf(element(z)),element(z)***) is added to ***reference*** or ***contains***, depending if the element is referenced or it is a nested declaration. In the case the container is a model group the reference between its own container and the model group must be added to the result as well.

6.4 Experimentation

With the purpose of proving the effectiveness of the algorithm we will use it to calculate the subset of the SOS schemas used in a case study. The case study is the implementation of the communication layer for a client for SOS targeted to the Android platform. The client must provide support for the *core profile* of the SOS specification, which includes the operations *GetCapabilities*, *DescribeSensor* and *GetObservation*.

On the server side we will use a 52° North SOS Server[2], containing information about air quality for the Valencian Community gathered by 57 control stations located in that area (Figure 6.4). The stations measure the level of different contaminants in the atmosphere.

To measure how much of the schemas can be reduced with the algorithm presented above, we compare the size of the original or *full schema set* with the size of the reduced or *simplified schema set*. To measure the size of a schema set S = (T_S, E_S, A_S, MG_S, AG_S, R_S), we calculate the cardinality of the first five sets conforming the schema set, and the cardinalities of every relation included in R_S.

After this, we use three different XML data binding generators to measure how much the generated binary files are reduced when using the simplified schema set. As our work is focused on the implementation of XML processing code, we just consider this part of the implementation code in the following subsections.

[2]http://52north.org/SensorWeb/sos/

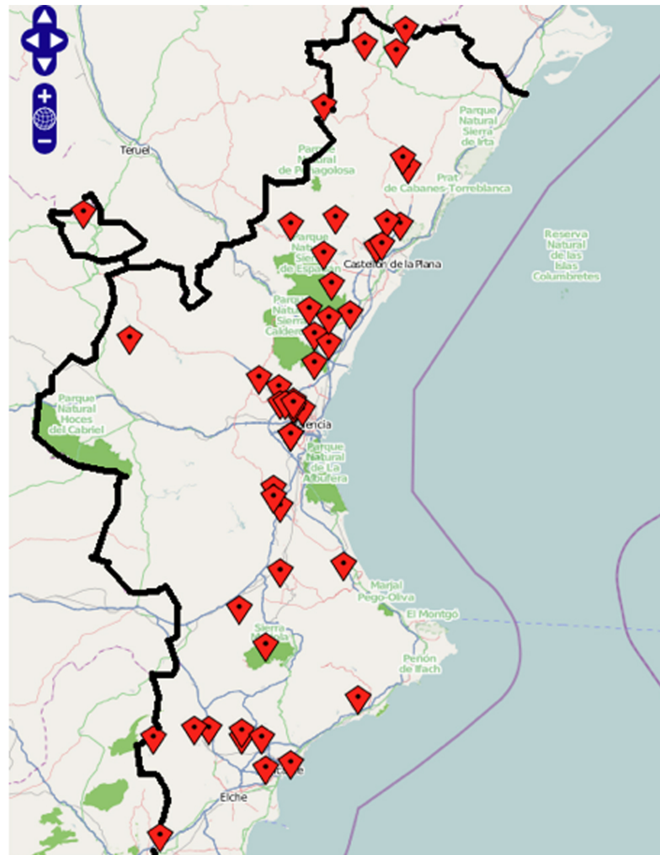

Figure 6.4: Location of air pollution control stations in the Valencian Community

6.4.1 Gathering Input Instance Files

In order to generate the schema subset needed for the SOS client we must decide which XML instances should be passed as input to the algorithm. To obtain these files a set of requests have been manually sent to the server and the server responses have been stored. As a result of this process, we gathered 2492 instance files as input: the capabilities

file, 2312 responses containing sensor descriptions, and 179 corresponding to observations[3]. Our application must be capable of processing the following root elements:

- *Capabilities*: Server response with the service capabilities file
- *SensorML*: Server response containing information about a sensor.
- *ObservationCollection*: Server response with observation data.

The first element is defined directly in the SOS specification and the other two are imported from the SensorML [OGC07c] and O&M [OGC07a] specifications, respectively. The number of files to be used as input will depend from the particular application being developed. It might depend on the availability of XML instances and how different the content of these files is.

6.4.2 Generating the Output Subset

After applying the algorithm with the inputs described above we obtained the results shown in Table 6.1 and Figure 6.5, where the original schema set is compared with the simplified set. In addition to cardinalities of components and relations we use two composite metrics: $Total_C$ for the summation of cardinalities of all components and $Total_R$ for the summation of cardinalities of relations. Results show that the new algorithm allows a substantial reduction of the original schema set of about 90% of its size.

6.4.3 Generating Binary Code

We explore next how this reduction is translated into generated code, using the generators presented in Chapter 2. As in Section 5.5.1 the main metric used to compare generated code is size measured in Kilobytes (KBs).

Source code is generated for the schemas before and after the simplification algorithm is applied. Then, the source is compiled and compressed into a JAR file. The size of the generated code is compared with and without considering the supporting libraries.

[3]In this case, a much smaller set of files would suffice as the structure of the XML documents of the same type were nearly identical

Table 6.1: Comparing original and simplified schema sets

Metric	Full Schema Set	Simplified Schema Set
$\|T_S\|$ (#T=#CT+#ST)	846	112
$\|E_S\|$ (#EL)	2,020	183
$\|A_S\|$ (#AT)	400	22
$\|MG_S\|$ (#MG)	28	7
$\|AG_S\|$ (#AG)	39	3
$\|isTypeOf_S\|$	2,420	205
$\|reference_S\|$	968	63
$\|contains_S\|$	739	81
$\|isDerivedFrom_S\|$	490	74
$\|isInSubstitutionGroups_S\|$	290	17
$\|Total_C\|$ (#GLOBAL)	3,333	327
$\|Total_R\|$	4,617	423

Table 6.2: Comparing size of code (KBs) for original and simplified schema sets

	XBinder	JAXB	XMLBeans
Full	3626	754	2822
Reduced	567	90	972
Full+libs	3816	1810	8879
Reduced+libs	684	1146	3655

Table 6.2 shows in the first two rows the comparison of the code size only for generated code (Full, reduced) showing a large reduction of between 79 and 88%. Next two rows compare code size including supporting libraries (Full+libs, reduced+libs). In this case the reduction is smaller as the size of the libraries remains constant. It ranges from a 37% reduction in JAXB to 84% in XBinder. In all cases the reduction of the generated code size is substantial. And the size of the code targeted to mobile devices (684 KB) seems like something that can be handled by modern devices.

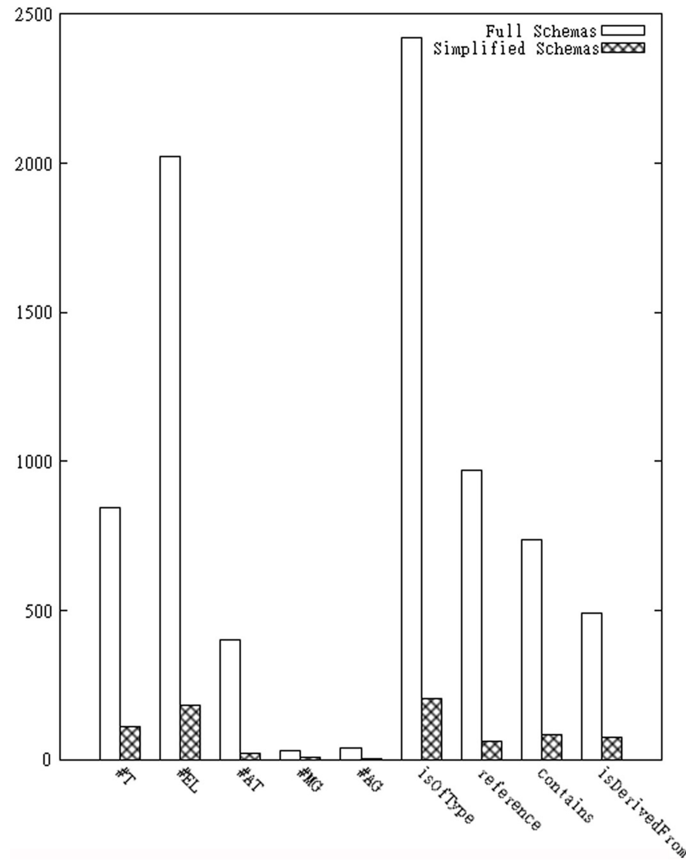

Figure 6.5: Simple schema metrics for original and simplified schemas

6.5 Concluding Remarks

In this chapter we have presented an algorithm to simplify large schema sets in an application-specific manner. The algorithm takes advantage of the fact that individual implementations use only portions of the schemas, which allows the simplification of large schema sets by using a set of XML instance files conforming to these schemas.

Results of applying the algorithm to a real-world use case scenario have shown that the algorithm allows a substantial reduction of the original schema set of about the 90% of its size. This huge reduction in schema size is translated into a reduction of generated binary code of more than 80% of its size for a SOS client targeted to the Android platform. As

the transformation is done at the schema level and no assumption about the target platform is made by the algorithm, it still can be used for other kind of SOS applications. Nevertheless, the resource constraints associated to mobile devices make the algorithm far more useful in this area. The algorithm could be also applied to other OWS specifications although based on the little experience we have with other specifications besides SOS, we cannot state that the reduction could be as large as that obtained in the use case scenario presented here.

CHAPTER 7

XML Data Binding for Mobile Devices

As mentioned before in this document, *Instance-based XML data binding code generation*, is a two-step process. The first step, *Instance-based schema simplification*, has been presented in the previous chapter. The second step, involving the generation of code for a mobile platform is presented in this chapter.

Generating code for mobile devices poses additional complications to XML Data Binding code generators. The constraints related to memory, processing and battery life has not made possible that existing generators

---

A short version of the content of this chapter with the title *"XML Data Binding for Mobile Applications Based on Large XML Schemas"* has been accepted to the *Third International Workshop Middleware for Pervasive Mobile and Embedded Computing*, M-MPAC 2011, (A workshop of the ACM/IFIP/USENIX 12th International Middleware Conference), December 12, Lisbon, Portugal.

An extended version of the sample application in Section 7.5.1 with the title *"Building Compact Standard-Based Geoprocessing Mobile Clients"* has been submitted to *The Fourth International Conference on Advanced Geographic Information Systems, Applications, and Services (GEOProcessing 2012).*

An extended version of the sample application in Section 7.5.2 has been published with the title *"Sensor Observation Service Client for Android Mobile Devices"* in *Proceedings of Workshop on Sensor Web Enablement 2011 (SWE2011)*, October 6-7, Banff, Alberta, Canada.

for desktop or server applications could be easily adapted to these devices. As a consequence, the availability of generators for mobile devices is much more limited than for other environments.

In our case, we will generate code for the Java programming language and the Android mobile platform. These choices have been made based on the availability of mature tools to implement a prototype which will be used to build some sample applications and to be readily applied to ongoing projects with actual requirements. Although the number of generators and development tools per mobile platform and programming language may vary, the techniques and conclusions drawn in this chapter are applicable to most platforms.

7.1 XML Data Binding Code Generator

Even when using the schema simplification algorithm may largely reduce the size of code generated with different XML data binding generators, the process can still be improved using other information gathered during the analysis of XML instances. For this reason, we decided to build our own generator to take advantage of this information. This information was labelled as *Schema Use Information* in Chapter 6 (Figure 6.1). It will contain a record of the usage of schema characteristics such as type and element substitutions, occurrence constraints, etc.

Figure 7.1 shows a more detailed view of the code generation process (Step 2 in Figure 6.1). The outputs of the schema simplification step are used as inputs to the *schema processor*, the component of the generator in charge of creating the data model that will be used later by the *template engine*. The *template engine* combines pre-existing *class templates* with the data model to generate the final source code. The use of a template engine allows the generation of code for other platforms and programming languages by just defining new class templates.

7.1.1 Supported Features

The code generator presented here presents the following features that contribute to the generation of optimised code. All of these features will be explained in detail later on this chapter:

- *Support for instance-based code generation*: The first step of the instance-based XML data binding process, in addition to calculating the subset of the schemas used, also gather other information that

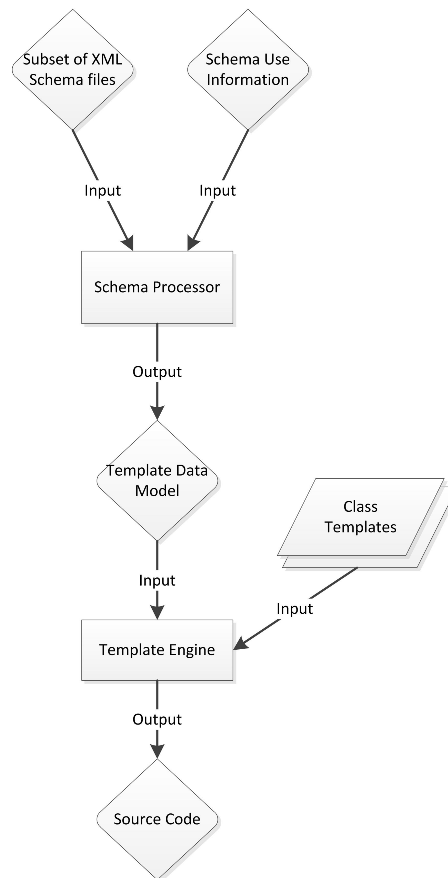

Figure 7.1: Flow diagram for the code generation process

allow customisations to be applied to the generated code such as those listed next:

- *Efficient handling of subtyping and wildcards*: Using the information contained in XML documents we can bound the number of entities that may replace types and substitution groups head elements in valid documents. Something similar can be done with substitutions of wildcards.
- *Inheritance flattening*: Geospatial schemas typically present deep subtyping hierarchies and by flattening these hierarchies we can reduce the number of classes in the generated code.
- *Adjust occurrence constraints:* Occurrence constraints define

the number of occurrences that are valid for an element. As the schemas are built to be applied to many different scenarios, they must be designed to be as general as possible. In actual applications, depending on its use of XML documents, occurrence constraints can be strengthened to generate more optimised code.

- *Source code based on simple code patterns*: The generated source code is straightforward to understand and modify in case it is necessary.
- *Tolerate common validation errors:* Occasionally, XML documents that are not valid against their respective schemas must be processed by our applications. In many cases, the validations errors can be ignored following simple coding rules.
- *Collapse elements containing single child elements*: Information items that will always contain single elements can be replaced directly by its content.
- *Disabling parsing/serialization operations as needed*: Some code generators always includes code for parsing and serialization even when only one of these functions is needed.
- *Ignoring sections of XML documents*: Frequently, we are not interested in all of the information contained in XML files, ignoring the unneeded portions of the file will reduce the memory and processing requirements of the applications

Except for the first group of features the rest has been implemented in existing code generators, although not frequently they can be found all together in the same generator.

7.2 Basic Mapping of Schema Components

In this section we explain how the schema processor maps schema components to programming language constructs. The basis of this mapping is very simple, containing rules for mapping complex types, simple types and global elements. To simplify exposition we say that given s, an schema component, *T(s)* will be the corresponding programming language construct generated from s.

7.2.1 Mapping Complex Types

Each complex type in the schemas is mapped to a class in the target programming language (Java in our case), i.e. if s is a complex type, $T(s)$ will be a Java class representing s. An example can be seen in Listing 7.1, where a portion of the generated code for the complex type *Child* (Listing 2.3) is shown[1]. The mapping is performed by applying the following rules:

1. Attribute references and declarations are transformed into fields of the class. The type of each field, which is always a simple type, is determined according to the rules for mapping simple types.

2. Element references and declarations are transformed into fields of the class. The type of each field, which in this case can be a simple or complex type, depends on whether the type of the element in the schemas t_e has a counterpart in the generated code, or is mapped to a primitive Java type. It also will depend on the element occurrence constraints. If the occurrence constraints of the element allow that instances contain at most one occurrence, the type of the element will be $T(t_e)$. If multiple occurrences are accepted it will be mapped to $List < T(t_e) >$.

3. Setter and getter methods are generated for each field

4. All classes will inherit directly or indirectly from *XMLInstanceTag*, a base class containing the common structure and behaviour for all the mapped classes.

5. An overridden version of the method *processTag* defined in *XMLInstanceTag* is generated for the class. This method will contain the code needed to parse the content of the XML node containing the information to be processed.

6. A method named *processAttributes* is generated for the class. This method will contain the code needed to parse the attributes of the XML node.

In Listing 7.1 we can see that the *processTag* and *processAttributes* have a parameter of type *KXmlParser*, a pull parser included in the open source library kXML[2]. This library is an implementation of the *XMLPull*

[1] The exceptions thrown from class methods have been omitted in this and the following code listings in this chapter

[2] `http://kxml.sourceforge.net/kxml2/`

Listing 7.1: Java class corresponding to complex type Child

```
public class Child extends XMLInstanceTag{

    private String baseElement;
    private String baseElement2;
    private String chdElement;

    public Child(int tagCode) {
        super(tagCode, true);
    }

    public String getBaseElement() {
        return baseElement;
    }

    public void setBaseElement(String baseElement) {
        this.baseElement = baseElement;
    }

    public String getBaseElement2() {
        return baseElement2;
    }

    public void setBaseElement2(String baseElement2) {
        this.baseElement2 = baseElement2;
    }

    public String getChdElement() {
        return chdElement;
    }

    public void setChdElement(String chdElement) {
        this.chdElement = chdElement;
    }

    @Override
    protected boolean processTag(KXmlParser parser, boolean ignore){
        // method body omitted
    }

    @Override
    protected void processAttributes(KXmlParser parser){
        // method body omitted
    }
}
```

API targeted to be used mostly in resource constrained devices such as mobile phones. The size of the version used in our implementation has a size of only 10.3 KB.

XMLInstanceTag

XMLInstanceTag is the base class for all of the types produced during the generation process. It provides the basic mechanisms to read the content of a node in an XML document. Listing 7.2 shows an extract of the code of this class.

The class constructor receives the tag code of the XML node to parse and a boolean value as parameter. The tag code, generated from the node names, is necessary because Java classes correspond to types in the schemas, and as such nodes with different names may have the same type in XML documents. This tag code is used by the method *fromXML* of the class to determine when it reaches the end of the node being processed. The boolean value defines if the node may have children elements or not.

The class also contains the methods *fromXML* and *toXML* for parsing and serialization respectively. *fromXML* reads the content of the current XML node during the parsing process. It first reads the content of attributes, by calling *processAtributtes*, method that must be overridden by classes extending *XMLInstanceTag*. After this, *fromXML* depending whether the node may have children nodes or not, iterates through the nodes processing them as they are reached, or just reads the corresponding value. Children nodes are processed inside the method *processTag* that must be overridden by subclasses as well. Similarly, node values are read using the method *processValue*. The method *fromXML* is an implementation of the *Template Method* design pattern [GHJV95].

Processing Children Nodes

The general structure of the code to process children of the current node is shown in Listing 7.3. First, the method gets the tag code for the child node to be processed. The method *NameResolver.getNextTagCode* also checks if the dynamic type of the node is different from its declared type. After this, an entry (a *case* statement) for each child node type that must be processed exists inside a *switch* statement.

The nature of the entries will vary depending on the type of the fields that are used to store the child node information. Listing 7.4 shows the general forms of these entries for child nodes that will be stored as single objects and as a list of objects. The notation <*value*> is used to denote the parts of the code that vary according to the context. The first case statement shows how a child node that must be read into a complex type field is processed. The code shows the most complex scenario where *dynamic typing* must be supported. Depending on whether a dynamic

Listing 7.2: Common base class for all types

```
public abstract class XMLInstanceTag{
  protected int tagCode;
  protected boolean hasChildren;

  protected XMLInstanceTag(int tagCode, boolean hasChildren){
    this.tagCode = tagCode;
    this.hasChildren = hasChildren;
  }

  public void fromXML(KXmlParser parser){
    boolean end = false;
    processAttributes(parser);

    if (hasChildren){
      int currentTag = parser.nextTag();
      while (!end){
        switch(currentTag){
         case KXmlParser.START_TAG:
          processTag(parser);
          break;
         case KXmlParser.END_TAG:
          end = (NameResolver.getNextTagCode(parser)==tagCode);
          break;
        }
        if (!end)
          currentTag = parser.next();
      }
    }
    else
      processValue(KXmlParser parser);
  }

  public String toXML() {
    throw new NotImplementedException();
  }

  protected abstract void processAttributes(KXmlParser parser);
  protected abstract boolean processTag(KXmlParser parser);
  protected abstract boolean processValue(KXmlParser parser);
}
```

type is present or not, a different object type is created. After this, the content of the node is read by calling the method *fromXML*. The second case statement is similar but as the information to be read is an object in a list and not the field itself, it must be read in an auxiliary variable and then inserted in the list.

It is worth noticing that when an element type is replaced by a subtype

Listing 7.3: *ProcessTag* method definition for class *Child*

```
@Override
protected boolean processTag(KXmlParser parser){
  boolean m_localResult = false;
  int nextTagCode = NameResolver.getNextTagCode(parser);

  switch(nextTagCode){
    // Case entries to process children nodes
  }
  return m_localResult;
}
```

through the use of attribute *xsi:type*, the name of the node is the same whether it has a type or another. As a consequence, the tag code will be the same as well. For this reason, the same case statement handles both situations. When substitution groups are used, the head element and the elements in its substitution group do not have the same names (nor tag code), hence different case statements are generated for possible substitutions.

Other cases that must be handled such as fields of primitives types are much more simpler than the ones presented here.

Processing Node Attributes

Processing the attributes of a node is very simple, we just have to override the *processAttributtes* method inherited from *XMLInstanceTag* as is shown in Listing 7.5. Method *getAttributeValue* is defined as a protected method in *XMLInstanceTag*.

Listing 7.5: Processing code for node attributes

```
@Override
protected void processAttributes(KXmlParser parser){
  // Read attribute values
  <field name 1> = getAttributeValue(parser, <Attribute Code 1>);
  <field name 2> = getAttributeValue(parser, <Attribute Code 2>);
  ...
}
```

7.2.2 Mapping Simple Types

Simple types in the schemas will be mapped whenever possible to primitive or predefined Java types. As a general rule, facets that constrain

Listing 7.4: Entry types for single field and list complex types in *ProcessTag*

```
// Complex type single field with dynamic type support
case <Element Code>:
  if (NameResolver.hasDynamicType())
    <field> = TypeFactory.getObjectForType(
                  NameResolver.getXSIType());
  else
    <field> = new <ObjectType>();

  <field>.fromXML(parser);
  m_localResult = true;
  break;
}

// Complex type list field with dynamic type support
case <Element Code>:
  <Object Type> auxValue;
  if (NameResolver.hasDynamicType())
    auxValue = TypeFactory.getObjectForType(
                  NameResolver.getXSIType());
  else
    auxValue = new <ObjectType>();

  auxValue.fromXML(parser);
  <field>.add(auxValue);
  m_localResult = true;
  break;
}
```

the values of schema predefined types will be ignored to speed up parsing of XML documents. This allows mapping most of the simple types defined in the schemas without having to create new types in the generated code.

The only exception will be when a simple type is declared as the union of several types that cannot all be mapped to the same Java type. In that case, a Java class is created with a field for each possible type of the contained values and boolean flags indicating which of the values are set.

7.2.3 Mapping Global Elements

By default, global elements will not be mapped to any programming language construct unless it is explicitly specified that they can act as root of XML documents. In that case, a parser class is created with methods to parse the instances for files or streams. An example is shown

Listing 7.6: Parser class generated for element *Capabilities*

```
public class CapabilitiesParser{

  public static Element_Capabilities parse(InputStream is){

    Element_Capabilities serviceDescriptor;
    // Create namespace-aware parser
    KXmlParser parser = new KXmlParser();
    parser.setFeature(KXmlParser.FEATURE_PROCESS_NAMESPACES, true);

    try
    {
      //Assign input stream to parser and read the first tag
      parser.setInput(is, encoding);
      parser.nextTag();
      int tagName = NameResolver.getNextTagCode(parser);

      if (tagName == Constants.SOS_CAPABILITIES){
        // Parse XML document
        serviceDescriptor =
              new Element_Capabilities(Constants.SOS_CAPABILITIES);
        serviceDescriptor.fromXML(parser);
      } else
        throw new SOSException(``Parsing failed with start tag: ''
                               + tagName);
    }
    catch(Exception e){
      //Handle exception
    }
    return serviceDescriptor;
  }

  public static Element_Capabilities parse(KXmlParser parser){
      // method body omitted
  }

  public static Element_Capabilities parse(File f){
      // method body omitted
  }
}
```

in Listing 7.6, where a fragment of the class to parse SOS capabilities files is shown.

7.3 Supported Features Explained

In this section we explain in greater detail the features supported by the code generator. We spend most of the time explaining features

related to instance-based code generation because to our best knowledge they have not been implemented previously in available code generators.

7.3.1 Support for Instance-based Code Generation

Instance-based code generation refers to the use of information extracted from XML documents to improve the generated code according to some criteria. These documents contain valuable information that can be very useful while generating code. The nature of this information and how it is used to optimise the code produced by our generator is shown in the following subsections.

Efficient Handling of Subtyping and Wildcards

The handling of subtyping and wildcards can be a complicated issue. As mentioned in Chapter 2, the *dynamic type* of elements may differ from its *declared type*, because of the substitution of a type for a subtype or the substitution of the head element of a substitution group.

In the general case, where no instance-based information is available, generic code to face any possible type or element substitution must be written, as was shown in Listing 7.4 for the case of type substitution. In Section 7.2.1, was also pointed out that a case statement is generated for each possible element substitution. Nevertheless, this scenario can be substantially simplified with instance-based information. Let us consider for example the case of complex type *gml:FeaturePropertyType* in the context of the SOS schemas. This type is used as a container for any feature and contains a reference to a global element *gml:_Feature* (Listing 7.7). *gml:_Feature* is the head element of the substitution group shown in Figure 7.2, as such, the source code for *gml:FeaturePropertyType* must be ready to parse any of these elements. If instead of generating code for the full SOS schemas, we consider the subset of the schemas needed for the case study introduced in Chapter 6, we reduce the number of elements in the substitution group to those in Figure 7.3. Even more, we can determine for every specific type which referenced elements are replaced by which element in its substitution group. In the example above we can figure out that class *FeaturePropertyType* only must be aware of parsing elements of type *FeatureCollection* and *SamplingPoint*, which makes the final code a lot simpler.

Similarly we can determine which inner elements of a type may have a dynamic type different from its declared type. If dynamic typing is used

Listing 7.7: Extract of *feature.xsd* containing the definition of *gml:FeaturePropertyType*

```
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://www.opengis.net/gml">

  <element name="_Feature" type="gml:AbstractFeatureType"
           abstract="true" substitutionGroup="gml:_GML"/>

  <complexType name="AbstractFeatureType" abstract="true">
    <complexContent>
      <extension base="gml:AbstractGMLType">
        <sequence>
          <element ref="gml:boundedBy" minOccurs="0"/>
          <element ref="gml:location" minOccurs="0"/>
        </sequence>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="FeaturePropertyType">
    <sequence minOccurs="0">
      <element ref="gml:_Feature"/>
    </sequence>
    <attributeGroup ref="gml:AssociationAttributeGroup"/>
  </complexType>
</schema>
```

in the XML documents we generate generic code such as that in Listing 7.4, but if it is not used we can generate simpler code. In addition, if dynamic typing must be supported the information about which types can be replaced and which subtypes are used in every case is used to optimise the implementation of the function *TypeFactory.getObjectForType*.

Last, the same technique is applied to wildcards and elements of type *anyType*, during the execution of the instance-based schema simplification algorithm, it recognises which elements are used to replace wildcards or *anyType* elements. This information is used by the generator to generate the appropriate code to handle valid substitutions.

Inheritance Flattening

The domain model defined by OWS schemas contains very deep type hierarchies. For example, Figure 7.4 shows the type definition hierarchy of GML 3.1.1. A *type definition hierarchy* is a tree including all of the derivation relationships between types [W3C04c]. If this specification were analysed in the context of SOS the hierarchy would be even bigger,

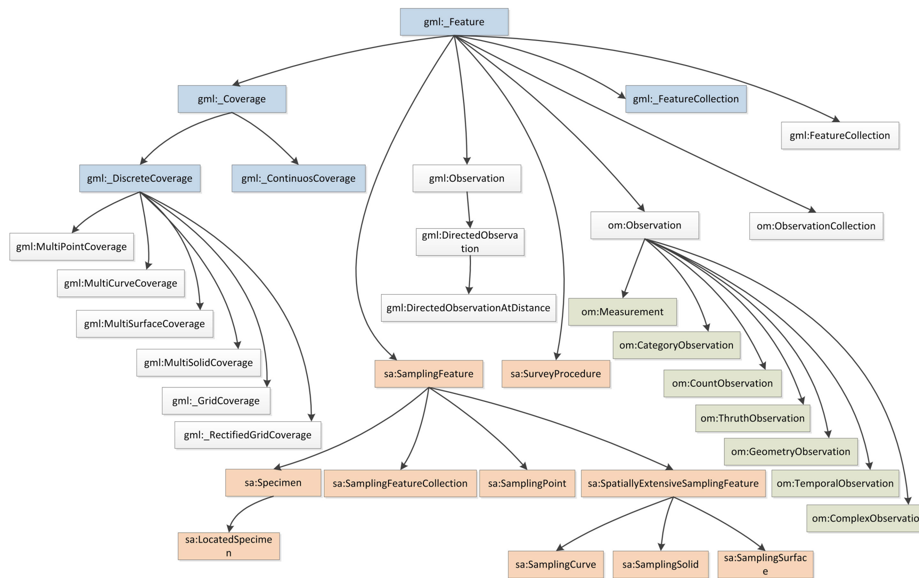

Figure 7.2: Elements in the substitution group of *gml:_Feature*

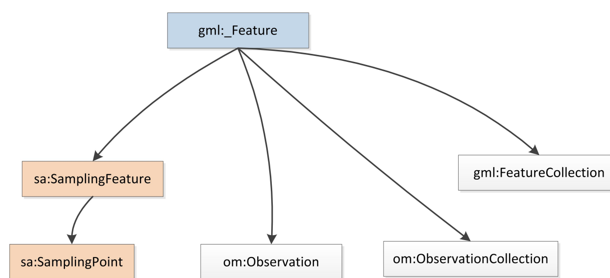

Figure 7.3: Elements in the substitution group of *gml:_Feature* for the a subset of the SOS schemas

as SOS uses several GML types as base types for locally defined types. Type hierarchies in OWS schemas contain many abstract classes, which by definition cannot be instantiated in XML documents.

Our code generator optionally allows to *"flatten"* the inheritance tree, eliminating all of the classes in the tree that are not instantiated in XML documents. Each class is transformed by adding all of the inherited fields and methods to the class declaration. The flattening process makes the number of generated classes smaller, although this does not imply that the final size will also be smaller. The reason for this is that elements and attributes included in base types eliminated during flattening must be replicated as fields in the generated code in all of the subtypes of these types. Depending on whether the number of these fields is high or low the size of the generated code can have a bigger or smaller size than the original one. The effect of a high number of replicated fields may be important if we consider that type hierarchies in the geospatial schemas tend to be deep. Anyway, the effect of inheritance flattening in general will tend to be beneficial as less classes must be loaded by the virtual machine to start execution.

One of the disadvantages of this technique is that all of the information related with subtyping between generated classes is lost. As this information might want to be kept, this option can be enabled and disabled as needed before performing code generation.

The technique of inheritance flattening has been widely explored and used in different computer science and engineering fields as is proven but the literature found on the topic [BLS00, CRC+06, BEMW08, CREP08, LSZ09, BEWZ10].

Adjust Occurrence Constraints

As explained in Section 7.2.1 the occurrence constraints of an element determine if it will be mapped to single object instances or to lists. If the element can have multiple occurrences it will be mapped to a list, otherwise it will be mapped to a single instance variable. In XML documents, many fields that are declared in schemas as having possibly multiple occurrences, have only a single occurrence. In this case, they can be safely mapped to a unique instance variable instead of a list. This way the final code will make a better use of memory during execution as instead of creating a list that will only contain a single object, it will create a single object instance.

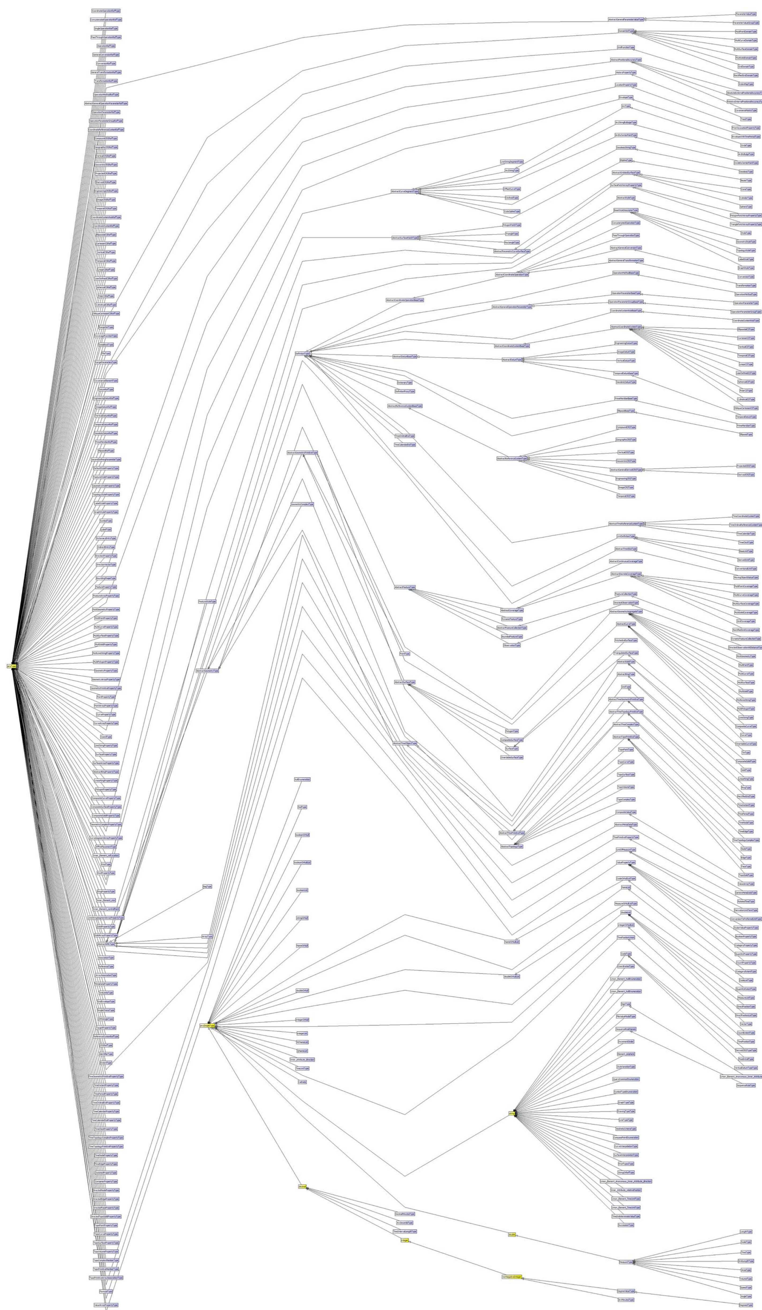

Figure 7.4: Type definition hierarchy for GML 3.1.1

7.3.2 Source Code Based on Simple Code Patterns

XML data binding code generators relieve the work of developers producing substantial parts of XML processing code. Unfortunately, occasionally the generated code must be manually modified because of different reasons, such as the generator does not support a certain schema feature, or it is not capable of mapping correctly certain situations to the target programming language[3]. In these cases, a valid alternative is to modify the generated code to solve any conflict that impedes the correct execution of the final code.

In order to allow the generated code to be modified easily when needed it must be based on simple code patterns, which is not the case for most code generators. In our case, the generated code is straightforward to use or modify. As shown in previous sections the complexity of the code, compared with the code generated by other tools, is not high and it can be assimilated by an average Java programmer in a short period of time.

7.3.3 Tolerate Common Validation Errors

As will be shown in Chapter 8, XML documents that must be processed in our applications occasionally contains validation errors. Ignoring these invalid documents and informing of this situation to be user is not always an option as users may need to access this information to do their jobs. For this reason, it is advisable for XML processing code to be ready to handle these situations and if possible, to continue normal execution of the application.

Most of the errors that are commonly found can be overcome by using simple coding rules. For example, mapping most simple types to Java primitives or predefined types, and ignoring facets constricting type values, will allow that values for elements and attributes of these types can be parsed despite possible validation errors. The use of a *switch* statement in the *processTag* method of every generated class implies that the order of the children nodes is not checked so misplaced elements will be successfully parsed. In addition, the absence of a node that according to the schemas is mandatory will not impede the document of being processed.

The general approach is to shift to upper layers the responsibility of determining if errors found in the data will affect the correct functioning of the whole application. This approach has as advantage that a larger

[3] We had these problems in the past while using XMLBeans and XBinder with geospatial schemas

number of documents will be successfully parsed and the parsing process will be faster. On the downside, the application must include extra code to validate the correctness of certain data values. The real effect of this disadvantage may vary depending on applications requirements. For example, if a client application gets data from a trusted source it will not have to double-check the data and the lack of strong validation in the XML processing code will have no negative impact. If the source is not trusted and the applications well-functioning relies in a strong adjustment of data to the structure defined on its schemas, validation code will have to be added to components using the generated XML processing code, which of course will augment development time and software costs.

7.3.4 Collapse Elements Containing Single Elements

A schema coding pattern commonly used on OWS schemas is the *object-property model* [LBT04]. This pattern defines object properties by encapsulating the actual value of a property in an extra element which usually explicitly indicates in its name that it is a property. These elements mostly contain a single child. Although very useful for clarity purposes these extra elements usually results in a higher number of classes in the generated code that may increase the size of the binary code and the amount of memory that must be used by the application.

Let us consider the example of the type *sos:ObservationOfferingType*[4] defined in the SOS schemas. An extract of the type definition is shown in Listing 7.8. The fragment of the type contains two children elements, *gml:boundedBy* and *time*, with types *gml:BoundingShapeType* and *swe:TimeGeometricPrimitivePropertyType* respectively. These types are mere containers for *gml:Envelope* objects and time objects (*gml:TimePeriod* and *gml:TimeInstant*). In the case study presented in the previous chapter the only time object used is *gml:TimePeriod*. If we make a straightforward mapping of complex types in schemas to Java classes at least six of these classes should be created: *ObservationOfferingType*, *BoundingShapeType*, *TimeGeometriPrimitivePropertyType*, *EnvelopeType*, *TimePeriodType* and *TimeInstantType*. But in practice, only three of them are really needed to get the job done: *ObservationOfferingType*, *EnvelopeType* and *TimePeriodType*. Types used only as containers can be eliminated from the generated code as well as unused types.

[4]Prefixes *sos* and *swe* refer here and in the remainder of this

Listing 7.8: Extract of *sosContents.xsd* containing the definition of *sos:ObservationOfferingType*

```
<complexType abstract="true" name="ObservationOfferingBaseType">
    <complexContent>
      <restriction base="gml:AbstractFeatureType">
        <sequence>
          <!-- Group reference omitted -->
          <element ref="gml:boundedBy"/>
        </sequence>
      </restriction>
    </complexContent>
  </complexType>

  <complexType name="ObservationOfferingType">
    <complexContent>
      <extension base="sos:ObservationOfferingBaseType">
        <sequence>
          <element name="time"
            type="swe:TimeGeometricPrimitivePropertyType"/>
          <!-- The rest of the elements have been omitted -->
        </sequence>
      </extension>
    </complexContent>
  </complexType>
</schema>
```

7.3.5 Disabling Parsing/Serialization Operations

Most generators produce code including parsing as well as serialization code. In different scenarios the importance of the role of these tasks varies greatly. For example, in the context of the case study presented in Chapter 6, parsing is a task much more important than serialization. Serialization would be needed to generate requests sent to the server in XML format, which in all cases had a size below 1KB. In addition, some of the request were sent using HTTP GET, which does not require XML serialization at all. On the other hand, the server responses varied from very small responses(<1KB) to responses with a size of about 1.5 MB. In this scenario, the classes in charge of processing server responses do not need serialization code to be generated, which will make the size of the generated code much smaller.

document to the namespaces *http://www.opengis.net/sos/1.0* and *http://www.opengis.net/swe/1.0.1*, respectively

7.3.6 Ignoring Sections of XML Documents

Not always our applications need to process all of the information included in XML documents. Ignoring the portions of the document that we do not need will improve the speed of the parsing process and it may have a significant impact in the amount of memory used by our application to store the data read from these documents. By default if a child of the current node is not explicitly handled in the *processTag* method it is ignored, but if this child node contains other children node the generated code will try to process them. If we want to completely ignore a child node and all of its content we can use class *IgnoredTag* to *"process"* the node. This class ignores the whole subtree of nodes starting at a given node.

7.4 Experimentation

In this section we present a simple experiment to measure if the current implementation of the generator fulfil its main goal which is to reduce the size of the generated code as much as possible. We present first a short introduction to the current implementation of the XML data binding code generator, called *DBMobileGen* (DBMG for short). After this, we analyse the size of the code produced by our generator for the case study presented in Chapter 6.

7.4.1 DBMobileGen

DBMobileGen is the current implementation of the *Instance-based XML Data Binding Approach*. It includes components implementing both the simplification algorithm and code generation process. It is implemented in Java and relies on existing libraries such as *Eclipse XSD*[5] for processing XML schemas, *Freemarker*[6] as template engine library, and as well as the generated code, *KXml* for low-level XML processing. Implementation details beyond what has been presented so far are not included in this document as in our opinion they do not help to understand the approach presented here or the process of measuring how effective it can be.

[5] `http://www.eclipse.org/modeling/mdt/?project=xsd#xsd`

[6] `http://freemarker.sourceforge.net`

The current implementation has some limitations. Because of the complexity of OWS specifications and the XML Schema language itself, support for certain features and operations have been only included if it is considered necessary for the case studies and sample applications considered in these document. Some of these limitations are listed next:

- *Serialization is not supported yet*: The role of parsing for our sample applications and case studies is far more important than serialization. This is mainly because request issued using HTTP GET do not need serialization at all, and the size of the XML payload in HTTP POST request use to be small and with a relatively simple structure.
- *Conflicts with the extension by restriction mechanism*: The simplification algorithm may produce reduced versions of complex types that are not valid XML Schema definitions in the presence of subtypes hierarchies using extension by restriction.
- *Dynamic typing using xsi:type not fully supported*: The mechanism described in section 7.2.1 for dealing with dynamic type substitution has not been fully implemented yet, as the XML documents processed in the case studies do not make use of this feature. Generally, the substitution group mechanism is preferred for dynamic typing scenarios.

7.4.2 Experiment Description

In section 6.4, a case study for an SOS client was presented. Within the context of this case study we analysed how much the schemas can be simplified and the impact of this simplification in the size of the code produced by various generators. In this section, we compare those results with the code generated by *DBmobileGen* for the same scenario. Here, we use *DBMobileGen* to generate code from the full SOS schemas and the simplified schemas. For both cases, we generate code with all of the optimisations enabled, and with all of the optimisations disabled (inheritance flattening, collapsing single child elements, etc.).

7.4.3 Results

We replicate here Table 6.2 adding the size of the code generated by DBMG. The results are also shown in Figures 7.5 and 7.6 and detailed next.

Table 7.1: Comparing size of code (KBs) for original and simplified schema sets

	XBinder	JAXB	XMLBeans	DBMG (Opt. Off)	DBMG (Opt. On)
Full	3626	754	2822	1251	88
Reduced	567	90	972	138	88
Full+libs	3816	1810	8879	1281	118
Reduced+libs	684	1146	3655	168	118

Figure 7.5 shows the code generated from the full SOS schemas. In the first group of columns we can observe that when optimisations are not used the code generated by DBMG is larger than JAXB, but still shorter than XBinder and XMLBeans. It must be noted that comparing this code with that generated by XBinder is not completely fair as serialisation is not still implemented[7]. JAXB has the smaller memory footprint for the generated code, except for DBMG with optimisations enabled, because as pointed out in the previous chapter it does not include processing code in the generated code. Nevertheless, if the supporting library were included in the comparison the advantage of JAXB regarding total size of the code vanishes as the supporting libraries are much bigger than those for DBMG (1056 KB against 30 KB). Again, it must be considered that if serialisation code for our generator were included the combined size will be probably slightly bigger than that of JAXB. The code generated by *DBMG* with all optimisations enabled is fairly smaller than the rest because it has built-in support for the schema simplification algorithm.

Figure 7.6 shows the code generated from the simplified schemas. Again, the code generated with DBMG with all of the optimisations enabled is the smallest. Although the code for JAXB, without considerings its supporting libraries is about the same size.

The size difference for the code generated for Android devices is significant. XBinder code is about six times larger than the one generated

[7] We roughly estimate an increase of about 30% of the code size when serialisation code is included. XBinder allows to choose whether parsing or serialization code is included or not when code is generated for C++. Unfortunately this option is not available for Java.

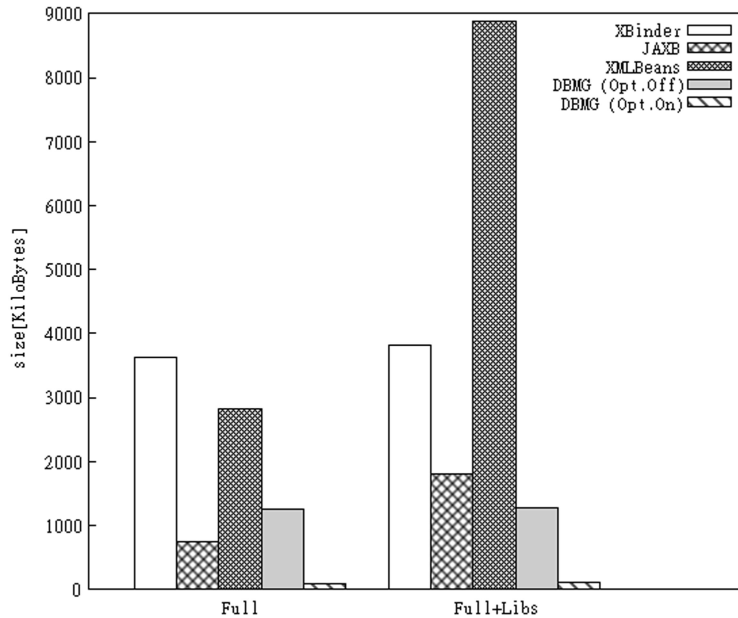

Figure 7.5: Size of generated code for full schemas

by DBMG. One of the reasons, the lack of serialisation support, was already mentioned above. Another reason is that XBinder produces code for most user-defined simple types, generating code for ensuring that the value follows all of the restrictions specified in the schemas. This is an advantage if we parse data obtained for a non-trusted source and the application requires the data to be carefully validated, but it is a disadvantage in the opposite case, as unneeded verification increases processor usage and memory footprint. In the case of DBMG, as it aggressively tries to lower final code size, these simple type restrictions are ignored and these types do not even have a counterpart in the generated code. This is the same strategy used by JAXB which justifies the small size of the code generated with this tool. A final point making DBMG code simpler is that it does not check the order of elements' children order when processing XML nodes. This, to maximise the number of XML documents potentially containing invalid data that must be parsed by the application.

Summarising, we can conclude that the code generated by *DBMobileGen* for the case study is substantially smaller than the code generated

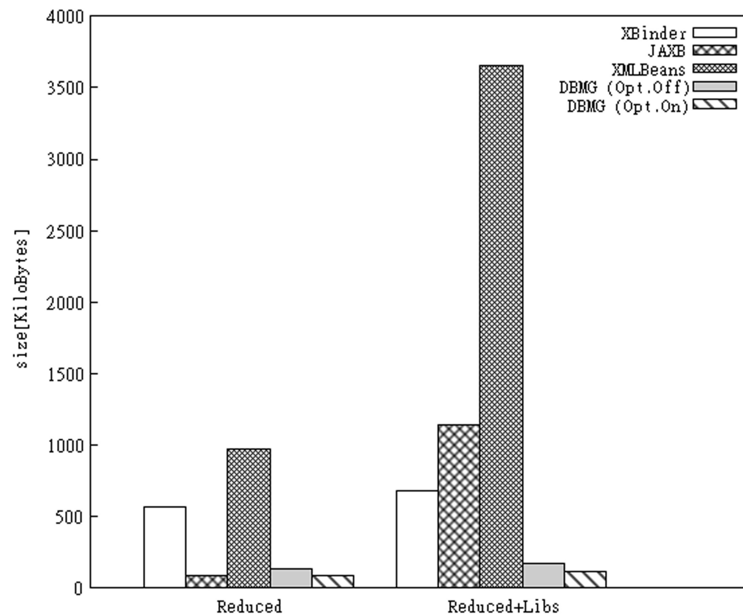

Figure 7.6: Size of generated code for simplified schemas

by other tools. If we compare this tool directly with XBinder, the only generator known by the authors producing code for Android, we can say that it generates much more compact code for the analysed scenario (almost 6 times smaller) and provides more flexibility to parse potentially invalid XML documents. This, at the price of delegating data validation code to other parts of the application.

7.5 Sample Applications

As a final proof of the feasibility of building actual mobile applications we briefly describe here two sample applications built using *DBMobile-Gen*. The first application is a basic WPS client and the second one is a full-fledged generic SOS client. Both applications are targeted to the Android mobile platform.

7.5.1 WPS Basic Client

The Web Processing Service (WPS) [OGC07g] is one of the newest implementation specifications. It aims to provide geospatial processes through the web, allowing the reutilisation of legacy systems, the execution of long-running geospatial algorithms in more capable servers, etc. Compared to other specifications WPS presents an additional challenge as processes may define their own data input and output formats, making the development of a client a very specific task, which cannot always be reused to build other clients. This also makes the possibility of building clients for running in mobile devices a harder task, as they present serious constraints in memory and processing capabilities.

The adoption of geoprocessing service clients in mobile devices is practically unexistent, even when the model provided by the Web Processing Services (WPS) seems to fit the philosophy of accessing computation intensive processes from devices with less capabilities. The main reason to such a low use of WPS in mobile computing is that the exchanged geospatial data is often encoded in some XML-based format that demands large processing power for the targeted devices.

The number of WPS clients, or software components to build them, is scarce. Due to the nature of WPS itself, which is just a generic protocol to execute remote processes, it is very difficult to build a generic client that suits every process because of the great degree of freedom of input and output parameters.

We present here a WPS client based on Google Maps that allows users to enter simple GML geometries that will be used as input to remote processes. The operations tested are: *buffer*, *intersection*, *area*[8], and *population statistics*[9]. Figure 7.7 shows screenshots for the calculation of the *area* (top) and *buffer* (bottom) of geometries introduced by touching the device screen. The geometries are sent to the server on user request and the result is displayed as a different layer on the map.

[8]Geometric processes are executed in `http://elcano.dlsi.uji.es:8080/topologywps/WebProcessingService`

[9]Populations statistics are provided by the *Population Estimation Service*: `http://sedac.ciesin.columbia.edu/gpw/wps.jsp`

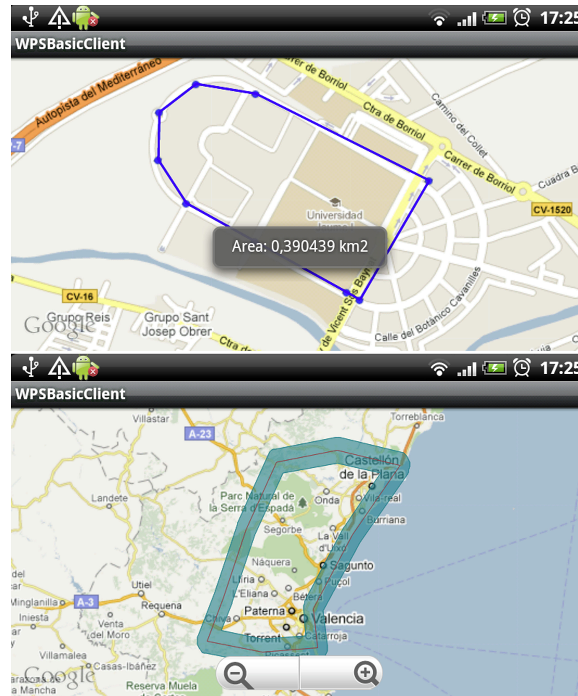

Figure 7.7: WPS Google Maps-based client

Application Architecture

Figure 7.8 shows the architecture of the application following a *Layered Application Pattern* [BMR+96]. The application contains layers for *user interface*, *bussines logic* and *communication*.

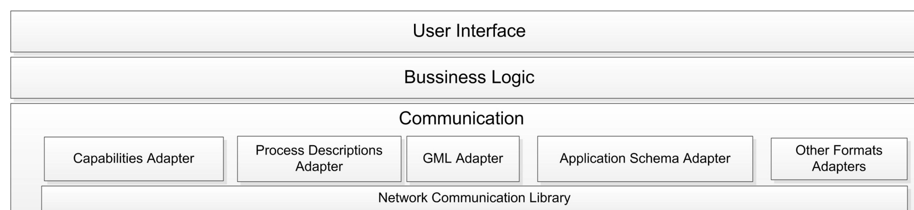

Figure 7.8: WPS basic client architecture.

Of special interest to us is the communication layer, which includes a light-weight network communication library, as well as a set of adapters

generated using *DBMobileGen*. Adapters are in charge of transforming WPS messages, encoded in XML, into application objects in the bussines layer (*Capabilities, Process descriptions, GML adapters* in Figure 7.8). Adapters for other formats, e.g. raster formats, can be added indepedently.

XML Processing Code

The XML processing code for this application is not trivial as different kinds of geometries are involved. If we look at the process descriptions of the supported operations we can see that responses to the *buffer* and *intersection* operations are XML instances based on several versions of the GML schemas. Even more, if GML 3, in any of its subversions, is used the result is encoded in a particular GML application schema (52NAS in Table 7.2). The *area* operation returns a value embedded in a WPS *ResponseDocument* file. Last, the *Population Estimation Service* returns its result in its own GML application schema (PSAS in Table 7.2).

If we capture some responses of the given remote operations and we apply the instance-based schema simplification algorithm we can reduce the schemas in a large degree. Being pragmatic in the case of the first two operations we will support only the application schema based on version 3.1.1 of GML. Table 7.2 show the number of complex types in the schemas before and after applying the algorithm.

Table 7.2: Comparison between the number of complex types in the schemas related to the WPS Basic Client before and after applying the instance-based simplification algorithm

	#CT (before)	#CT (after)	% reduction
WPS schemas	99	11	87.1
52NAS	395	18	94.4
PSAS	59	6	89.8

The WPS schemas can be largely simplified because only the subset needed to parse a simple *Execute* response document with a single literal value as result must be processed. The 52 North WPS Application Schemas includes explicitly all of the GML 3.1.1 schemas, but only reference directly types *gml:AbstractFeatureType* and *gml:SurfacePropertyType*. In the instances files considered the server always used *gml:Polygon* to

describe the result of the operations. In the case of the *Population Estimation Service*, it uses a geometry in the response, but as we are only interested in the statistics we can safely ignore it and focus only on the rest of the data.

7.5.2 SOS Mobile Client

The second application is a generic SOS client. This client must support the following functionality:

- *Support for the core profile of SOS specification*: The operations *GetCapabilities*, *DescribeSensor* and *GetObservations* must be supported
- *Visualisation of sensor locations and information*: The application must allow users to visualise sensor and sensor systems location and its associated metadata.
- *Visualisation of observations in tables and graphics*: The application must provide flexible ways to manipulate the parameters used to request information about observations.
- *Support for temporal and spatial filters*: The opportunity of filtering observations requested to the server using temporal and spatial filters must be provided. Specifically, observations for a given instant of time, or gathered during a time interval, or restricted spatially to a bounding box may be retrieved.
- *Easy server information management*: The information about the URLs of the servers must be stored to allow users to connect to these servers without having to introduce all of the information again.

The application has an architecture that is almost identical to the WPS client, having layers for *user interface*, *bussines logic* and *communication*. The communication layer, in addition of handling connections to SOS servers, must allow to connect to the SQLite[10] database used to store the information about servers. Fortunately, Android provide native support for these databases, so it is an almost trivial task.

[10] `http://www.sqlite.org/`

XML Processing Code

In this case, the XML processing code for this application may be generated similarly as it was done for the case study presented in Chapter 6. The main difference in this case is that the SOS application is meant to be generic, meaning this that it must be capable of connecting to a potentially large number of SOS servers. Still, using the full SOS schemas to generate XML processing code is not advised, as it has been shown that the size of the generated code can be too large. For this reason what we do is to use a larger set of XML documents as input to maximise the number of XML documents that can be parsed. As input we use the set of XML instances that will be described in Chapter 8. This set has been gathered from a set of 56 SOS servers available online.

The XML processing code has been generated using the input files described above with *DBMobileGen* with only the optimisation related to the use of the subset of the schemas enabled and the rest of the optimisations disabled. These optimisations are disabled because the generated code will probably has to be extended in the future if servers that use in their responses schema components not used in the initial XML instances must be supported. If the new documents to be parsed are substantially different than the initial input set the best solution is to regenerate the code, but if not, modifying the code manually will be easier if it is not highly optimised.

The size of the generated code for this client is 214 KB including the supporting libraries in compressed JAR format.

Application funcionality description

We describe here briefly the functionality of the SOS client. We start describing the server management facilities offered by the application. Figure 7.9 shows to the left the first screen shown to the user which contains a list of all of the servers introduced previously to the application. The user may add or remove server from this list. Selecting a server from the list establishes a connection with the server. This process implies that the capabilities file is retrieved from the server. Once the connection is established the user is informed about it.

At this point, the user may select from the list of observation offerings those we are interested in, and the related sensors are shown in the map. Figure 7.10 shows a set of sensors in the application map window. If several sensors have the same exact location the number of sensors at

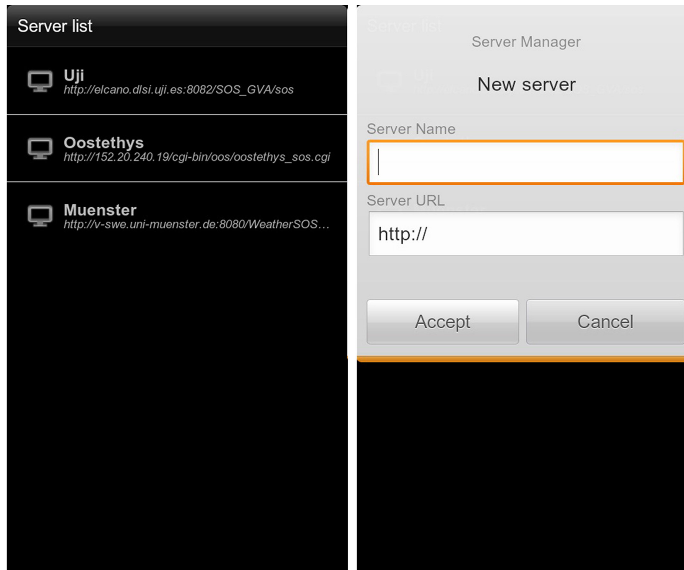

Figure 7.9: Selecting or adding servers

that location is added to the marker representing the sensors. Users can tap on a marker to see the metadata associated to a sensor.

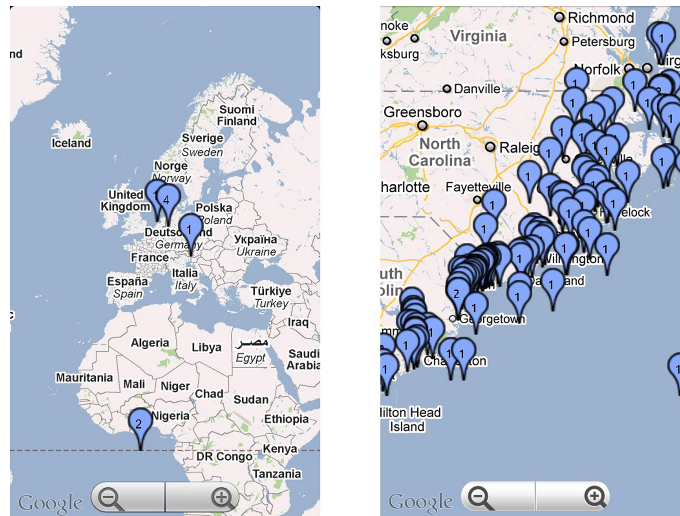

Figure 7.10: Showing sensors stations in the map

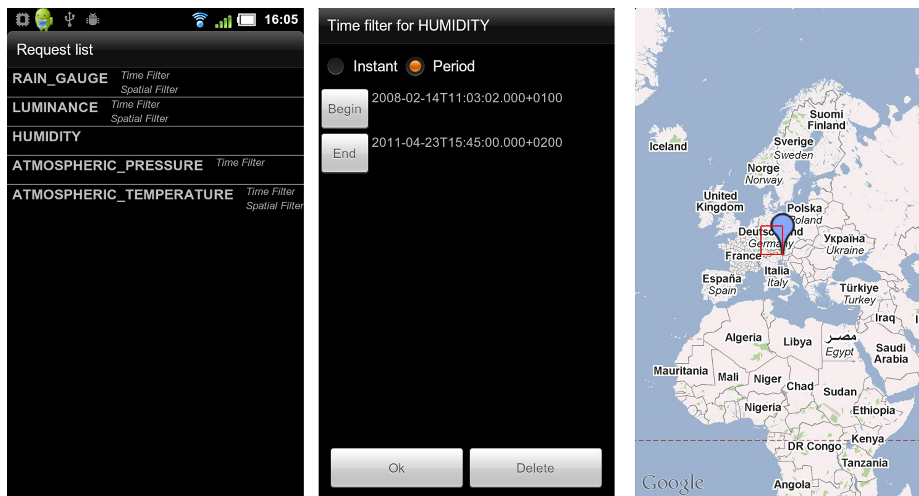

Figure 7.11: Specifying Filters

To request observations, a set of sensors must be selected in the map. These sensors are used to initialise some of the parameters of the *GetObservation* request, which can be further refined through the *GetObservation* wizard. The wizard checks that mandatory parameters are correctly set and allows the spatial and temporal filters to be specified as is shown in Figure 7.11.

Last, after the observations have been retrieved from the servers they can be shown in tables or charts as shown in Figure 7.12

7.5.3 Challenges and Open Issues

During the development of the sample applications we faced several challenges. The most important one was the size of XML schemas used to describe the encoding of geospatial data. For example, in the WPS basic client, even with the assumption that only the two described application schemas were supported, the overall number of complex types for the whole schemas was more than 500. Considering the limited application functionality, this number is rather large. This problem was solved by extracting only the section of the schemas that are needed for the application using the instance-based simplification algorithm intergrated in *DBmobileGen*. Although, the process of using only sections of the schema

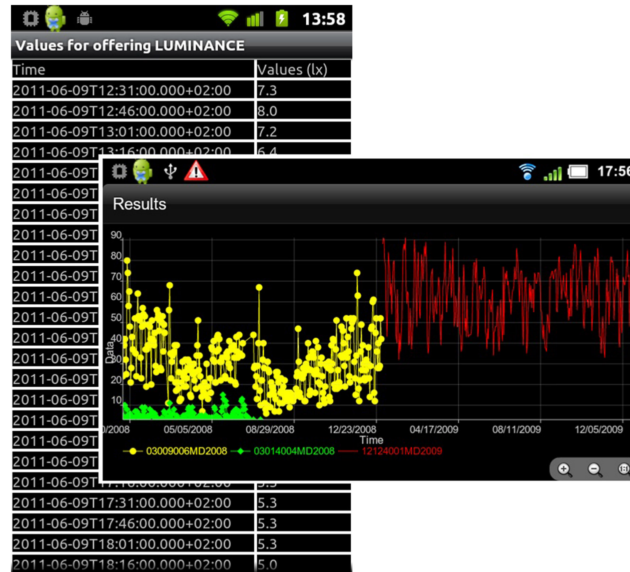

Figure 7.12: Tables and charts

can be accomplished using a manual approach, using a dedicated tool is a more time and effort saving approach.

Another challenge was writing light-weight communication libraries, which was accomplished by implementing only the required behaviour. For example, the WPS specification allows operation requests to be encoded using HTTP GET with KVP or HTTP POST with XML payload, but for each operation only one of them is required. In the case that the specification do not mandates a binding type or the other we preferred HTTP GET request for operations with simple parameters and HTTP POST for operations with more complex parameters.

In addition to these challenges, we have identified some open issues in the topic of OWS for mobile devices. The first issue is the lack of a mechanism to now beforehand is we have enough memory or processing capabilities to handle a server response. When a request is issued to the server, there is no way to know if the response will be too large to be handled by the device. A mechanism to estimate the response size would be of invaluable importance for a resource constrained device. Large responses can cause memory problems and, as the network hardware of the phones consumes a lot of battery power, they may also drain the phone battery. The problem with large responses is aggravated by the use of

a verbose base encoding format such as XML. In this regard, in version 2.0 of SOS, still passing through the OGC standardisation process, an extension to retrieve metadata about available data has been proposed [OGC10d]. This extension contains the operation *GetDataAvailability*, which according to [BEJ+11] enables clients to discover the temporal relationship between given procedures, observed properties and features of interest. The operation contains parameters that can be used by clients to indicate for which period of time these relationships are to be discovered and also to generalise the information about the temporal relationships.

The second issue is related with the use of code generation techniques for sections of the applications in the business logic and user interface layer. During the development of the applications, common coding patterns for drawing geometries on the screen or related with the user interface components used to interact with the servers were noted. A further look into these topic to generate a larger sections of the final application might be interesting.

7.6 Concluding Remarks

In this chapter we have presented a detailed description about how we generate code using the Instance-based XML data binding approach. Utilising information about how XML documents make use of its associated schemas we can refine the code generated to optimise it according to certain criteria. In this case, our main goal is to make the final binary code as small as possible to ease its inclusion in application targeted to resource-constrained devices. We have shown here how the two steps of the approach link to each other, using the output of the schema simplification step as input to the code generation process. We have also implemented a code generator to asses the approach, and targeted to the Android mobile platform. The generator, called *DBMobileGen*, offers an interesting number of features that allow the production of very compact code. The effectiveness of this tool has been demonstrated through a set of experiments, and with the generation of the XML processing code for two sample applications.

Part IV

Experiments

CHAPTER 8

Empirical Study of SOS Server Instances

Early in our research endeavour, we realised that we must provide some proof for one of our main assumptions that claims that actual OGC web services implementations do not use all of the capabilities included in the specifications schemas. This statement may be obvious to some extent to people with experience in the development of OGC standard-based software, but it does not have to be so for a wider audience. For this reason we present in this chapter a study about how a set of Sensor Observation Service (SOS) server instances use the schemas associated to this specification. This is done not only to prove that the assumption is true, but also to try to quantify how much of the schemas is used in actual implementations.

The study was initially focused on which parts of the specification schemas were used and which parts not, but is later widened to include many practical issues about how SOS server are implemented in practice.

The content of this chapter has been published with the same title in *"Advancing Geoinformation Science for a Changing World"*, Lecture Notes in Geoinformation and Cartography, 2011, Volume 1, Part 3, pages 185-209. DOI: 10.1007/978-3-642-19789-5_10.

8.1 SOS Server Instances

In order to carry out our study we gathered information from a set of SOS server instances freely available on the Internet. The URLs of these servers are listed in Appendix A. These servers where located using web services catalogs such as the *OWS Search Engine*[1] and *IONIC RedSpider Catalog Client*[2], and using general-purpose search engines such as *Google* and *Yahoo!*. The server list only shows those claiming to support version 1.0.0 of the standard.

Starting from these servers we retrieved a sample set of XML documents including service metadata, and sensor and observation information. These documents were then analysed mainly regarding to schema validity and used features. The results from this analysis are shown extensively in Section 8.4.

8.2 Limitations of the Study

This study presents some limitations. First, it is impossible to retrieve all of the information published on the servers. We tried to overcome the effects of this limitation by making the sample dataset as large as possible and, in cases where several alternatives existed for making a request, we retrieved at least one instance file from each alternative. Second, only responses from the core profile operations were considered. This is because most servers do not implement the rest of the operations (see Section 8.4.1). Third, we did not test server instances for full compliance to the SOS specification; we only dealt with the information contained in the XML instance files and XML schema files. Last, we analysed server instances without considering the server product used to deploy the instance. This is because for several instances we were not able to determine which product was used, and in some cases handcrafted servers have been developed for specific problems.

[1] `http://ows-search-engine.appspot.com/index`

[2] `http://dev.ionicsoft.com:8082/ows4catalog/elements/sos.jsp`

8.3 Dataset Description

Details about the information contained in the sample dataset are presented in Table 8.1. The table includes the following information for the responses of the considered operations:

- *Number of files (NF)*: Number of files retrieved for the operation.
- *Number of objects described (NO)*: Depending on the operation these objects can be observation offerings, in the case of the *GetCapabilities* operation; sensor systems, in the case of *DescribeSensor*; and observations, in the case of *GetObservation*.

Table 8.1: Dataset description

Operation	NF	NO
GetCapabilities	56	7190
DescribeSensor	6,719	6,719
GetObservation	204	3,990,656
Total	6979	4,004,565

8.4 Results

In this section we present the results of computing the sample dataset according to the following metrics:

- *Number of invalid files:* Number of invalid files according to the schema files.
- *Most frequent validation errors:* List with most frequent errors found during validation, including an error description and the number of occurrences of each error.
- *Used Features:* The features presented depend on the analysed operation. For example, while analysing the capabilities files we considered supported operations or filters and response formats. While analysing observation files, we considered, for example, which observation type is most frequently used to encode the information gathered by sensors.

- *Parts of the schemas that are actually used:* Schema files defining the message structures for SOS are large and complex, moreover, SOS schema files depend on schema files included on other specifications as well. For these reasons actual implementations only use a subset of these schemas.

We present the results of applying the first three metrics divided by operation. Then, in a different section we analyse the part of schemas that are actually used.

8.4.1 Capabilities Files

The capabilities file of a server contains all of the information needed to access the data it contains. In the case of SOS servers, this file contains available observation offerings, supported operations and filters, etc.

Instance validation

The first important fact extracted from the sample dataset is that 34 out of 56 (60.7%) capabilities files are invalid according to the schemas defining their structure. Table 8.2 shows the most frequent errors found in the instance files.

The most frequent error found was the use of a different name for an element that the one specified in the schemas. For example, this was the case for element *sos:Time*, which specifies the time instant or period for the observations within an offering. The element name was changed to *sos:eventTime* in some of the servers, maybe because that was the name used in previous versions of the specification. The second most frequent error was elements with invalid content (errors 2, 3, 4 and 5). Common mistakes were time values with incorrect format, or offering ID values containing white spaces or colons.

Despite the large number of errors found, most of them did not prevent the files from being correctly parsed, although they supposed and extra amount of work while implementing the parsers. At the end only 2 of the 56 files contained serious errors, which makes parsing their content impossible for us.

Supported Operations

The capabilities files also indicate which operations are supported by the servers, including information about how to access them and which

Table 8.2: Most frequent validation errors for capabilities files

	Error code	Description	Number of Ocurrences
1	cvc-complex-type.2.4.a	Invalid content was found starting with element [element name]. One of valid element list is expected	2,754
2	cvc-complex-type.2.2	Element must have no element [children], and the value must be valid	978
3	cvc-datatype-valid.1.2.3	[value] is not a valid value of union type	960
4	cvc-attribute.3	The value of attribute on element is not valid with respect to its type	468
5	cvc-datatype-valid.1.2.1	[value] is not a valid value of union type	379
6	cvc-id.2	There are multiple occurrences of ID value	107

values are allowed as parameters. Table 8.3 shows which and how frequently the different operations are supported.

The results, also depicted in Figure 8.1, show that all of the servers implement the *GetCapabilities* request using HTTP GET as required by the SOS implementation specification. Apart from that, most of them also implement the operation using HTTP POST. Most complex requests such as *GetObservation* are implemented easier using HTTP POST than using HTTP GET, as the SOS specification does not define KVP encodings for this operation.

The core profile is mandatory for every server but 10 of the 56 servers do not implement the *DescribeSensor* request, or at least they do not include it in the capabilities file. Some operations for the transactional and enhanced profile are implemented by a few server instances and some of them are not implemented at all.

Table 8.3: Operations supported by the server instances

	Operation Name	Profile	GET Support	POST Support
1	GetCapabilities	Core	56	54
2	DescribeSensor	Core	33	45
3	GetObservation	Core	42	54
4	RegisterSensor	Transactional	0	2
5	InsertObservation	Transactional	1	2
6	GetFeatureOfInterest	Enhanced	0	12
7	GetObservationById	Enhanced	0	10
8	GetResult	Enhanced	0	1
9	GetFeatureOfInterestTime	Enhanced	0	0
10	DescribeFeatureType	Enhanced	0	0
11	DescribeObservationType	Enhanced	0	0
12	DescribeResultModel	Enhanced	0	0

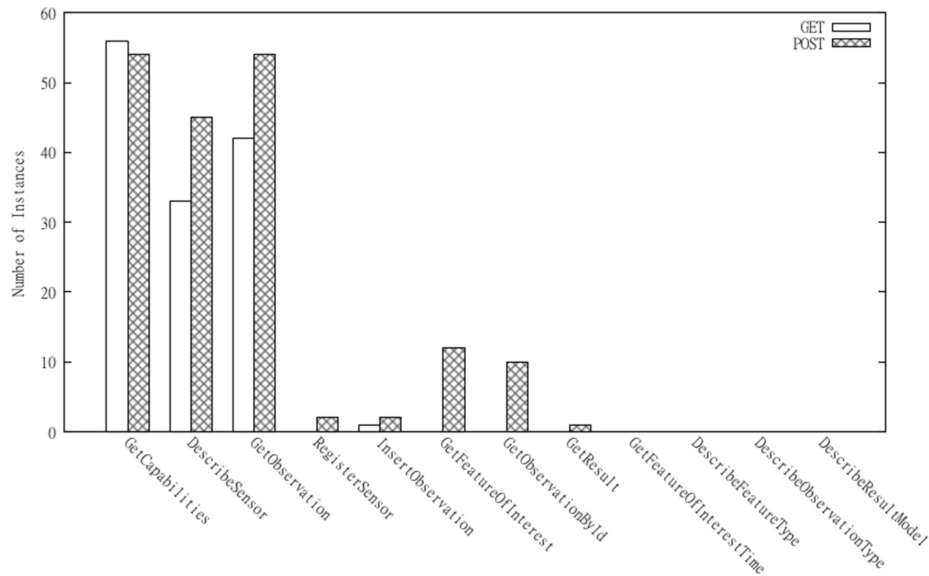

Figure 8.1: Support of SOS operations in actual server instances

Supported Filters

The number of potential observations published on a server can be very large. For this reason, filters are used to request just the observations in which we are interested. Filters for SOS fall into four categories: *spatial*, *temporal*, *scalar*, and *identifier* filters. Only 16 of the 56 (28.5%) capabilities files include information about the supported filters. These filters are detailed in Table 8.4. For each filter category the supported operands and operators are shown, as well as how frequently they have been used.

The most implemented filters are *BBOX* and *TM_During* that allow to restrict the location of the observations to a given bounding box or to a given time period respectively. Id filters are also frequently implemented. They allow information to be filtered by specifying the ID of entities related with the request. Even though some servers do not include the filter capabilities section, most of them allow observations to be also filtered using a bounding box or a time interval.

Supported Response Formats

Observations published on different server instances are encoded using several different formats. These formats and the number of offerings that represent information with them are presented in Table 8.5. The most widely supported format to represent observations is O&M 1.0.0, which is the default format specified by SOS.

Offerings Information

Observation offerings contain information about a set of related sensor observations. The SOS specification does not say how observations, procedures or observed properties should be grouped into offerings. For this reason, it would be very interesting to know how this grouping is realised in actual implementations. Regarding observation offerings we computed the followed metrics:

- *Number of offerings per server* (OpS): How many offerings are usually published on a server.
- *Number of procedures per server* (PpS): How many sensor or sensor systems are published on a server.
- *Number of observed properties per server* (OPpS): How many observed properties are usually published on a server.

Table 8.4: Support of filters

Filter Category			Number of Appearances
Spatial Filters	Operands	gml:Envelope	16
		gmlPolygon	11
		gml:Point	11
		gml:LineString	11
	Operators	BBOX	15
		Contains	11
		Intersects	11
		Overlaps	11
		Equals	1
		Disjoint	1
		Touches	1
		Within	1
		Crosses	1
		DWithin	1
		Beyond	1
Temporal Filters	Operands	gml:TimeInstant	16
		gml:TimePeriod	16
	Operators	TM_During	15
		TM_Equals	14
		TM_After	14
		TM_Before	14
		TM_Begins	1
		TM_Ends	1
Scalar Filters	Operators	Between	14
		EqualTo	13
		NotEqualTo	13
		LessThan	13
		LessThanEqualTo	13
		GreaterThan	13
		GreaterThanEqualTo	13
		Like	12
		NullCheck	1
Id Filters		eID	16
		fID	15

Table 8.5: Formats supported to represent observation information

Format	Number
text/xml; subtype="om/1.0.0"	5110
text/xml;schema="ioos/0.6.1"	2064
text/csv	664
application/vnd.google-earth.kml+xml	664
text/tab-separated-values	664
application/zip	110
text/xml	4
application/com-binary-base64	1
application/com-tml	1

- *Number of offering as points*: An interesting peculiarity observed during the experiments is that location of most offerings is a point, instead of a bounding box.

The result of computing the first three metric values is shown in Figure 8.2. The figure shows the values grouped into 6 categories. The number of offerings per server ranges from 1 to 1772. 48% of the servers contain 1-4 offerings, and 63 % contain 16 or less. This indicates that servers tend to group observations in a few offerings. Similarly, the number of procedures per server ranges from 1 to 1957. Although in a lesser degree than the case of offerings, the number of servers with a large amount of procedures per server is always lower than the number of servers with a small number of procedures. The number of observed properties per server ranges from 1 to 114. This number behaves much like the previous ones having 65% of the server instance with less than 16 observed properties advertised.

A last interesting phenomenon found here is the number of observation offerings which are restricted to a point in the space. Each offering has a property named *boundedBy* defining a bounding box where the observations grouped in the offering are located. In 6,575 offerings in the sample data set the bounding box was indeed a point, representing the 95.7% of the total number of offerings. This clearly indicates that the first criteria followed to group observations into offerings is the sensors location, which in most of the cases is a single point on Earth. Figure 8.3 shows as an example a set of offerings located in North America represented in Google Earth. In the figure, placemarks represent point offerings and bounding boxes represent other offerings.

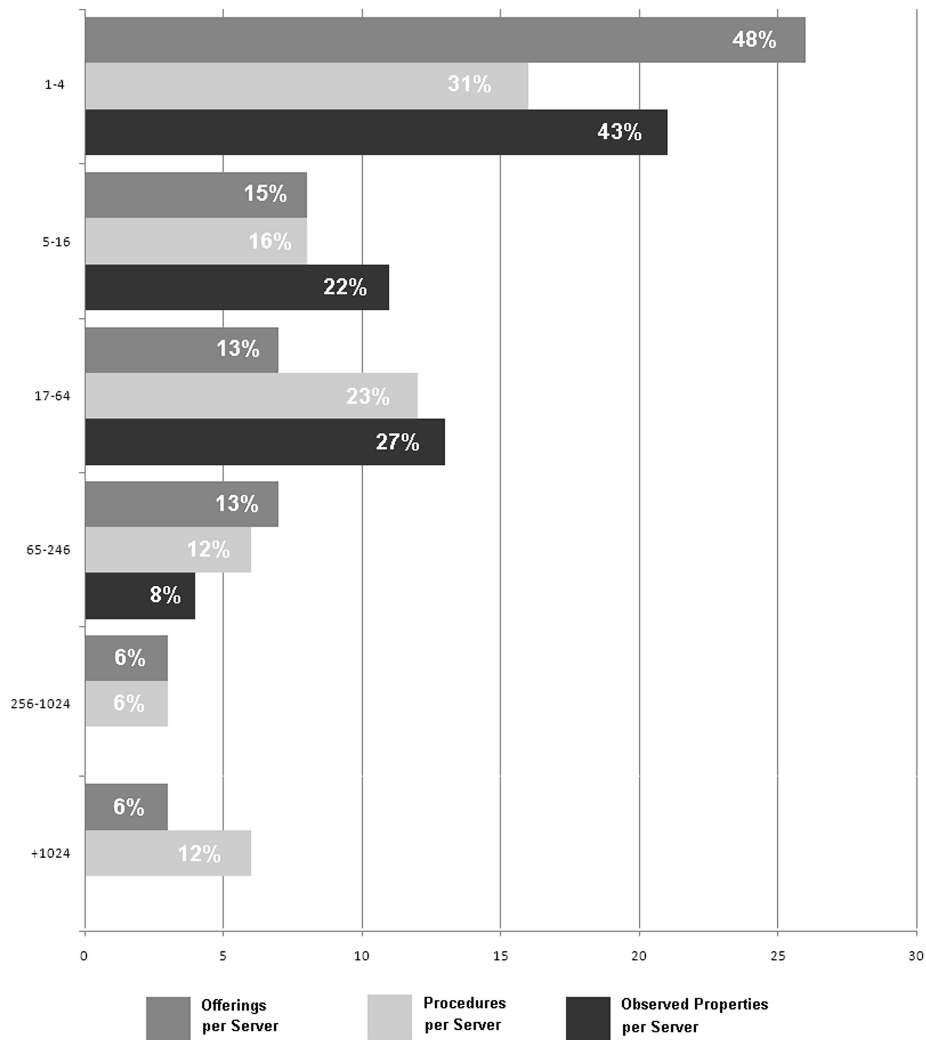

Figure 8.2: Number of servers classified by the number of offerings, procedures and observed properties

8.4.2 Procedure Description Files

The 56 servers considered in this study mention in their capabilities files 12,222 procedures. From this number we were able to retrieve the description files of 6719 of them (54.9%). All of these files were encoded in the *SensorML* format.

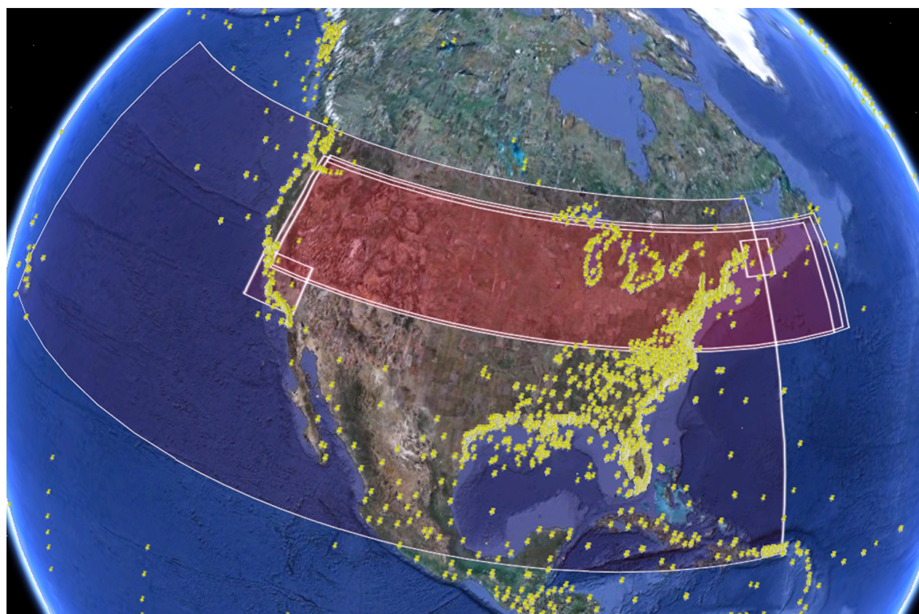

Figure 8.3: Observation offerings in North America

Instance Validation

The validation of the sensorML files gave as result that 1,896 files were invalid according to the XML schemas files defining the structure of these documents. The value represents 28.2% of the overall number of files. The most frequent errors found are presented in Table 8.6. The first error type occurred frequently because required elements where omitted or elements not defined in the schemas where introduced in the wrong place. Errors 2, 4 and 5, similarly to the case of capabilities files refer to incorrectly formatted values: identifiers including whitespaces or colons, incorrect time values, or values just being left empty. The most serious errors were those of type 2. In these cases, wrong use of namespaces, or not specifying the version of the schemas used, made it impossible to process the documents at all.

Procedure Description Types

The sensorML specification models sensor systems as a collection of physical and non-physical processes. Physical processes are those where

Table 8.6: Most frequent validation errors for sensor description files

	Error code	Description	Number of Ocurrences
1	cvc-complex-type.2.4.a	Invalid content was found starting with element [element name]. One of valid element list is expected	1,778
2	cvc-attribute.3	The value of attribute on element is not valid with respect to its type	556
3	cvc-elt.1.a	Cannot find the declaration of element [element name]	500
4	cvc-datatype-valid.1.2.1	[value] is not a valid value of union type	300
5	cvc-pattern-valid	Value is not facet-valid with respect to pattern for type	256

information regarding their positions and interfaces may be relevant. Examples of these processes are detectors, actuators, and sensor systems. Non-physical or *"pure"* processes according to the specification can be treated as merely mathematical operations [OGC07c]. These categories are further subdivided as shown next:

- *Physical processes*
  - *Component*: Any physical process that cannot be subdivided into smaller sub-processes.
  - *System*: It may group several physical or non-physical processes.
- *Non-physical processes*
  - *Process Model*: Defines an atomic pure process which is used to form process chains.
  - *Process Chains*: Collection of executable processes in a sequential manner to obtain a desired result.

From 6,219 processed *sensorML* files, 6,215 described *Systems* (99.9%), and 4 of them *ProcessChains*. This indicates that the usual is to describe

sensor systems that have a location in space and measure an observed property for a period of time.

Specifying Location

An important piece of information about the procedure is its location. Unfortunately for programmers, location can be specified in different parts of the procedure description file (sensorML file). In the sample dataset we have found this information located in at least three different places and using different names to identify coordinates:

- Under the location tag in the description of a *System* as a point:

```
<SensorML xmlns="http://www.opengis.net/sensorML/1.0.1"
  version="1.0.1" [Other attributtes]>
  <member>
    <System gml:id=[System ID]>
      ...
      <location>
        <gml:Point srsName=[SRS Name]>
          <gml:coordinates>39.99 -0.068 0</gml:coordinates>
        </gml:Point>
      </location>
      ...
    </System>
  </member>
</SensorML>
```

- Under the *position* tag in the description of a *System* as a vector with named elements:

```
<SensorML xmlns="http://www.opengis.net/sensorML/1.0.1"
  version="1.0.1" [Other attributtes]>
  <member>
    <System gml:id=[System ID]>
      ...
      <sml:position name=[name]>
        <swe:Position referenceFrame=[SRS name]>
          <swe:location>
            <swe:Vector>
              <swe:coordinate name="x">
                <swe:Quantity>
                  <swe:value>-0.068</swe:value>
                </swe:Quantity>
              </swe:coordinate>
              <swe:coordinate name="y">
                <swe:Quantity>
                  <swe:value>39.99</swe:value>
                </swe:Quantity>
              </swe:coordinate>
              <swe:coordinate name="z">
                <swe:Quantity>
```

```
                    <swe:value>0</swe:value>
                  </swe:Quantity>
                </swe:coordinate>
              </swe:Vector>
            </swe:location>
          </swe:Position>
        </sml:position>
        ...
      </System>
    </member>
  </SensorML>
```

- Under the *positions* tag in the description of a *System* as a list of positions:

```
<SensorML xmlns="http://www.opengis.net/sensorML/1.0.1"
  version="1.0.1" [Other attributtes]>
  <member>
    <System gml:id=[System ID]>
      ...
      <sml:positions>
        <sml:PositionList>
          <sml:position name=[name position 1]>
            [Position data]
          </sml:position>
          <sml:position name=[name position 2]>
            [Position data]
          </sml:position>
          ...
        </sml:PositionList>
      </sml:positions>
      ...
    </System>
  </member>
</SensorML>
```

In the first case reading the coordinates values is straightforward, the values are grouped together into a *gml:Point* object. In the second one, several tags must be parsed to reach the coordinates; a problematic issue at this point is that different names are used by servers to refer to the coordinate values. For example *longitude* was also named *x* or *easting*; *latitude* was also named *y* or *northing*; and *altitude* was also named *z*. The contents and attributes of the tags involved are also slightly different, some servers includes unit of measurements, some include the axis they refer to, etc. The third case is a generalisation of the second one, where positions are included in a list, allowing more than one to be specified. None of the analysed files included more than one position for a sensor or sensor system.

8.4.3 Observation Files

To analyse *GetObservation* responses, 1.7 GB of observation data was retrieved from the server instances. All of the retrieved files follow the format specified by O&M 1.0.0 encoding specification. As it was shown in Table 8.5 this is the most widely used format and it is the default encoding for observations in SOS 1.0.0.

Instance Validation

Validation of observation files was much more difficult than expected. The validation process failed repeatedly to process correctly large files (>10MB) and did not allow the validation of files containing measurements alleging that schema files were incorrect. *Measurements* are specialised observations where the observation value is described using a numeric quantity with a scale or using a scalar reference system [OGC07a]. Large files were only a few, so the first limitation was not a great problem but files containing measurements were about half of the whole observation files. Although we were able to parse correctly all of the observations, we were only able to apply the validation process to 62 files (31.3%). From these 62 files, 56 were reported to be invalid (90%). Details about the errors found are shown in Table 8.7.

Table 8.7: Most frequent validation errors for observation files

	Error code	Description	Number of Ocurrences
1	cvc-attribute.3	The value of attribute on element is not valid with respect to its type	206
2	cvc-complex-type.2.4.a	Invalid content was found starting with element [element name]. One of valid element list is expected	189
3	cvc-datatype-valid.1.2.1	[value] is not a valid value of union type	121

The validation errors for observation files are similar to those for capabilities and sensor description files. Values with wrong formats, and

wrong named or misplaced elements made up all of the errors found in the instance files.

Observation Types

According to the O&M 1.0.0 encoding specification observation types are organised as shown in Figure 8.4. The base type for all observations is *ObservationType*, which inherits from *AbstractFeatureType* located in GML schemas. Starting from *ObservationType* a set of specialisations is defined based on the type of the results contained in the observations. Additionally, information providers can derive their own observation data types from the different types in the figure.

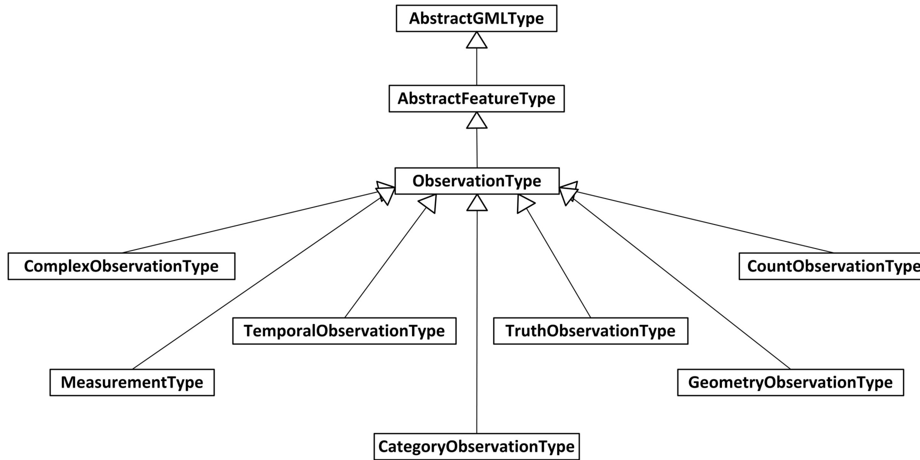

Figure 8.4: Hierarchy of observation types

From 3,990,656 observation values processed in the dataset, 56.3 % (2,246,639) of the values were *Observation* elements (instances of *ObservationType*), and 43.7 % (1,744,017) were instances of *Measurement* elements (instances of *MeasurementType*). Values corresponding to none of the other types were found in the sample dataset. Despite the fact that the number of measurement values was lower than the number of observations, the amount of disk space needed to contain these values was about 7 times larger than the space occupied by the observations (1533 MB against 213 MB). This difference in size seems to be the cause why most

implementations choose not to use observation specialisation types. Although the lack in the O&M specification of well-defined semantic models might influence this decision as well [Pro08, Kuh09].

8.5 Subset of XML Schemas Used

The last piece of information we extracted from the sample dataset is the subset of the XML schemas that is actually used by the server instances. The number of schema files associated to the SOS specification is huge. If we follow all of the dependencies from the main schema files of the specification we obtain a set of 87 files. If we additionally consider the observations specialisation schemas (containing the definition of *MeasurementType*) and their own dependencies this number grows up to 93. The size of schemas brings as a consequence that server instances only provide support for a subset of them.

Next, we calculate from the sample dataset which part of the schemas is used and which part is not used at all. To calculate this information we inspect the information contained on the instance files to determine which schema components are directly used in the files (initial set). After doing this, we determine which other schema components are used to define the initial set[3]. The algorithm used is similar to the one included in the GML subsetting profile tool, a tool used to extract subsets of the GML schemas [OGC04]. We present the results in two steps. First, we detail the subset of the GML schemas that is actually used. Second, a similar analysis with the overall results for the SOS specification is presented.

8.5.1 GML

GML constitutes more than 50% of the overall number of global schema components (types, elements, model groups) comprising the SOS schemas. It is used to model geographic features embedded into the instance files, and its components are extended or composed into new components of the SOS specification. As shown in Figure 8.5, most of the specifications relevant to our study depend to a large extent on GML. Table 8.8 shows a comparison between the number of components in the original GML schemas for version 3.1.1 (original files) and the subset of

[3] The algorithm used here is an early version of the Instance-based algoritm presented in Chapter 6. The results obtained with this older version has been kept here to maintain consistency with the results published in [TVGH11]

the schemas that is referenced directly, or used in the definition of others components referenced directly in the sample dataset (profile).

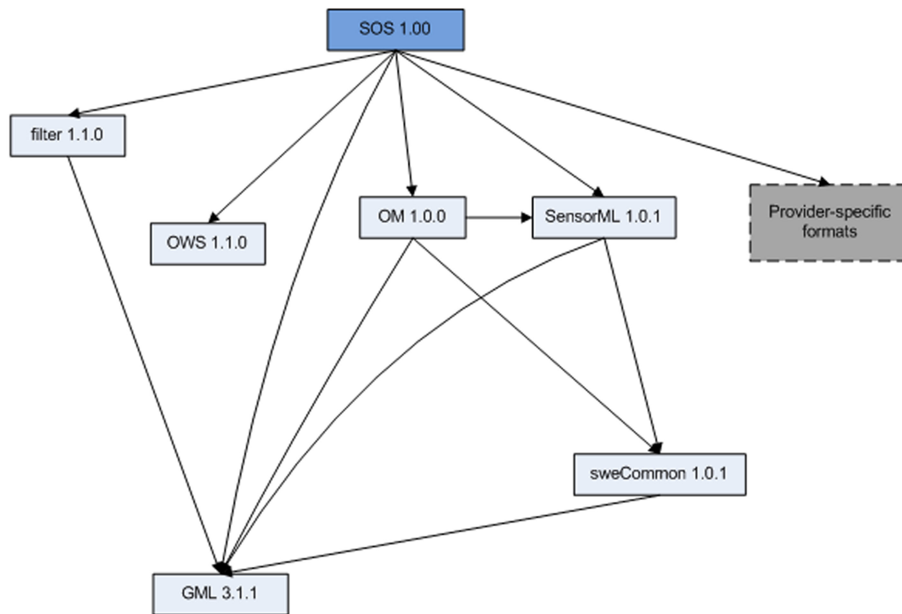

Figure 8.5: Dependencies of SOS from other specifications

Table 8.8: Comparison between overall number of components in GML and number of components actually used

	Original Files	Profile
#CT	394	60
#ST	64	15
#EL	485	74
#AT	15	9
#MG	12	2
#AG	35	4
#GLOBAL	1005	164

The results are divided by component type: complex types (#CT), simple types (#ST), global elements (#EL), global attributes (#AT),

model groups (#MG) and attribute groups (#AG). It turned out that only 16.3% of the components were actually used. All of the components contained in the following files were not used at all: *coverage.xsd, dataQuality.xsd, defaultStyle.xsd, direction.xsd, dynamicFeature.xsd, geometricComplexes.xsd, geometricPrimitives.xsd, grids.xsd, measures.xsd, temporalreferenceSystems.xsd, temporalTopology.xsd, topology.xsd,* and *valueObjects.xsd*.

8.5.2 SOS

As mentioned before, the full SOS schemas are comprised by 93 files, distributed by specification as presented in Table 8.9. This full set is calculated starting from the SOS main schemas and following the references specified with *include* and *import* tags. For example, a typical practice when accessing a component in the GML schemas is to import the whole schemas through the file *gml.xsd*. This way all of the GML schemas become referenced even when most of them are never used.

Table 8.9: Distribution of SOS schema files by specification

Specification	Version	Number of files
SOS	1.0.0	16
GML	3.1.1	32
SensorML	1.0.1	5
OM	1.0.0	3
SWE Common	1.0.1	11
Sampling	1.0.0	5
OWS	1.1.0	14
Filter	1.1.0	4
Others		3

Table 8.10 shows a comparison between overall number of components in the full SOS schemas (original files) and the subset of the components that is really needed as explained before in the case of GML (profile). The results are also displayed in Figure 8.6.

Table 8.10: Comparison between the overall number of components in SOS and number of components actually used

	Original Files	Profile
#CT	772	266
#ST	119	61
#EL	745	201
#AT	39	3
#MG	28	16
#AG	40	8
#GLOBAL	1743	515

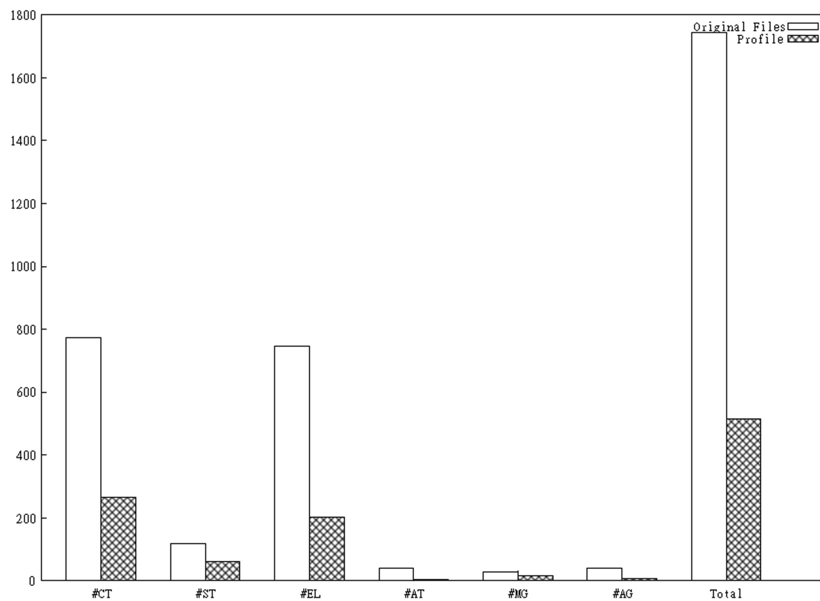

Figure 8.6: Overall number of schema components vs actually used components in SOS

8.6 Discussion

As the amount of information extracted from the sample dataset is large we present a summary of our findings:

1. The number of invalid instances files is high: 29% (1986 out of 6837),
2. Most of the validation errors found are not serious enough to prevent correct parsing,
3. Some servers do not implement all of the mandatory operations in the core profile,
4. Most servers do not advertise filtering capabilities,
5. Most servers use O&M to encode observations,
6. Most servers group observations into a small number of offerings, and they usually contain information about a small number of procedures and observation properties,
7. Offering locations were frequently points in space indicating that the first criteria to distribute observations into offerings is the sensors location,
8. Most procedure descriptions refer to *Systems*,
9. All of the observations in the sample dataset belong to only two types: observations and measurements,
10. The size on disk needed to represent measurements is much higher than the one used to represent the same information as basic observations.
11. Most servers only support operations from the core profile,
12. Procedure location is specified in at least three different parts of the sensorML documents, and sometimes coordinates are referred to under different names. This problem could be solved by allowing only one of the three choices. If multiple locations can be specified for a procedure the more general solution would be the most appropriate, although we did not find any instance with more than one location in the sample dataset.
13. Only 29.5 % of the full schema set for SOS is used by the sample dataset

The first four points are closely related to interoperability. The presence of invalid files increases the chance of parsing errors in client-side applications. The fact that most errors are easy to overcome if writing the parsers manually, does not deny the fact that may limit the applicability of XML data binding code generators if they are strict regarding schemas validity. Not supporting mandatory operations may also lead clients to fail if they request these operations to the server. Not advertising filtering capabilities simply prevents the clients to effectively filter the observations, unless they know beforehand how the server works. Next six points (5-10) provides useful insight for optimizing server and client implementations. Knowing which formats, offering grouping strategies, and types of sensor and observation representations are more commonly used could be utilised to optimise implementations in these scenarios. Even more, they could indicate which features are most likely to stay in future versions of the specification. Point 10 is specially revealing if large amounts of information are being handled. In this case using measurements is not the right choice for encoding information.

The last three points, in our opinion, reflect the complexity of the SOS specification. The number of operations in the specification is high if compared with others OGC specifications. In addition, the complexity of the formats that must be supported such as SensorML, O&M, SWE common, and GML, makes the implementation of the core profile itself a complex task. The example of how location is specified for procedures shows that even getting a simple piece of information can be a difficult thing to do. The last point could be the result of two options: the schemas are too complex to be implemented in its entirety or most of the information included or referenced by the schemas is not needed in real scenarios. In our opinion both options are true to a certain degree. Schemas are complex enough to make it almost impossible to fully implement them manually. This complexity also makes code generation based on them tricky, as they use schema features that are not supported by some generators. In addition, some of schemas contain validation errors. Regarding if all of the information included in the schemas are really needed, they have been designed to be useful in as many scenarios as possible. Even if the design process starts with a very well defined use cases, how real users are going to utilise them is not easy to predict.

8.7 Concluding Remarks

In this chapter we have presented an empirical study of actual instances of servers implementing SOS. The study has focused mostly on which parts of the specification are more frequently included in real implementations, and how exchanged messages follow the structure defined by XML Schema files. Several interesting outcomes have been obtained such as the main criteria to group observations into offerings, the small subset of the schemas that are actually used, the large number of files that are invalid according to the schemas, etc.

All of these findings must be taken with care because the study has presented several limitations such as the impossibility to retrieve all of the information published on servers or only the responses from the core profile operations were considered. Nevertheless, they can be of practical use when implementing SOS servers and clients. For example, to decide which parts of the schemas to support, to suggest how to encode large datasets of observations, to know where to look for the sensors location, just to mention some.

CHAPTER 9

Performance Evaluation

In this chapter we present a set of experiments to prove that the main objectives of the code generation process, to reduce generated code size and allow flexible XML parsing, are fulfilled without a significant loss in execution time. In addition to performance tests carried out on a mobile device we include tests targeted to desktop computers to show that the generated code can also be useful for different hardware and software configurations.

9.1 Performance Considerations for Java Programs

Calculating performance of programs in a modern system is a difficult task. The main cause is the existence of multiple factors that may alter the final results, such as cache memories, multi-threading, background processes, etc. For Java programs we have the added complexity that they are not executed directly over the hardware but in a *virtual machine* (VM). In this runtime systems there are additional sources of non-determinism that may affect the overall performance [GBE07] such as *Just-In-Time compilation* (JIT) or garbage collection. All these aspects provoke that different executions of the same program may lead to very different results. For example, Figure 9.1 shows the execution times

---

A version of the content of this chapter has been published as a short paper with the title *"Analysing Performance of XML Data Binding Solutions for SOS Applications"* in *Proceedings of Workshop on Sensor Web Enablement 2011 (SWE 2011)*, October 6-7, Banff, Alberta, Canada.

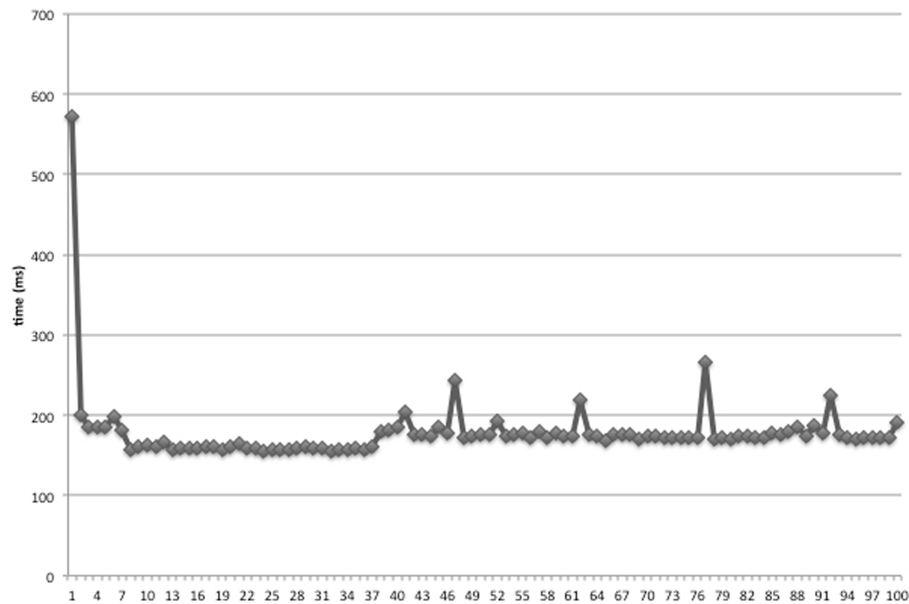

Figure 9.1: Typical execution time pattern followed by a Java program

of 100 iterations of a Java program. The first iteration takes considerably more time than the rest because of the overhead related with class loading and the action of the JIT compiler. The rest of the iterations have a more regular behaviour, but still fluctuates as a result of the many factors that affect execution time.

For this reason, to obtain accurate measurements of the execution time of a Java program, a sound methodology that considers all these situations must be chosen. For our experiments we selected the methodology presented in [GBE07] as it provides a statistically rigorous approach to deal with all these factors.

The selected methodology differentiates between *startup performance* and *steady state performance. Startup performance* measures the execution time of the first run, which as shown before tends to be larger than the rest. *Steady-state performance* refers to the normal action of the application once all of its classes are loaded in memory and, in case it is available, the JIT compiler has also done its job and the application is supposed to run without major interferences from other factors.

The methodology tries to cope with all the non-deterministic factors

that may affect performance measurements. As the number of tests covered here is a bit large we will focus mostly in steady-state performance, although an explanation about how startup performance is calculated is also presented in the following sections.

The methodology uses the following notation for both situations:

- $x_{i,j}$ refers to the measurement of the j-th benchmark iteration of the i-th VM invocation.

9.1.1 Startup Performance

To measure startup performance [GBE07] proposes the following two-step methodology:

1. Measure the execution time of different VM invocations, each VM invocation running a single benchmark iteration. This results in p measurements $x_{i,j}$ with $1 \leq i \leq p$ and $j = 1$.
2. Compute the confidence interval for a given confidence level.

The use of confidence intervals is recommended as they allow determining if differences observed are due to random fluctuations in the measurements or due to actual differences in the alternatives compared against each other. A *confidence interval for the mean* quantifies the range of values that have a given probability (*confidence level*) to include the actual population mean. Depending on whether the number of measurements is large ($n \geq 30$) or small ($n < 30$) the confidence interval is calculated using a *Gaussian distribution* or a *Student's t-distribution.*

9.1.2 Steady-State Performance

To measure steady-state performance [GBE07] proposes a more complex sequence of steps:

1. Consider p VM invocations, each VM invocation running at most q benchmark iterations supposed that we want to retain k measurements per invocation.
2. For each VM invocation i, determine the iteration s_i where steady-state is reached, which is when the *coefficient of variation* (CoV) of the k iterations ($s_i - k$ to s_i) falls below a preset threshold. CoV is the ratio of the standard deviation to the mean.

3. For each VM invocation, compute the mean $\bar{x}_i$ of the k benchmark iterations under steady-state:

$$\bar{x}_i = \sum_{j=s_i-k}^{s_i} x_{i,j}$$

4. Compute the confidence interval for a given confidence level across the computed means from the different VM invocations. The overall mean $\bar{x}$ and the confidence interval is computed over the $\bar{x}_i$ measurements. The formula to calculate $\bar{x}$ is as follows:

$$\bar{x} = \sum_{j=1}^{p} \bar{x}_{i,j}$$

The first step states that for every virtual machine invocation, we want to retain k measurements, where the program is supposed to be in its steady state. The use of different VM invocations is advised as it is documented that different VM invocations may result in different steady-states performances ([AHR02] cited in [GBE07]). The second step reflects the difficulty of determining if the program is actually in its steady-state. As many factors may affect the program execution time, we assume that it has reached a steady-state when for a certain number of consecutive measurements the variability of the execution time is minimal. This variability is calculated through the *coefficient of variation* (standard deviation divided by the mean), which is a normalised measure of dispersion of a probability distribution. The third and fourth steps calculate mean values and confidence intervals for the aggregated results.

9.2 Experimental Setup

Once introduced the detailed methodology to measure performance we present now further details about our specific experimental setup. We attempt to compare the performance of the code generated by DBMobileGen with other XML data binding generators in the following scenarios: a mobile device running Android (Mobile Scenario 1) and two personal computers with different hardware and software configurations (PC Scenarios 1 and 2).

In the first scenario we compare DBMG with XBinder because it is to our best knowledge the only generator available for the Android

Table 9.1: Datasets description

Dataset	Description
Capabilities Dataset (CAPS)	A set of 38 capabilities files. The files are all of the valid capabilities file in SOS Dataset, plus others more files that do not contain critical errors that prevent them to be parsed by all of the generated parsers
Sensor Descriptions Dataset (SD)	A set of 20 sensor descriptions selected from the valid files in SOS Dataset. The files were randomly selected from the available valid sensor descriptions.
Observations Dataset (OBS)	A set of 45 observations files selected from SOS Dataset. This set includes all of the observation files with a size below 10 MB[1] that were processed correctly by all of the parsers, although some of them are invalid.
Measurements Dataset (MEA)	A set of 31 measurements files selected from SOS Dataset. This set includes all of the files with a size below 10 MB that were processed correctly by all of the parsers.

platform that produces Java code. For the other scenarios, DBMG is compared with JAXB, XMLBeans, and again XBinder, which is capable of generating Java code for desktop and server computers as well.

Although the code generated by DBMG is targeted mainly to mobile devices, care has been taken to keep the code compatible to be run on desktop or server Java applications. The reason behind this is that it has not been discarded that it may be also useful in these scenarios. We have also decided to use two different personal computer configurations to illustrate how final results highly depend on the hardware and software configuration.

9.2.1 Test Datasets

To measure generated code performance we will use four datasets. These are based on the dataset presented in the previous chapter, referred to as *SOS Dataset* in the remainder of this chapter, containing real data gathered from different SOS server instances. The test datasets are described in Table 9.1.

Table 9.2: Generated parsers for each dataset

	XMLBeans	JAXB	XBinder (PC)	XBinder (Android)	DBMG
CAPS	X	X	X	X	X
SD	X	X	X	X	X
OBS	X	X	-	-	X
MEA	X	X	X	X	X

It would be ideal to generate a single parser[2] for each selected generator that were capable of processing the four datasets. Unfortunately, we were unable to generate code with XBinder and JAXB from the whole SOS schemas. Although the problems found could maybe be solved adjusting several configuration parameters or modifying the generated code manually, we preferred to use the datasets separately as input to the schema simplification algorithm (Chapter 6). This way four different schema subsets were obtained and were used to produce independent parsers with each generator. This procedure resulted in the generation of 20 different parsers as shown in Table 9.2.

The code generated by XBinder for parsing observations had to be discarded because it was not capable of processing elements defined with type *anyType*. In the case of observation files the element *om:result*[3] is defined with this type with the purpose of acting as a container for different forms of encoding observations.

9.2.2 Hardware and Software

A description of the hardware configurations where the generated parsers will be executed is presented next. Network and storage specifications are not included because the tests have been designed in a way that these parameters do not influence the results, except for the case of storage in the mobile configuration.

[2]The term *parser* is used in this context to refer to the XML processing code generated with the different tools considered here. We use the term *underlying parser* to refer to the vocabulary-independent data access interface used by this code for low-level access to XML data (StaX, DOM, etc.)

[3]Prefix *om* is used for namespace *http://www.opengis.net/om/1.0*

- Mobile Scenario: Mobile Configuration (HTC Desire Phone)
  - Processor: Qualcomm QuadDragon 1 GHz
  - RAM: 576 MB
  - Storage: microSD card 2 Mbit/s transfer speed
  - OS: Android 2.2
- PC Configurations:
  - PC Scenario 1: Apple MacBook Laptop
    * Processor: Intel Core Duo
    * RAM: 4 GB
    * OS: Mac OS X 10.6 64-bit
  - PC Scenario 2: High-End Windows PC
    * Processor: Intel Quad Core i7-860 2.80 GHz
    * RAM: 8 GB DDR3-1333
    * OS: Windows 7 Enterprise 64-bit

An interesting difference between personal computer configurations is that *Mac OS X* uses its own implementation of the Java VM, while *Windows 7* uses the VM distributed by Oracle.

As we are using three different hardware configurations that must be combined with four datasets we will use a single VM invocation ($p = 1$) in all of the tests with the purpose of keeping the exposition of the experiments as simple as possible.

9.3 Results

Because of the great length of the experiment results we only detail here those for the first datasets. The presentation of the results for the other datasets can be found in Appendix B and will be summarised at the end of this chapter.

9.3.1 CAPS Dataset

The capabilities dataset (CAPS) is composed by 38 files taken from *SOS Dataset*. From the 56 files sample, we selected all of the files that could be correctly parsed with the code produced by all of the generators considered in our study. Some of these files were invalid according to the

SOS schemas and required small modifications to make them "parsable" by the generated parsers. The 38 files have sizes ranging from less than 4 KB to 3.5 MB, with a mean size of 315 KB and a standard deviation of 26.7 KB. As the size range is large and with the purpose of simplifying presentation we divide the files in two groups, those with a size below 100 KB, CAPS-S (30 files), and those with size equal to or higher than 100 KB, CAPS-L (8 files).

Mobile Configuration

For the mobile device under consideration the parsers generated by XBinder and DBMG were tested. To avoid possible interferences related with network delays the dataset files were stored locally. In addition, to minimise the interference of data transfer delays from the storage medium all of the files below 500 KB were read into memory before being parsed. It was impossible to do the same for files with sizes above 500 KB because of the device memory restrictions.

The value of CoV selected to control the detection of the steady state for files below 100 KB was 0.05. This value was determined experimentally as we were unable to get values for this variable much smaller than this threshold during a battery of tests executed before the final measures were taken. The value of CoV is calculated over a sample of 30 consecutive measures. For files with size above 100 KB we chose a CoV of 0.02 as the relative variability of the measures was much lower.

Figures 9.2 and 9.3 show the mean execution times in the steady-state for the files in CAPS. In both cases, the execution time of DBMG was better for almost all of the files parsed (35 out of 38). For most of the larger files, DBMG code was about 30% faster than XBinder code, which is a significant difference of 2-3 seconds. The main reason for this is one of those presented in Section 7.4.3 to justify the smaller size of code generated by DBMG, restrictions defined in the schemas for simple types are not enforced in the generated code. As the values of these types are passed by DBMG directly to upper layers of the application, no time is wasted checking for restrictions such as that a number is in predefined interval or a string matches to a regular expression.

Mac OS X Laptop

In the case of the Mac OS X laptop, four different parsers were tested. All of the files were stored locally and loaded into memory before starting

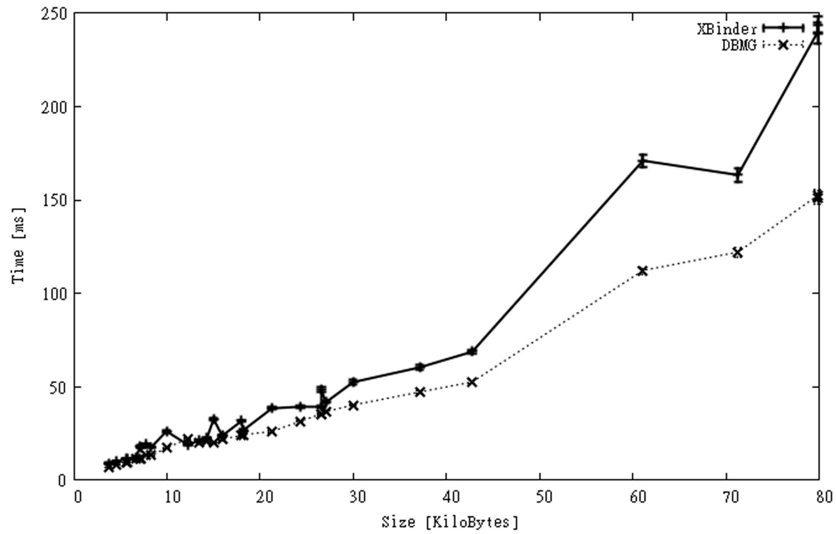

Figure 9.2: Execution times of XBinder and DBMG for CAPS-S dataset (Mobile Scenario)

its processing. This way network and disk speed transfer have no effect over the measured execution times. The value of CoV used to detect the steady state was 0.02.

For this scenario the results obtained by XBinder when compared to DBMG were worse than for the previous one (Figures 9.4 and 9.5). Again, code generated with DBMG was faster for 35 of the 38 files, but the differences were larger being more than 60% faster in some cases. For small files (<30 KB), the DBMG parser was even faster than those generated by JAXB and XMLBeans. For larger files, except for XBinder that had worse results, the rest of the parsers had very similar execution times, with some difference in favour of JAXB.

The worst performance shown by XBinder can be justified again because all of the validation code included in the generated parser. Nevertheless, XMLBeans generates similar validation code and it is as fast as JAXB and DBMG. A possible reason for this is that XMLBeans does not transform all of the data contained in XML documents into application objects during parsing. Instead, it creates an *XML Store*, an in-memory structure with the XML data, and then creates object instances containing portions of this data on user request.

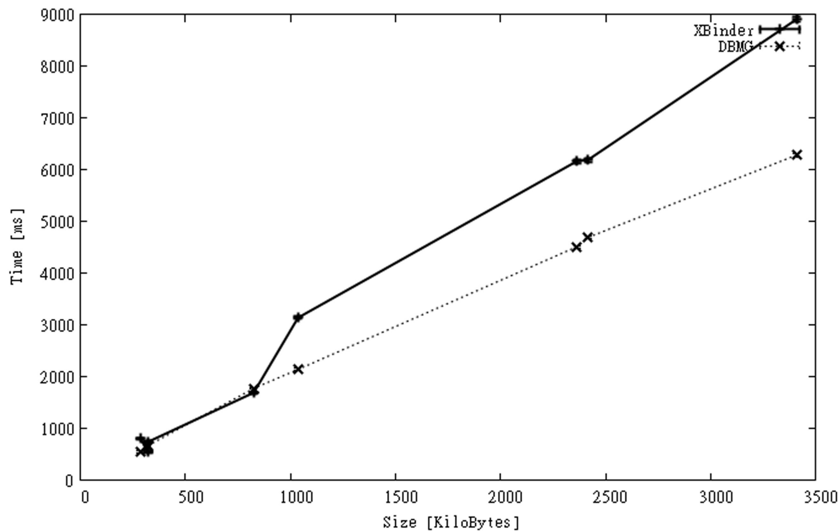

Figure 9.3: Execution times of XBinder and DBMG for CAPS-L dataset (Mobile Scenario)

Windows PC

Similarly to the previous case, in the high-end Windows PC four parsers were tested. Processed files were loaded into memory before parsing. For this test the results obtained by XBinder and DBMG, differing from the previous experiments, were very similar (Figures 9.6 and 9.7). The execution times for JAXB and XMLBeans continued to be the best and were very close to each other for small files . For larger files JAXB had a performance about 30% better than XMLBeans .

The cause of DBMG generated code being as slow as XBinder can be found in the underlying parser. An additional experiment shown in Appendix C has shown that execution times of the StAX parser on which XBinder code is based is a lot faster than KXml. It is important to note the KXml is a parser optimised to be executed in resource-constrained devices. For this reason, it does not use sophisticated data structures or algorithms that might speed up parsing at the expense of consuming more computational resources.

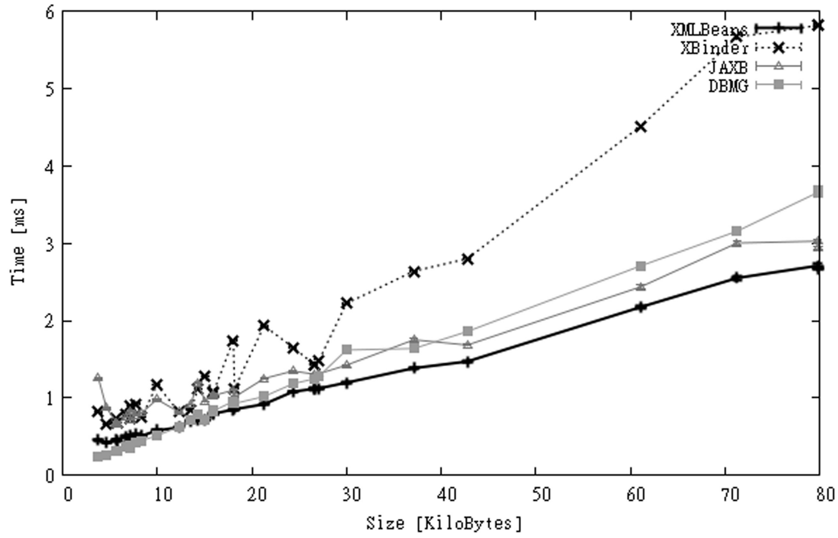

Figure 9.4: Execution times for CAPS-S dataset (PC Scenario 1 - Mac OS X Laptop)

9.4 Discussion

The results presented in the previous section show that in the mobile scenario, DBMG not only generates code with smaller footprint, but this code also has better execution times for the CAPS dataset than XBinder. In other scenarios DBMG code is at least as good as that of XBinder, and in some cases being as fast as more powerful tools such as JAXB and XMLBeans.

The differences in the relative comparison of the generated code for different scenarios show that external factors, such as hardware or VM implementation, have a large influence over these results. These factors might provoke that these differences could be enlarged or shortened when the configuration is changed.

As mentioned before, the results for the rest of the datasets are presented in Appendix B. Nevertheless, a summary of these results relevant to our discussion is presented in the following paragraphs.

The SD dataset is characterised for including very small files ($< 20KB$). The measures for this dataset show a slight advantage of XBinder

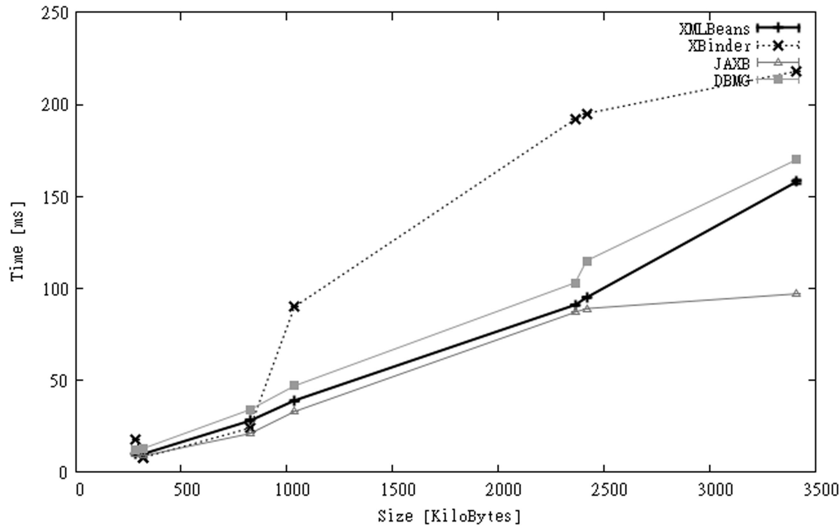

Figure 9.5: Execution times for CAPS-L dataset (PC Scenario 1 - Mac OS X Laptop)

over DBMG for the mobile configuration. When executed in a more capable hardware the measures for all of the parsers show a high relative variability, which could be caused by the fact that most measured values are below 1 millisecond, and at this time resolution any minimal external disturbance may affect the measurement process. In any case, as the time taken to process these files is so small for PC scenarios we consider these differences negligible.

On the other hand, the OBS dataset is characterised for having about half of the files with sizes above 1 MB. Unfortunately, the code generated by XBinder is not capable of processing correctly this data. For this reason, we only included tests for the personal computer configurations. In these scenarios, for small files, XMLBeans had an important advantage over JAXB and DBMG, which showed similar results. For large files, the figures for XMLBeans and JAXB were almost identical.

Last, the MEA dataset, where most of the files were above 2 MB, showed similar results for XBinder and DBMG for the first two scenarios, but XBinder was better in the third one. For the other generators, JAXB tended to be better as the size of the files increased.

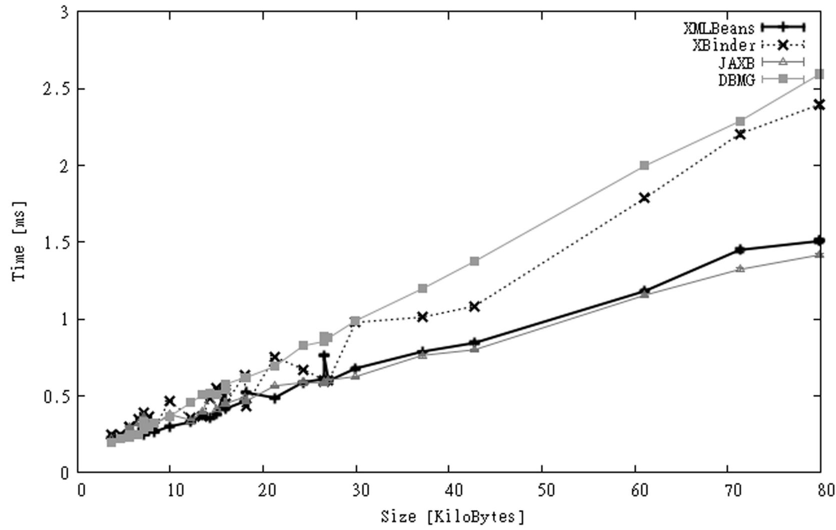

Figure 9.6: Execution times for CAPS-S dataset (PC Scenario 2 - Windows PC)

9.5 Concluding Remarks

The experiments presented in this chapter have shown that the code generated by DBMG is faster that the one produced by XBinder in most of the test cases. DBMG code has the added value that can be compiled unmodified for mobile and desktop systems. The tests have also shown that the hardware and software components conforming a given system have a large influence over the relative performance of parsers. This means that if a parser is faster than other for a given configuration, this does not imply that it will be faster if the configuration is changed.

The performance of JAXB and XMLBeans was better than the performance of DBMG in most of the test cases. But, experiments measuring the performance of underlying parsers suggest that these differences can be substantially shortened if this parser were changed for PC scenarios.

The selection of which generator would be more adequate for a given application may require the consideration of other factors not covered in this document such as memory consumption, support for certain XML Schema features or the possibility of merging generated code with existing business logic code. In the case of our research, the priority was to make the size of parser binaries as small as possible, because existing

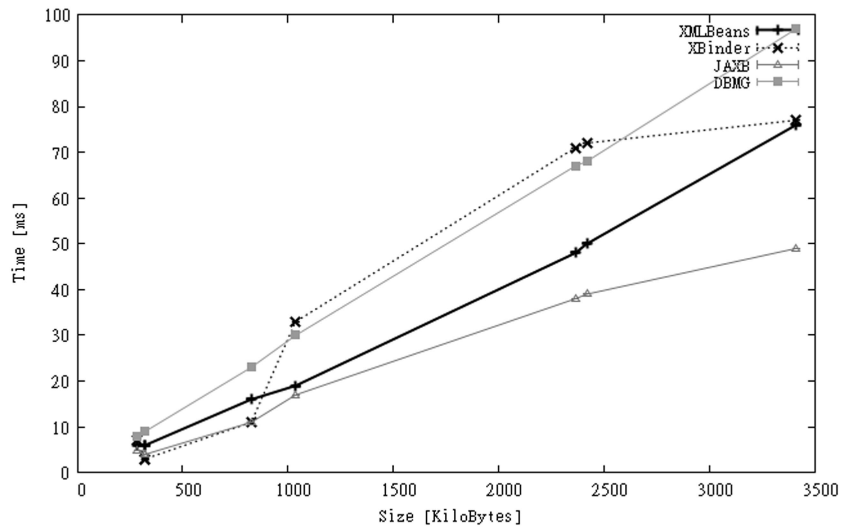

Figure 9.7: Execution times for CAPS-L dataset (PC Scenario 2 - Windows PC)

tools generate binaries that are excessively large to be executed in mobile phones. Still, we wanted to achieve our main goal without sacrificing the parsers performance, which has been demonstrated by the experiments presented here.

Part V

Conclusions and Future Work

CHAPTER 10

Conclusions

While geospatial applications are becoming commonplace in mobile devices, the number of these applications that are based on OWS specifications is really low. In our opinion, the complexity of the protocols defined by the standards makes the task of writing reliable and efficient mobile applications based on them an arduous task. These protocols are based on complex data structures exchanged in XML documents, which are reputed as being too verbose, provoke that they cannot be efficiently processed in mobile devices with still limited capabilities. In this dissertation we have tried to find ways to overcome the complexity of the communication protocols to make them suitable for mobile devices. We detailed next the contributions of our research, followed by a description of future research lines that could follow the results obtained so far.

10.1 Contributions

The work presented in this dissertation includes several contributions. The first one is a comprehensive complexity study for the OGC specification schemas presented in Chapter 5. The study uses a set of metrics to measure different aspects of complexity of schemas and offers a quantitative way to analyse and measure the complexity of OWS schemas. The use of adequate metrics allows us to quantify the complexity and other properties of the schemas. The results of the analysis have shown that at least half of the presented specifications can be considered as large and complex according to all of the metrics included in our study. The metrics also allowed to group specifications according to their complexity. A

set of new metrics was proposed to show different views of the effect of the use of subtyping mechanism on complexity. We have also presented use case scenarios where metrics could be applied. The metric set presented here should not be seen as a closed set, many other metrics can be useful in many different scenarios. Some of the potential uses of metrics are evaluating the impact of design decisions, assessing the effectiveness of different solutions to deal with schemas complexity, and to detect potential design problems such as components with too many information items, or excessively deep subtyping hierarchies.

The second contribution is an approach to generate XML processing code for OGC-based applications targeted to mobile devices. The approach, named *Instance-based XML data binding code generation*, is composed first of an algorithm that allows the simplification of specification schemas according to the needs of particular applications (Chapter 6), and a code generator targeted to the Android Platform (Chapter 7). The algorithm output is used by the code generator to produce Java code. The *schema simplification algorithm* works based on the assumption that a representative subset of the XML instances that must be manipulated by the application is available and that actual applications implementing OWS specifications use only portions of their associated schemas. Results of applying the algorithm to a real-world use case scenario showed that it allows a substantial reduction of the original schema set of about 90% of its size. This huge reduction in schema size is translated into a reduction of generated binary code of more than 80% of its size for an SOS client targeted to the Android platform. The transformation performed by this algorithm is done at the schema level and no assumption about the target platform is made and, as a consequence, the output schemas can be combined with any other available XML data binding code generator.

In order to prove that actual applications use only portions of the schemas and to quantify how much of the schemas is used for actual implementation, we presented an empirical study of actual instances of servers implementing SOS in Chapter 8. The study focused mostly on which parts of the specification are more frequently included in real implementations, and how exchanged messages follows the structure defined by XML Schema files. Several interesting outcomes have been obtained such as the main criteria to group observations into offerings, the small subset of the schemas that are actually used, the large number of files that are invalid according to the schemas, etc.

The code generator, named *DBMobileGen*, utilises the information about how XML documents make use of its associated schemas to refine

the generated code with the aim of reducing its final binary size. This tool offers an interesting number of features that allow the production of very compact code, and its effectiveness has been demonstrated through a set of experiments and by building two sample applications using the generator. These experiments have shown that code generated by *DBMobileGen* for the case study is substantially smaller than code generated by other tools. If we compare this tool directly with XBinder, the only generator known by the authors producing code for Android, we can say that it generates much more compact code for the analysed scenario (almost 6 times smaller) and provides more flexibility to parse potentially invalid XML documents. The experiments presented in Chapter 9 showed that code generated by DBMG is faster that the one produced by XBinder in most of the test cases. The tests have also showed that the hardware and software components conforming a given system have a large influence over the relative performance of parsers. This means that if a parser is faster than others for a given configuration, this does not imply that it will be faster if the configuration is changed.

10.2 Future Work

As future work in the topic of OWS implementations for mobile devices we are following several lines of research derived from the results presented in this document. The first line is related with the exploration of alternative formats to exchange information between clients and servers. The use of XML to encode the information adds an overhead that could be unbearable for some resource-constrained devices.

Another research line is related with the generalisation of the results presented for SOS and WPS to the rest of the OGC specifications. The sample applications built as demonstration of our approach had a lot in common, presenting similar challenges and open issues that could be further explored in the context of other specifications. Some of the open issues such as the use of code generation techniques for sections of the applications in the business logic and user interface layer of the applications can potentially reduce the complexity of building these applications. We are currently planning to build a framework for OGC-based client development which will include communication libraries for several specifications, the possibility of customising XML processing code as well as a set of common user interface components that could be useful to application developers.

Last, the topic of complexity of schemas can also be further explored. New metrics can be considered as well as a more precise definition of which interesting aspect of the schemas can be measured by which metrics. It can be also questioned if XML Schema is the most appropriate option to specify the structure of the exchanged information.

Bibliography

[3GP] 3GPP. GSM 03.60, General Packet Radio Service (GPRS); Service description; Stage 2. Available from: `http://www.3gpp.org/ftp/Specs/html-info/0360.htm`.

[AHR02] M. Arnold, M. Hind, and B. G. Ryder. Online feedback-directed optimization of java. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '02, pages 111–129, New York, NY, USA, 2002. ACM.

[AIM+04] P. Apparao, R. Iyer, R. Morin, N. Nayak, M. Bhat, D. Halliwell, and W. Steinberg. Architectural Characterization of an XML-Centric Commercial Server Workload. In *Proceedings of the 2004 Int. Conference on Parallel Processing*, ICPP '04, pages 292–300, Washington, DC, USA, 2004. IEEE Computer Society.

[AM96] F. Brito Abreu and W. Melo. Evaluating the Impact of Object-Oriented Design on Software Quality. *IEEE International Symposium on Software Metrics*, 0:90, 1996.

[Ant11] G. Anthes. Invasion of the mobile apps. *Commun. ACM*, 54:16–18, September 2011.

[Apa] Apache Software Foundation. XMLBeans. Available from: `http://xmlbeans.apache.org/`.

[App] Apple. iOS 4.3 Software Update. Available from: `http://www.apple.com/ios/`.

[BA03] K. Barr and K. Asanović. Energy aware lossless data compression. In *Proceedings of the 1st international conference on Mobile systems, applications and services*, MobiSys '03, pages 231–244, New York, NY, USA, 2003. ACM.

[Bar11] B. Barkstrom. When is it sensible not to use xml? *Earth Science Informatics*, 4:45–53, 2011. 10.1007/s12145-010-0063-2.

[BCG+05] B. Benatallah, F. Casati, D. Grigori, H. Nezhad, and F. Toumani. Developing adapters for web services integration. In Oscar Pastor and João Falcão e Cunha, editors, *Advanced Information Systems Engineering*, volume 3520 of *Lecture Notes in Computer Science*, pages 415–429. Springer Berlin / Heidelberg, 2005.

[Bea09] J. Bean. *SOA and Web Services Interface Design: Principles, Techniques, and Standards (The MK/OMG Press)*. Morgan Kaufmann, 2009.

[BEJ+11] A. Bröering, J. Echterhoff, S. Jirka, I. Simonis, T. Everding, C. Stasch, S. Liang, and R. Lemmens. New Generation Sensor Web Enablement. *Sensors*, 11(3):2652–2699, 2011.

[BEMW08] H. J. Bungartz, W. Eckhardt, M. Mehl, and T. Weinzierl. DaStGen– A Data Structure Generator for Parallel C++ HPC Software. In *Proceedings of the 8th international conference on Computational Science, Part III*, ICCS '08, pages 213–222, Berlin, Heidelberg, 2008. Springer-Verlag.

[BEWZ10] H. J. Bungartz, W. Eckhardt, T. Weinzierl, and C. Zenger. A precompiler to reduce the memory footprint of multiscale pde solvers in c++. *Future Generation Computer Systems*, 26:175–182, January 2010.

[BL06] T. Berners-Lee. Linked data, 2006. Available from: `http://www.w3.org/DesignIssues/LinkedData.html`.

[BLS00] D. Beyer, C. Lewerentz, and F. Simon. Impact of inheritance on metrics for size, coupling, and cohesion in object-oriented systems. In *Proceedings of the 10th International Workshop on New Approaches in Software Measurement*, IWSM '00, pages 1–17, London, UK, 2000. Springer-Verlag.

[BM09a] D. Basci and S. Misra. Data Complexity Metrics for XML Web Services. *Advances in Electrical and Computer Engineering*, 9(2):9–15, 2009.

[BM09b] D. Basci and S. Misra. Measuring and evaluating a design complexity metric for XML schema documents. *Journal of Information Science and Engineering*, 25(5):1405–1425, September 2009.

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley, 1996.

[BMV05] D. Barbosa, L. Mignet, and P. Veltri. Studying the XML Web: Gathering Statistics from an XML Sample. *World Wide Web*, 8:413–438, 2005.

[Bra03] T. Bray. XML Is Too Hard For Programmers, 2003. Available from: `http://www.tbray.org/ongoing/When/200x/2003/03/16/XML-Prog`.

[CCL09] C. Canali, M. Colajanni, and R. Lancellotti. Performance Evolution of Mobile Web-Based Services. *IEEE Internet Computing*, 13:60–68, March 2009.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.

[Cod] Code Synthesis. XSD/e: : XML for Light-Weight C++ Applications. Available from: `http://codesynthesis.com/products/xsde/`.

[CRC+06] C. B. Chirila, M. Ruzsilla, P. Crescenzo, D. Pescaru, and E. Tundrea. Towards a reengineering tool for java based on reverse inheritance. In *In Proceedings of SACI 2006 the 3-rd Romanian-Hungarian Joint Symposium on Applied Computational Intelligence*, pages 963–7154, 2006.

[CREP08] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. Automating co-evolution in model-driven engineering. In *Proceedings of the 2008 12th International IEEE Enterprise Distributed Object Computing Conference*, pages 222–231, Washington, DC, USA, 2008. IEEE Computer Society.

[Das01] E. M. Dashofy. Issues in Generating Data Bindings for an XML Schema-Based Language. In *In Proceedings of the Workshop on XML Technologies and Software Engineering (XSE2001*, 2001.

[dee] Degree Homepage. Available from: http://deegree.org/.

[Deu96] P. Deutsch. GZIP file format specification version 4.3, 1996. Available from: `http://www.gzip.org/zlib/rfc-gzip.html`.

[DKDF09] C. A. Davis, Y. Kimo, and F. L. P. Duarte-Figueiredo. OGC Web Map Service Implementation Challenges for Mobile Computers. pages 1–6, August 2009.

[ESR98] ESRI. ESRI Shapefile Technical Description. *White Paper*, 1998.

[FC05] P. Farley and M. Capp. Mobile Web Services. *BT Technology Journal*, 23:202–213, July 2005.

[Fie00] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, University of California, Irvine, Irvine, California, 2000.

[FMLPNI10] A.J. Florczyk, P. Maué, F.J. López-Pellicer, and J. Nogueras-Iso. Finding OGC Web Services in the Digital Earth. In *Proceedings of the Workshop Towards Digital Earth: Search, Discover and Share Geospatial Data at Future Internet Symposium, Berlin, Germany*, 2010.

[FN99] N. E. Fenton and M. Neilf. Software Metrics: Success, Failures and New Directions. *J. Syst. Softw.*, 47:149–157, July 1999.

[Gar11] Gartner Inc. Gartner Says Worldwide Mobile Device Sales to End Users Reached 1.6 Billion Units in 2010; Smartphone Sales Grew 72 Percent in 2010, 2011. Available from: `http://www.gartner.com/it/page.jsp?id=1543014`.

[GBE07] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 57–76, New York, NY, USA, 2007. ACM.

[geo] Geoserver Homepage. Available from: `http://geoserver.org/`.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, Boston, MA, 1995.

[Goo] Google. Android.com. Available from: `http://www.android.com/`.

[gvsa] gvSIG Portal. Available from: `http://www.gvsig.gva.es`.

[gvsb] gvSIG Portal. Available from: `http://www.gvsig.org/web/home/projects/gvsig-mobile`.

[GYC07] M. F. Goodchild, M. Yuan, and T. J. Cova. Towards a general theory of geographic representation in gis. *Int. J. Geogr. Inf. Sci.*, 21:239–260, January 2007.

[Hau] S. Haustein. kXML. Available from: `http://kxml.sourceforge.net/`.

[Hay09] R. B. Hayun. *Java ME on Symbian OS: Inside the Smartphone Model*. Wiley Publishing, 2009.

[Her03] J. Herrington. *Code Generation in Action*. Manning Publications Co., Greenwich, CT, USA, 2003.

[HKM+10] Sayed Y. Hashimi, Satya Komatineni, Dave MacLean, Sayed Y. Hashimi, Satya Komatineni, and Dave MacLean. *Pro Android 2*. Apress, 2010.

[Hos10] H. Hosoya. *Foundations of XML Processing: The Tree-Automata Approach*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 8RU, UK, 2010.

[HS] S. Haustein and A. Slominski. XMLPull API. Available from: `http://www.xmlpull.org`.

[IEE07] IEEE Standards Asociation. 802.11-2007 - IEEE Standard for Local and Metropolitan Area Networks - Specific Requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications, 2007. Available from: `http://standards.ieee.org/findstds/standard/802.11-2007.html`.

[IET99] W3C IETF. Hypertext Transfer Protocol – HTTP/1.1, 1999. Available from: http://tools.ietf.org/html/rfc2616/.

[ISO86] ISO. Information processing – Text and office systems – Standard Generalized Markup Language (SGML), 1986. Available from: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=16387.

[Jav04] Java Community Process. JSR 173: Streaming API for XML, 2004. Available from: http://jcp.org/en/jsr/detail?id=173.

[Jav06] Java Community Process. JSR 222: Java Architecture for XML Binding (JAXB) 2.0, 2006. Available from: http://www.jcp.org/en/jsr/detail?id=222.

[JAX] JAXB Project. JAXB Reference Implementation. Available from: http://jaxb.java.net/.

[Kay03] Michael H. Kay. XML five years on: a review of the achievements so far and the challenges ahead. In *Proceedings of the 2003 ACM symposium on Document engineering*, DocEng '03, pages 29–31, New York, NY, USA, 2003. ACM.

[KL09] Yeon-Seok Kim and Kyong-Ho Lee. A lightweight framework for mobile web services. *Computer Science - Research and Development*, 24:199–209, 2009. 10.1007/s00450-009-0091-7.

[KLT07] Jaakko Kangasharju, Tancred Lindholm, and Sasu Tarkoma. XML Messaging for Mobile Devices: From Requirements to Implementation. *Comput. Netw.*, 51:4634–4654, November 2007.

[KMB04] Cem Kaner, Senior Member, and Walter P. Bond. Software Engineering Metrics: What Do They Measure and How Do We Know? In *In METRICS 2004. IEEE CS*. Press, 2004.

[Kra07] Athanasios Tom Kradilis. Geospatial Web Services: The Evolution of Geospatial Data Infrastructure. *The Geospatial Web: How Geobrowsers, Social Software and the*

Web 2.0 are Shaping the Network Society, Scharl A. and Tochtermann K. (Eds), Springer London, pages 223–228, 2007.

[KTL05] Jaakko Kangasharju, Sasu Tarkoma, and Tancred Lindholm. Xebu: A binary format with schema-based optimizations for XML data. In *6th International Conference on Web Information Systems Engineering, volume 3806 of Lecture Notes in Computer Science*, pages 528–535. Springer-Verlag. Short, 2005.

[Kuh09] Werner Kuhn. A Functional Ontology of Observation and Measurements. In *Proceedings of the 3rd International Conference on GeoSpatial Semantics*, GeoS '09, pages 26–43, Berlin, Heidelberg, 2009. Springer-Verlag.

[LBT04] R. Lake, D. S. Burggraf, and M. Trninic. *Geography markup language GML: foundation for the geo-web*. John Wiley and Sons, New Jersey, 2004.

[LCT05] Steve H. L. Liang, Arie Croitoru, and C. Vincent Tao. A distributed geospatial infrastructure for Sensor Web. *Computer and Geosciences*, 31:221–231, March 2005.

[LDL08] Tak Cheung (Brian) Lam, Jianxun Jason Ding, and Jyh-Charn Liu. XML Document Parsing: Operational and Performance Characteristics. *Computer*, 41:30–37, 2008.

[LF07] Ron Lake and Jim Farley. Infrastructure for the Geospatial Web. *The Geospatial Web: How Geobrowsers, Social Software and the Web 2.0 are Shaping the Network Society, Scharl A. and Tochtermann K. (Eds), Springer London*, pages 15–26, 2007.

[LK08] Tancred Lindholm and Jaakko Kangasharju. How to edit gigabyte XML files on a mobile phone with XAS, RefTrees, and RAXS. In *Proceedings of the 5th Annual International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services*, Mobiquitous '08, pages 50:1–50:10, 2008.

[LKR05] R. Lämmel, Stan Kitsis, and D. Remy. Analysis of XML schema usage. In *Proceedings of XML 2005*, pages 1–35, 2005.

[LM07] Ralf Lämmel and Erik Meijer. Revealing the x/o impedance mismatch: changing lead into gold. In *Proceedings of the 2006 international conference on Datatype-generic programming*, SSDGP'06, pages 285–367, Berlin, Heidelberg, 2007. Springer-Verlag.

[LPBHF+10] F.J. López-Pellicer, R. Béjar-Hernández, A. J. Florczyk, P .R. Muro-Medrano, and F. J. Zarazaga-Soria. State of Play of OGC Web Services across the Web. In *Proceedings of INSPIRE Conference 2010: INSPIRE as a framework for cooperation, Krakow, Poland*, 2010.

[LSZ09] Giovanni Lagorio, Marco Servetto, and Elena Zucca. Flattening versus direct semantics for featherweight jigsaw. In *FOOL'09, International Workshop on Foundations of Object Oriented Languages*. ACM Press, 2009.

[Mac10] MacWorld. Apple inside: the significance of the iPad's A4 chip, 2010. Available from: `http://www.macworld.com/article/145998/2010/01/apple%20a4.html?lsrc=rss%20main`.

[map] MapServer Homepage. Available from: `http://mapserver.org/`.

[McC76] T.J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.

[McL02] Brett McLaughlin. *Java and XML Data Binding*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[MD01] Tom Mens and Serge Demeyer. Future Trends in Software Evolution Metrics. In *Proceedings of the 4th International Workshop on Principles of Software Evolution*, IWPSE '01, pages 83–86, New York, NY, USA, 2001. ACM.

[Mic] Microsoft. Windows Phone 7. Available from: `http://www.microsoft.com/windowsphone/`.

[MNSB06] Wim Martens, Frank Neven, Thomas Schwentick, and Geert Jan Bex. Expressiveness and Complexity of XML Schema. *ACM Transactions on Database Systems*, 31:770–813, September 2006.

[MS06] Anders Mller and Michael I. Schwartzbach. *An Introduction to Xml And Web Technologies*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[MSY05] A. McDowell, C. Schmidt, and K. Yue. Analysis and Metrics of XML Schema. In *Proceedings of Intl Conference on Software Engineering Research and Practice*, pages 538–544, 2005.

[Nev02] Frank Neven. Automata Theory for XML Researchers. *SIGMOD Rec.*, 31:39–46, September 2002.

[NJ03] Matthias Nicola and Jasmi John. Xml parsing: a threat to database performance. In *Proceedings of the twelfth international conference on Information and knowledge management*, CIKM '03, pages 175–178, New York, NY, USA, 2003. ACM.

[Nok] Nokia. Symbian at Nokia. Available from: `http://symbian.nokia.com/`.

[NR00] Mike Ruth Niles Ritter. GeoTIFF Format Specification GeoTIFF Revision 1.0, 2000. Available from: `http://www.remotesensing.org/geotiff/spec/geotiffhome.html`.

[Oba95] Dare Obasanjo. Why You Should Very Carefully Use Restriction Of Complex Types, 1995. Available from: `http://www.xml.com/pub/a/2002/11/20/schemas.html?page=4#restriction`.

[Obj] Objective Systems, Inc. XBinder XML Schema Compiler. Available from: `http://jaxb.java.net/`.

[OdP09] Guadalupe Ortiz and Alfonso Garcia de Prado. Mobile-Aware Web Services. *Mobile Ubiquitous Computing, Systems, Services and Technologies, International Conference on*, 0:65–70, 2009.

[OGCa] OGC. Open Geospatial Consortium. Available from: `http://www.opengeospatial.org/`.

[OGCb] OGC. Registered Products. Available from: `http://www.opengeospatial.org/resource/products`.

[OGC04] OGC. OpenGIS Geography Markup Language (GML) Implementation Specification 3.1.1. *OGC Document*, (03-105r1), 2004.

[OGC05a] OGC. GML 3.1.1 common CRSs profile. *OGC Document*, (05-095r1), 2005.

[OGC05b] OGC. GML 3.1.1 CRS support profile. *OGC Document*, (05-094r1), 2005.

[OGC06a] OGC. Geography Markup Language (GML) simple features profile. *OGC Document*, (06-049r1), 2006.

[OGC06b] OGC. OGC Sensor Alert Service Candidate Implementation Specification 0.9. *OGC Document*, (06-028r3), 2006.

[OGC06c] OGC. OpenGIS Web Mapping Server Implementation Specification 1.3.0. *OGC Document*, (06-042), 2006.

[OGC07a] OGC. Observations and Measurements - Part 1 - Observation schema. *OGC Document*, (07-022r1), 2007.

[OGC07b] OGC. OGC Web Services Common Specification 1.1.0. *OGC Document*, (06-121r3), 2007.

[OGC07c] OGC. OpenGIS Sensor Model Language (SensorML) Implementation Specification 1.0.0. *OGC Document*, (07-000), 2007.

[OGC07d] OGC. OpenGIS Sensor Planning Service Implementation Specification 1.0.0. *OGC Document*, (07-014r3), 2007.

[OGC07e] OGC. OpenGIS Transducer Markup Language (TML)Implementation Specification 1.0.0. *OGC Document*, (06-010r6), 2007.

[OGC07f] OGC. OpenGIS Web Notification Service Implementation Specification. 0.0.9 . *OGC Document*, (06-095), 2007.

[OGC07g] OGC. OpenGIS Web Processing Service 1.0.0. *OGC Document*, (05-007r7), 2007.

[OGC07h] OGC. Sensor Observation Service 1.0.0. *OGC Document*, (06-009r6), 2007.

[OGC08a] OGC. OGC KML. *OGC Document*, (07-147r2), 2008.

[OGC08b] OGC. OGC Sensor Web Enablement: Overview And High Level Architecture. *OGC Whitepaper,*, 2008.

[OGC08c] OGC. OpenGIS Sensor Event Service Interface Specification 0.3.0. *OGC Document*, (08-133), 2008.

[OGC08d] OGC. Wrapping OGC HTTP-GET/POST Services with SOAP. *OGC Discussion Paper*, (07-158), 2008.

[OGC10a] OGC. OGC WCS 2.0 Interface Standard - Core. *OGC Document*, (09-110r3), 2010.

[OGC10b] OGC. OGC Web Services Common Standard version 2.0.0. *OGC Document*, (06-121r9), 2010.

[OGC10c] OGC. OpenGIS Web Feature Service 2.0 Interface Standard. *OGC Document*, (09-025r1), 2010.

[OGC10d] OGC. SOS 2.0Get Data Availability Extension. *Candidate Standard, OGC Document*, (10-167), 2010.

[OGC11] OGC. OGC SWE Common Data Model Encoding Standard. *OGC Document*, (08-094r1), 2011.

[ope] OpenJUMP. Available from: `http://www.openjump.org/`.

[Oraa] Oracle. Connected Limited Device Configuration (CLDC); JSR 139. Available from: `http://java.sun.com/products/cldc/`.

[Orab] Oracle. Java ME Technology. Available from: `http://www.oracle.com/technetwork/java/javame/tech/index.html`.

[Pap08] Michael P. Papazoglou. *Web Services: Principles and Technology.* Pearson, Prentice Hall, 2008.

[Pas06] James Pasley. Avoid XML Schema Wildcards For Web Service Interfaces. *IEEE Internet Computing*, 10:72–79, May 2006.

[Per10] George Percivall. Progress in OGC Web Services Interoperability Development. In Liping Di and H. K. Ramapriyan, editors, *Standard-Based Data and Information Systems for Earth Observation*, Lecture Notes in Geoinformation and Cartography, pages 37–61. Springer Berlin Heidelberg, 2010.

[PK10] Kolin Paul and Tapas Kumar Kundu. Android on mobile devices: An energy perspective. *International Conference on Computer and Information Technology*, 0:2421–2426, 2010.

[Pro] Prodevelop SL. gvSIG Mini. Available from: `https://confluence.prodevelop.es/display/GVMN/Home`.

[Pro08] Florian Probst. Observations, measurements and semantic reference spaces. *Appl. Ontol.*, 3:63–89, January 2008.

[PS05] Joseph A. Paradiso and Thad Starner. Energy scavenging for mobile and wireless electronics. *IEEE Pervasive Computing*, 4:18–27, January 2005.

[PSH10] Christian Pichler, Michael Strommer, and Christian Huemer. Size Matters!? Measuring the Complexity of XML Schema Mapping Models. *Services, IEEE Congress on*, 0:497–502, 2010.

[PZL08] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful Web Services vs. "Big"' Web Wervices: Making the Right Architectural Decision. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 805–814, New York, NY, USA, 2008. ACM.

[QS05] Mustafa H. Qureshi and M. H. Samadzadeh. Determining the Complexity of XML Documents. In *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume II - Volume 02*, ITCC '05, pages 416–421, Washington, DC, USA, 2005. IEEE Computer Society.

[Qua] Qualcomm. Mobile Processors - Snapdragon - Qualcomm. Available from: `http://www.qualcomm.com/snapdragon`.

[Ref] Refraction Research. uDig: User-friendly Desktop Internet GIS. Available from: `http://udig.refractions.net/`.

[RIM] RIM. Blackberry OS 6. Available from: `http://us.blackberry.com/apps-software/blackberry6/`.

[RR07] Leonard Richardson and Sam Ruby. *RESTful Web Services*. O'Reilly, Beijing, 2007.

[SAX] SAX Project. SAX. Available from: `http://www.saxproject.org/`.

[SGD10] S. Schade, C. Granell, and L. Díaz. Augmenting SDI with Linked Data. In *Proceedings of the Workshop on Linked Spatiotemporal Data 2010 (LSTD 2010)*, 2010.

[SNMRRP07] E. Sánchez-Nielsen, S. Martín-Ruiz, and J. Rodríguez-Pedrianes. Mobile and Dynamic Web Services. In M. Calisti, M. Walliser, S. Brantschen, M. Herbstritt, C. Pautasso, and C. Bussler, editors, *Emerging Web Services Technology*, Whitestein Series in Software Agent Technologies and Autonomic Computing, pages 117–133. 2007.

[Tam09] A. Tamayo. gvSOS: A New Client for OGC SOS Interface Standard. *Master Thesis, Universitat Jaume I, Castellón de la Plana, Spain*, 2009.

[TGD+11] A. Tamayo, C. Granell, L. Díaz, M. Gould, and J. Huerta. Client-side processing for Sensor Web. In Paulo Alencar and Donald Cowan, editors, *Handbook of Research on Mobile Software Engineering: Design Implementation and Emergent Applications*. IGI Global, 2011.

[TGH11a] A. Tamayo, C. Granell, and J. Huerta. Analysing complexity of XML schemas in geospatial web services. In *Proceedings of the 2nd International Conference on Computing for Geospatial Research & Applications*, COM.Geo '11, pages 17:1–17:9, New York, NY, USA, 2011. ACM.

[TGH11b] A. Tamayo, C. Granell, and J. Huerta. Dealing with large schema sets in mobile SOS-based applications. In *Proceedings of the 2nd International Conference on Computing for Geospatial Research & Applications*, COM.Geo '11, pages 16:1–16:9, New York, NY, USA, 2011. ACM.

[THG+09] A. Tamayo, J. Huerta, C. Granell, L. Díaz, and R. Quiros. gvSOS: A New Client for the OGC Sensor Observation Service Interface Standard. *Transactions in GIS*, 13(s1):47–61, June 2009.

[TKLR06] S. Tarkoma, J. Kangasharju, T. Lindholm, and K. Raatikainen. Fuego: Experiences with Mobile Data Communication and Synchronization. In *Personal, Indoor and Mobile Radio Communications, 2006 IEEE 17th International Symposium on*, pages 1 –5, sept. 2006.

[TVGH11] A. Tamayo, P. Viciano, C. Granell, and J. Huerta. Empirical study of sensor observation services server instances. In Stan Geertman, Wolfgang Reinhardt, and Fred Toppen, editors, *Advancing Geoinformation Science for a Changing World*, volume 1 of *Lecture Notes in Geoinformation and Cartography*, pages 185–209. Springer Berlin Heidelberg, 2011.

[TVN+03] M. Tian, T. Voigt, T. Naumowicz, H. Ritter, and J. Schiller. Performance Considerations for Mobile Web Services. *Elsevier Computer Communications Journal*, 27:1097–1105, 2003.

[vB09] C. H. (Kees) van Berkel. Multi-core for mobile phones. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 1260–1265, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[VEG02] Robert A. Van Engelen and Kyle A. Gallivan. The gSOAP Toolkit for Web Services and Peer-to-Peer Computing Networks. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, CCGRID '02, pages 128–, Washington, DC, USA, 2002. IEEE Computer Society.

[Vis06] J. Visser. Structure Metrics for XML Schema. In *Proceedings of XATA 2006*, pages 236–247, 2006.

[vZSM08] T. L. van Zyl, I. Simonis, and G. Mcferren. The Sensor Web: systems of sensor systems. *International Journal of Digital Earth*, 99999(1):1–14, 2008.

[W3C99a] W3C. WAP Binary XML Content Format, 1999. Available from: http://www.w3.org/TR/wbxml/.

[W3C99b] W3C. XML Path Language (XPath) Version 1.0, 1999. Available from: http://www.w3.org/TR/xpath.

[W3C01] W3C. Web Services Description Language (WSDL) 1.1, 2001. Available from: http://www.w3.org/TR/wsdl.

[W3C04a] W3C. Document Object Model (DOM) Level 3 Core Specification, 2004. Available from: http://www.w3.org/TR/DOM-Level-3-Core/.

[W3C04b] W3C. XML Information Set (Second Edition), 2004. Available from: http://www.w3.org/TR/xml-infoset.

[W3C04c] W3C. XML Schema Part 1: Structures Second Ed., 2004. Available from: http://www.w3.org/TR/xmlschema-1.

[W3C04d] W3C. XML Schema Part 2: Datatypes Second Ed., 2004. Available from: http://www.w3.org/TR/xmlschema-2.

[W3C07] W3C. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition), 2007. Available from: http://www.w3.org/TR/soap12-part1.

[W3C08] W3C. Extensible Markup Language (XML) 1.0 (Fifth Edition), 2008. Available from: http://www.w3.org/TR/xml/.

[W3C09] W3C. Namespaces in XML 1.0 (Third Edition), 2009. Available from: http://www.w3.org/TR/xml-names.

[W3C11] W3C. Efficient XML Interchange (EXI) Format 1.0, 2011. Available from: http://www.w3.org/TR/exi.

[WG08] Erik Wilde and Robert J. Glushko. Xml fever. *Commun. ACM*, 51:40–46, July 2008.

[Wil03] Erik Wilde. Xml technologies dissected. *IEEE Internet Computing*, 7:74–78, September 2003.

[WKNS05] Jules White, Boris Kolpackov, Balachandran Natarajan, and Douglas C. Schmidt. Reducing application code complexity with vocabulary-specific XML language bindings.

In *Proceedings of the 43rd annual Southeast regional conference - Volume 2*, ACM-SE 43, pages 281–287, New York, NY, USA, 2005. ACM.

[WTS07] M. Walker, R. Turnbull, and N. Sim. Future mobile devices: an overview of emerging device trends, and the impact on future converged services. *BT Technology Journal*, 25:120–125, April 2007.

[YLRY07] Yijun Yu, Jianguo Lu, Juan F. Ramil, and Phil Yuan. Comparing web services with other software components. In *2007 IEEE International Conference on Web Services (ICWS 2007)*, 2007. Appears in IEEE International Conference on Web Services 2007 (ISBN 0-7695-2924-0).

[ZDL09] Sonja Zaplata, Viktor Dreiling, and Winfried Lamersdorf. Realizing Mobile Web Services for Dynamic Applications. *AIS Transactions on Enterprise Systems*, 2009(2):3–12, 11 2009.

[ZMCO04] Olaf Zimmermann, Sven Milinski, Michael Craes, and Frank Oellermann. Second generation web services-oriented architecture in production in the finance industry. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '04, pages 283–289, New York, NY, USA, 2004. ACM.

[ZXjC^{+}10] Deng-Hui Zhang, Bin Xie, Hua jun Chen, Yang Lv, and Le Yu. Using Geodata and Geoprocessing Web Services in Embedded Device. In *Education Technology and Computer (ICETC), 2010 2nd International Conference on*, volume 1, pages V1–272 –V1–275, 2010.

Part VI

Appendices

Appendix A

List of SOS server instances

The following table lists the URLs of the servers used in the study presented in Chapter 8. The data used in this study was retrieved from these servers between July 1 and September 15, 2010.

Server URLs
http://152.20.240.19/cgi-bin/oos/oostethys_sos.cgi
http://204.115.180.244/server.php
http://81.29.75.200:8080/oscar/sos
http://ak.aoos.org/ows/sos.php
http://bdesgraph.brgm.fr/swe-kit-service-ades-1.0.0/REST/sos
http://ccip.lat-lon.de/ccip-sos/services
http://compsdev1.marine.usf.edu/cgi-bin/sos/v1.0/oostethys_sos.cgi
http://coolcomms.mote.org/cgi-bin/sos/oostethys_sos.cgi
http://data.stccmop.org/ws/util/sos.py
http://devgeo.cciw.ca/cgi-bin/mapserv/sostest
http://elcano.dlsi.uji.es:8080/SOS_MCLIMATIC/sos
http://esonet.epsevg.upc.es:8080/oostethys/sos
http://gcoos.disl.org/cgi-bin/oostethys_sos.cgi
http://gcoos.rsmas.miami.edu/dp/sos_server.php
http://gcoos.rsmas.miami.edu/sos_server.php
http://gis.inescporto.pt/oostethys/sos
http://giv-sos.uni-muenster.de:8080/52nSOSv3/sos
http://habu.apl.washington.edu/cgi-bin/xan_oostethys_sos.cgi
http://lighthouse.tamucc.edu/sos/oostethys_sos.cgi
http://mmisw.org/oostethys/sos

http://nautilus.baruch.sc.edu/cgi-bin/sos/oostethys_sos.cgi
http://neptune.baruch.sc.edu/cgi-bin/oostethys_sos.cgi
http://oos.soest.hawaii.edu/oostethys/sos
http://opendap.co-ops.nos.noaa.gov/ioos-dif-sos/SOS
http://rtmm2.nsstc.nasa.gov/SOS/footprint
http://rtmm2.nsstc.nasa.gov/SOS/nadir
http://sccoos-obs0.ucsd.edu/sos/server.php
http://sdf.ndbc.noaa.gov/sos/server.php
http://sensor.compusult.net:8080/SOSWEB/GetCapabilitiesGFM
http://sensorweb.cse.unt.edu:8080/teo/sos
http://sensorweb.dlz-it-bvbs.bund.de/PegelOnlineSOS/sos
http://sos-ws.tamu.edu/tethys/tabs
http://swe.brgm.fr/constellation-envision/WS/sos-discovery
http://vast.uah.edu/ows-dev/dopplerSos
http://vast.uah.edu/ows-dev/tle
http://vast.uah.edu/vast/nadir
http://vast.uah.edu:8080/ows-dev/footprint
http://vastserver.nsstc.uah.edu/vast/adcp
http://vastserver.nsstc.uah.edu/vast/airdas
http://vastserver.nsstc.uah.edu/vast/weather
http://v-swe.uni-muenster.de:8080/WeatherSOS/sos
http://weather.lumcon.edu/sos/server.asp
http://webgis2.como.polimi.it:8080/52nSOSv3/sos
http://wron.net.au/BOM_SOS/sos
http://wron.net.au/CSIRO_SOS/sos
http://ws.sensordatabus.org/Ows/Swe.svc/
http://www.cengoos.org/cgi-bin/oostethys_sos.cgi
http://www.csiro.au/sensorweb/BOM_SOS/sos
http://www.csiro.au/sensorweb/CSIRO_SOS/sos
http://www.csiro.au/sensorweb/DPIW_SOS/sos
http://www.gomoos.org/cgi-bin/sos/V1.0/oostethys_sos.cgi
http://www.mmisw.org:9600/oostethys/sos
http://www.pegelonline.wsv.de/webservices/gis/sos
http://www.wavcis.lsu.edu/SOS/server.asp
http://www.weatherflow.com/sos/sos.pl
http://www3.gomoos.org:8080/oostethys/sos

Appendix B

Performance Results

B.1 Sensor Descriptions Dataset

The SD dataset is characterised for containing very small files if compared with the rest of the datasets. The mean size of the 20 files is only 9 KB with a standard deviation of 4,7 KB. The execution time of all of the generated parsers in the three scenarios are presented in the following sections.

The measurements for this dataset show a slight advantage of XBinder over DBMG for the mobile configuration. Nevertheless, they both have fairly good results for small files. When executed in a more capable hardware the measures for all of the parsers show a high relative variability, which could be caused by the fact that most measured values are below 1 millisecond, and at this time resolution any minimal external disturbance may affect the measurement process. In any case, as the time taken to process these files is so small for PC scenarios we consider these differences negligible.

B.1.1 Mobile Configuration

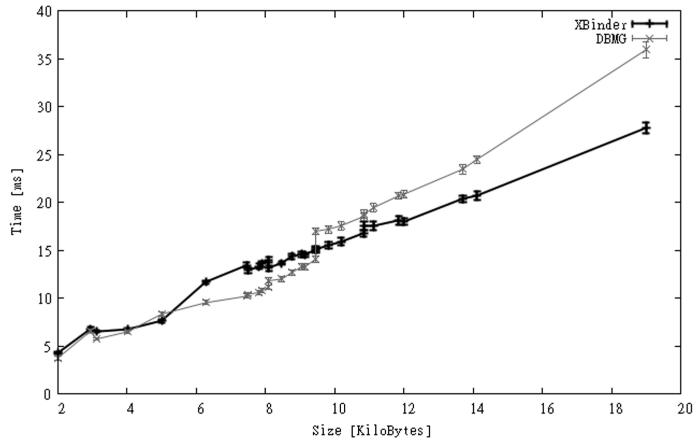

Figure B.1: Execution times for SD dataset (Mobile Scenario)

B.1.2 Mac OS X Laptop

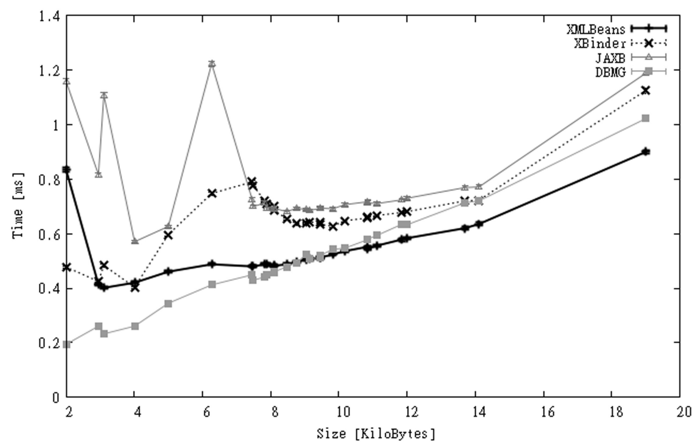

Figure B.2: Execution times for SD dataset (PC Scenario 1 - Mac OS X Laptop)

B.1.3 Windows PC

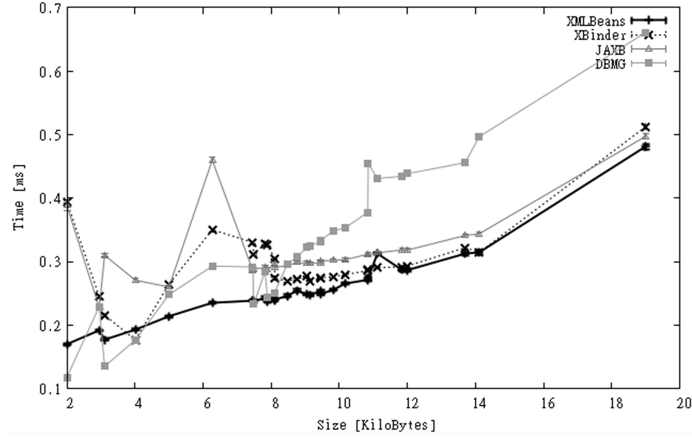

Figure B.3: Execution times for SD dataset (PC Scenario 2 - Windows PC)

B.2 Observations Dataset

The Observations dataset contains files with very different sizes. For this reason, we have separated the results exposition in two parts: files below 100KB (OBS-S) and files with size equal to or above 100 KB (OBS-L). The mean size is 1.32 MB with a standard deviation of 1.29 MB. The value for the standard deviation is rather large because most files were very small (< 10KB) or very big (> 2MB).

Unfortunately, the code generated by XBinder was not capable of processing correctly observations data. For this reason, we only included tests for the personal computer configurations. In these scenarios, for small files (OBS-S), XMLBeans had a significant advantage over JAXB and DBMG, which showed similar results. Although in general terms the processing times were very small for all cases. For large files (OBS-L), the figures for XMLBeans and JAXB were almost identical and both better than DBMG.

B.2.1 Mac OS X Laptop

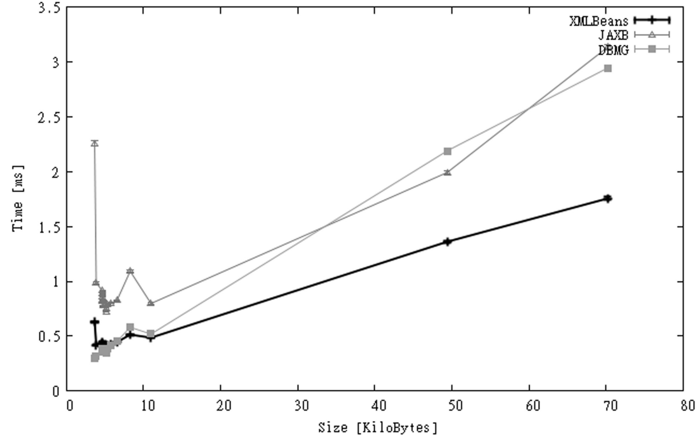

Figure B.4: Execution times for OBS-S dataset (PC Scenario 1 - Mac OS X Laptop)

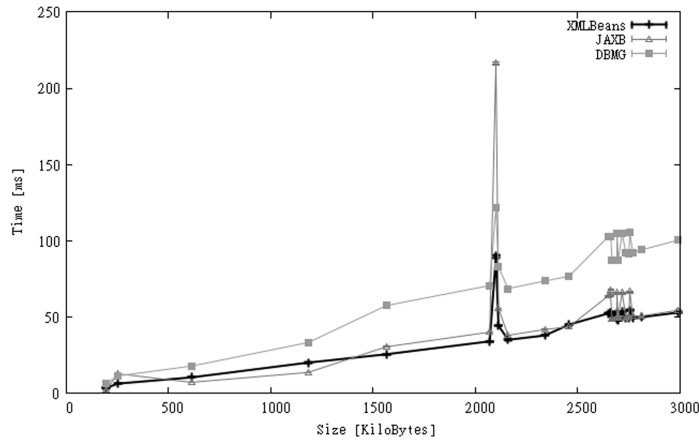

Figure B.5: Execution times for OBS-L dataset (PC Scenario 1 - Mac OS X Laptop)

The peak values in figure B.5 and B.7 are for a file that use a different form to encode observations. Listing B.1 shows the typical way of encoding observations, where a group of observations is included in the same *om:Observation* element. Using this method the metadata for the observations is specificied only once for the whole set of observations. The observations file showing the abnormal behaviour includes only a single observation value in every om:Observation element, which results in a file much bigger than an equivalent file encoded with the first method and with the same number of observations.

Listing B.1: Typical observations encoding

```
<om:Observation>
  <om:samplingTime>...</om:samplingTime>
  <om:procedure xlink:href=.../>
  <om:observedProperty>...</om:observedProperty>
  <om:featureOfInterest>...</om:featureOfInterest>
  <om:result>
    <swe:DataArray>
      <swe:elementCount> ... </swe:elementCount>
      <swe:elementType name=...>...</swe:elementType>
      <swe:encoding>
        <swe:TextBlock decimalSeparator="." tokenSeparator=","
          blockSeparator=" "/>
      </swe:encoding>
      <swe:values>
        2004-05-15T08:15:00+10,feature_name,0.43
        2004-05-20T01:45:00+10,feature_name,0.473
        2004-05-20T02:15:00+10,feature_name,0.485
        2004-05-20T02:30:00+10,feature_name,0.491
        2004-05-20T04:00:00+10,feature_name,0.522
        ...
      </swe:values>
    </swe:DataArray>
     </om:result>
 </om:Observation>
```

B.2.2 Windows PC

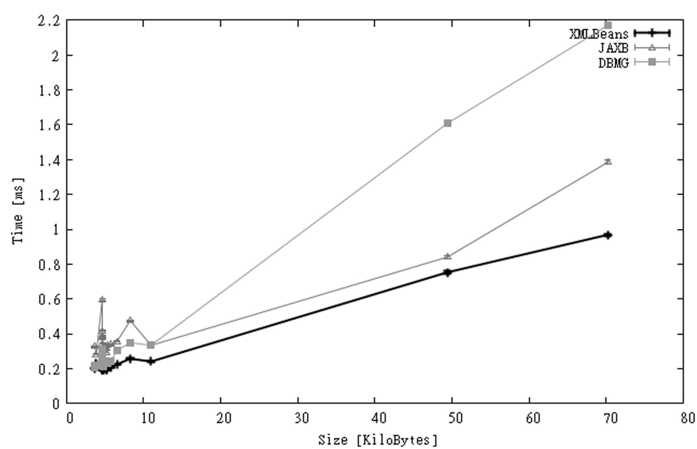

Figure B.6: Execution times for OBS-S dataset (PC Scenario 2 - Windows PC)

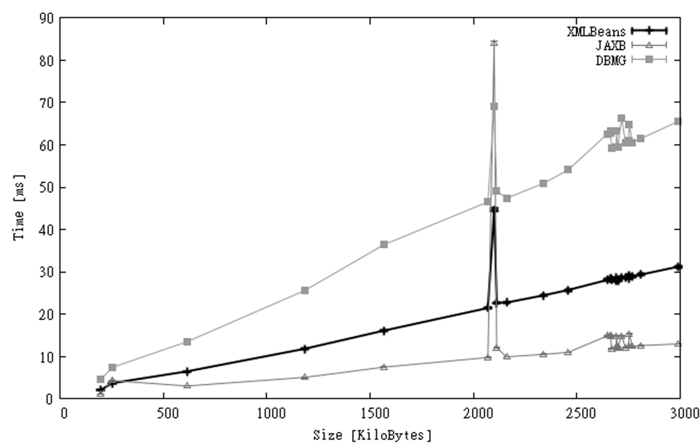

Figure B.7: Execution times for OBS-L dataset (PC Scenario 2 - Windows PC)

B.3 Measurements Dataset

Last, the Measurements Dataset contains mostly large files with a mean size of 3.14 MB. The standard deviation is 1.52 MB. For this dataset the experimetns showed similar results for XBinder and DBMG for the first two scenarios, but XBinder was better in the third one. For the other generators, JAXB tended to be better as the size of the files increased.

B.3.1 Mobile Configuration

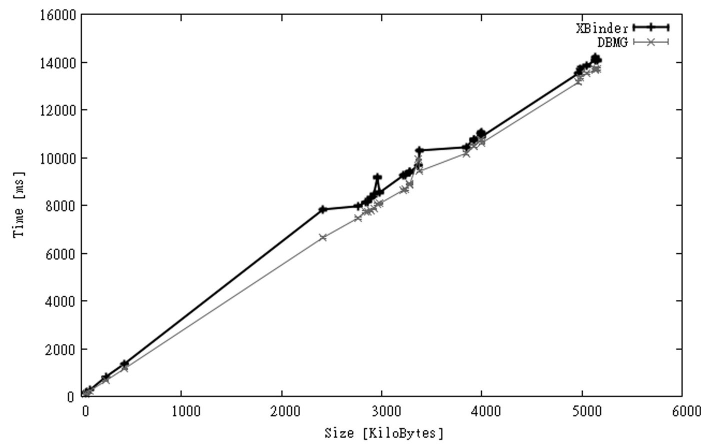

Figure B.8: Execution times for MEA dataset (Mobile Scenario)

B.3.2 Mac OS X Laptop

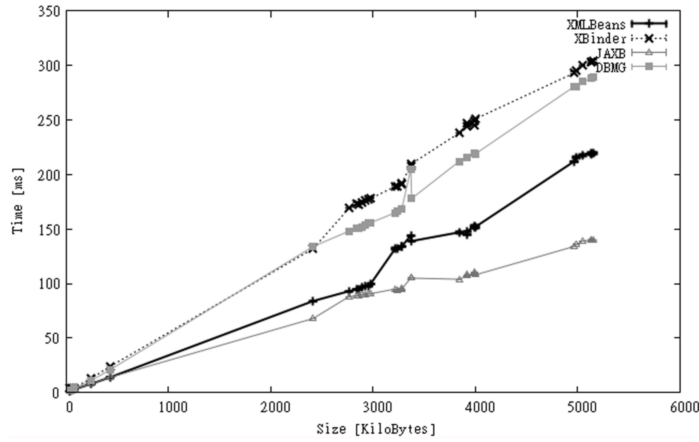

Figure B.9: Execution times for MEA dataset (PC Scenario 1 - Mac OS X)

B.3.3 Windows PC

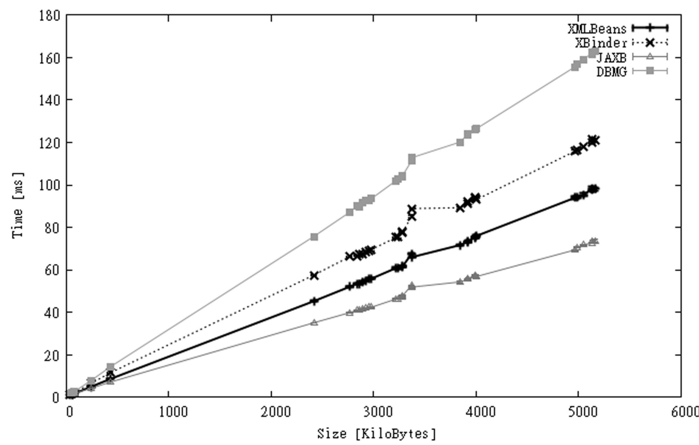

Figure B.10: Execution times for MEA dataset (PC Scenario 1 - Mac OS X)

Appendix C

Parsers Comparison

In this appendix we present a last experiment to measure the influence of the underlying parsing implementation used by the code generated with the different XML data binding tools presented in Chapter 9. In order to measure parsers performance, we created test cases for each one of them where the files where parsed but empty event handlers were executed each time a start or end tag was found. We only preformed these experiments for the CAPS dataset as we were only interested in having an approximate idea of the parsers behaviour.

XMLBeans use *Piccolo*[1] as XML parser. Piccolo is non-validating SAX parser reputed as being very fast. XBinder use the StAX parser distributed with the Java SE libraries based on Xerces[2] for the PC configurations. For code targeted to Android it uses an implementation of the XML Pull API provided by this platform. The code generated with DBMG use kXML as underlying parser. Last, JAXB use its own internal parser implementations, as a consequence we were not able to include measurements for it in this comparison.

In the mobile scenario the XML Pull parser provided by Android showed a slightly, but not significant, better performance that KXml. On the other hand, Piccolo and StaX outperformed KXml in the PC scenarios. In these scenarios StaX was also significantly faster than Piccolo. These results suggest that if the underlying parser for DBMG were changed for the PC scenarios the overall tool performance can be improved.

---

[1] `http://piccolo.sourceforge.net`

[2] `http://xerces.apache.org/xerces-j/`

C.1 Mobile Configuration

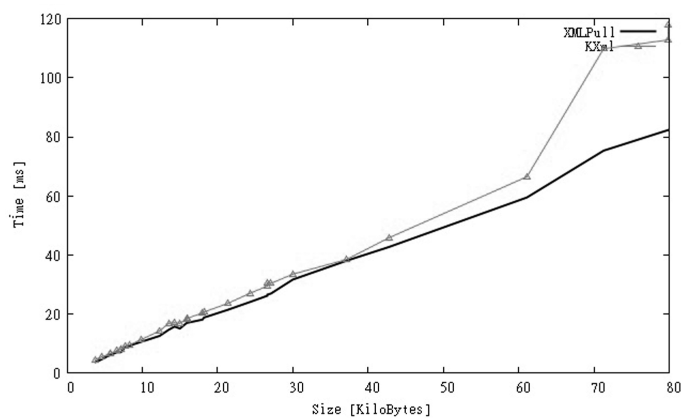

Figure C.1: Execution times for CAPS-S dataset (Mobile Scenario)

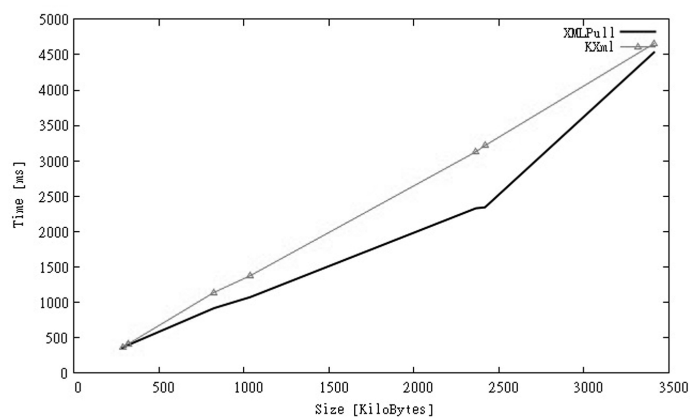

Figure C.2: Execution times for CAPS-L dataset (Mobile Scenario)

C.2 Mac OS X Laptop

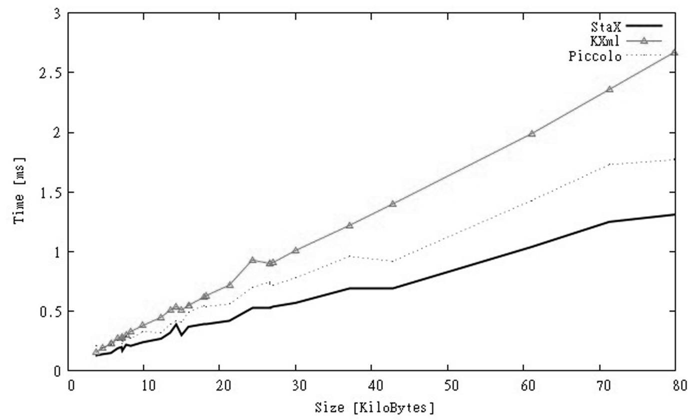

Figure C.3: Execution times for CAPS-S dataset (PC Scenario 1 - Mac OS X Laptop)

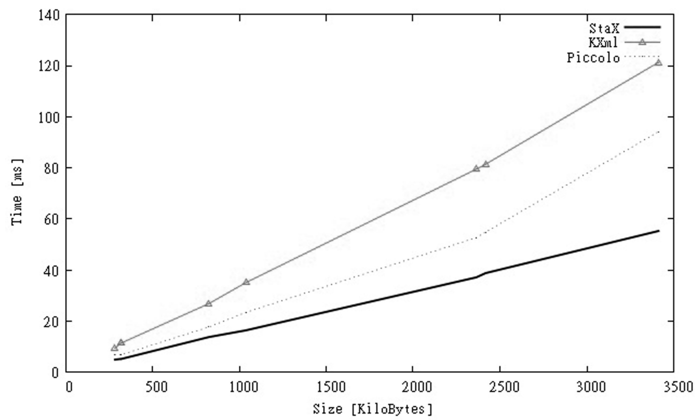

Figure C.4: Execution times for CAPS-L dataset (PC Scenario 1 - Mac OS X Laptop)

C.3 Windows PC

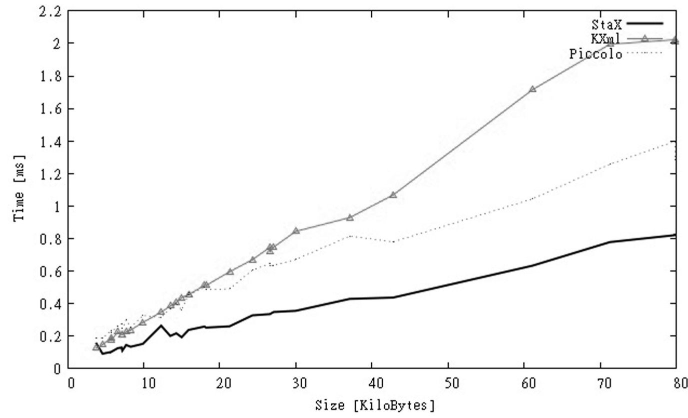

Figure C.5: Execution times for CAPS-S dataset (PC Scenario 2 - Windows PC)

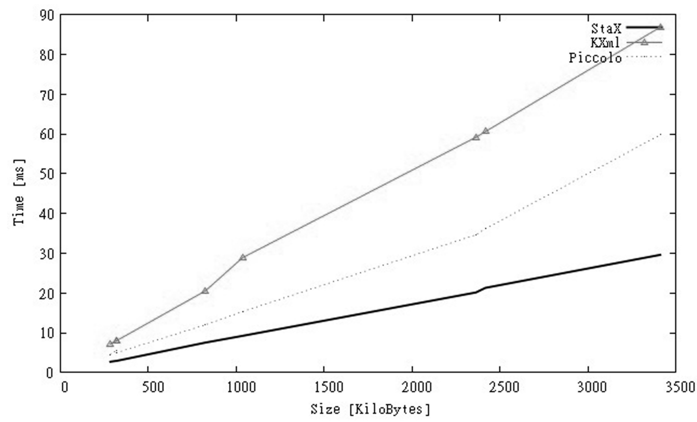

Figure C.6: Execution times for CAPS-L dataset (PC Scenario 2 - Windows PC)