On the performance of markup language compression

UNIVERSITY OF WEST LONDON

DOCTORAL THESIS

# On The Performance of Markup Language Compression

Antonio D. Kheirkhahzadeh

*Supervisors:*

Dr. John P. T. MOORE
Prof. Peter KOMISARCZUK

*A thesis submitted in partial fulfilment of the requirements*
*for the degree of Doctor of Philosophy*
*in the*
Sustainable Computing Research Group
School of Computing and Technology

*Scrutiny Panel:*

Dr. Nasser Matoorian, Dr. Ali Bahadori-Jahromi, Dr. Stephen Roberts

April 2015

# Declaration of Authorship

I, Antonio D. Kheirkhahzadeh, declare that this thesis titled, 'On The Performance of Markup Language Compression' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

*“If you have to do something, do it well.”*

My father

UNIVERSITY OF WEST LONDON

# *Abstract*

Sustainable Computing Research Group
School of Computing and Technology

Doctor of Philosophy

**On The Performance of Markup Language Compression**

by Antonio D. Kheirkhahzadeh

Data compression is used in our everyday life to improve computer interaction or simply for storage purposes. Lossless data compression refers to those techniques that are able to compress a file in such ways that the decompressed format is the replica of the original. These techniques, which differ from the lossy data compression, are necessary and heavily used in order to reduce resource usage and improve storage and transmission speeds. Prior research led to huge improvements in compression performance and efficiency for general-purpose tools which are mainly based on statistical and dictionary encoding techniques.

Extensible Markup Language (XML) is based on redundant data which is parsed as normal text by general-purpose compressors. Several tools for compressing XML data have been developed, resulting in improvements for compression size and speed using different compression techniques. These tools are mostly based on algorithms that rely on variable length encoding. XML Schema is a language used to define the structure and data types of an XML document. As a result of this, it provides XML compression tools additional information that can be used to improve compression efficiency. In addition, XML Schema is also used for validating XML data. For document compression there is a need to generate the schema dynamically for each XML file. This solution can be applied to improve the efficiency of XML compressors.

This research investigates a dynamic approach to compress XML data using a hybrid compression tool. This model allows the compression of XML data using variable and fixed length encoding techniques when their best use cases

are triggered. The aim of this research is to investigate the use of fixed length encoding techniques to support general-purpose XML compressors. The results demonstrate the possibility of improving on compression size when a fixed length encoder is used to compressed most XML data types.

# *Acknowledgements*

Firstly, I would like to thank my supervisor Dr. John Moore for making this research possible. His guidance and support, as well as his encouragement and enthusiasm, laid the foundations to this work. I would like to offer my gratitude for the academic and personal advice he provided me throughout the course of my PhD. He has been a role model who inspire me to start my career in academia and develop it further. Thank you John.

I would like to thank the School of Computing and Technology of the University of West London. Each member of this department encouraged and inspired me to undertake my PhD studies. My sincere thanks also goes to Dr. Thomas Roth-Berghofer for his trust on my potential and the encouragement he provided me to complete my studies. A special thanks goes to my colleague and friend Jiva Bagale with whom I shared the PhD life style and experience. I would also like to thank my PhD colleagues and friends Dean Kramer, Malte Ressin and Christian Sauer for the stimulating discussions and all the fun we had in the last years.

I would like to acknowledge the academic, technical and financial support of the University of West London who provided me the means to complete my studies.

Most importantly, I would like to thank my family for their encouragement and support during these years. My father for his words of wisdom and my mother for her patience and support. I would like to thank my brother and my two sisters for their unconditional love and caring. Thank you all.

# Contents

# List of Figures

# List of Tables

# List of Listings

# Abbreviations

| | |
|---|---|
| **API** | **A**pplication **P**rogramming **I**nterface |
| **ASN.1** | **A**bstract **S**yntax **N**otation One |
| **BER** | **B**asic **E**ncoding **R**ules |
| **BNF** | **B**ackus-**N**aur **F**orm |
| **CER** | **C**anonical **E**ncoding **R**ules |
| **CSS** | **C**ascading **S**tyle **S**heet |
| **DER** | **D**istinguished **E**ncoding **R**ules |
| **DOM** | **D**ocument **O**bject **M**odel |
| **DSL** | **D**omain-**S**pecific **L**anguage |
| **DTD** | **D**ocument **T**ype **D**efinition |
| **HTML** | **H**yper**T**ext **M**arkup **L**anguage |
| **IDL** | **I**nterface **D**escription **L**anguage |
| **IP** | **I**nternet **P**rotocol |
| **ISO** | **I**nternational **O**rganization for **S**tandard |
| **MAC** | **M**edia **A**ccess **C**ontrol |
| **PDU** | **P**rotocol **D**ata **U**nit |
| **PER** | **P**acked **E**ncoding **R**ules |
| **SAX** | **S**imple **A**PI for **X**ML |
| **SGML** | **S**tandard **G**eneralized **M**arkup **L**anguage |
| **SNMP** | **S**imple **N**etwork **M**anagement **P**rotocol |
| **UTF-8** | **U**CS **T**ransformation **F**ormat **8**-bit |
| **URL** | **U**niversal **R**esource **L**ocator |
| **W3C** | **W**orld **W**ide **W**eb **C**onsortium |
| **WAP** | **W**ireless **A**pplication **P**rotocol |

**XHTML** E**X**tensible **H**yper**T**ext **M**arkup **L**anguage

**XML** E**X**tensible **M**arkup **L**anguage

**XER** **X**ML **E**ncoding **R**ules

**XSD** **X**ML **S**chema **D**efinition

**XSL** E**X**tensible **S**tylesheet **L**anguage

**XSLT** E**X**tensible **S**tylesheet **L**anguage **T**ransformation

*To my parents, for their love and support.*

# Chapter 1

# Introduction

Data compression is a major field and research topic in computer science and information theory. The use of this technology has heavily influenced our everyday activities and interactions with computers. Although its implementation is transparent to the user, most of the technology available today depends on these compression techniques to improve software quality and overcome the hardware limits. As a major branch of computer science, data compression is applied to various fields that utilise some degree of compression.

A highly efficient compressor can significantly reduce the size of the original data depending on the technique used and the nature of the source. A trade-off usually exists between compression size and compression speed as higher levels of compression require more processing time. However, these resources may not be available on a constrained device. Therefore, a correct balance between these two variables is needed when operating in resource-limited environments.

Depending on the scenario, data compression can be either lossy or lossless. Lossy compression techniques are able to reduce the size of the file by removing bits of information without having a noticeable impact on the quality of the original file. For this reason, these techniques are commonly related and applied to multimedia, where the difference between the original and compressed format is less noticeable to the human perception. Conversely, lossless compression techniques are able to compress a file in such ways that the decompressed format is a replica of the original. Contrary to lossy compression, these techniques are heavily used in areas such as data storage, networking and

the compression of data formats. For each field, a specific set of compression algorithms have been developed and optimised for specific tasks.

## 1.1 Markup Languages

A markup language is a data format capable of annotating a document with syntax distinguishable from the main text. Commonly referred as the language of the web, this system allows to set further instructions on a document in order to specify its purpose. Examples of markup languages are Extensible Markup Language (XML), HyperText Markup Language (HTML) and TeX. HTML has been defined as the markup language for documents of the World Wide Web whereas, XML has been widely used to define data structure (Bray et al., 2008) due to its ability to extend itself and being able to define unique tags.

XML is a standard for data storage and data exchange over the Internet. It has the ability to represent structured data in a human and computer readable format and provides support for Unicode (Bray et al., 1998). However, XML has several disadvantages mostly related to its verbose syntax. Lengthy tags and redundant data can have a significant impact on system performance, especially within constrained networking environments. XML Schema is one of a number of validation methods which can be applied to XML. In addition to validation, an XML Schema, through the use of data type information, can provide sufficient knowledge to allow compression of XML data (Sperberg-McQueen and Thompson, 2000). As a result of its support for custom data types, XML Schema is seen as a replacement for the Document Type Definition (DTD). In addition, unlike DTD, XML Schema is written in XML which provides the ability to use standard XML parsing libraries to process the information.

## 1.2 XML Compression

Over the last decade, the demand of processing and storing of XML has increase exponentially. A number of optimisation techniques have been devised in order to overcome the limitations of this verbose language. Research has mainly focused on minimising the memory consumption required to process

XML and reduce the compressed size required to store or exchange data. XML compression techniques have been developed to take advantage of the verbose and redundant structure to improve compression efficiency. Several tools for compressing XML documents have been developed resulting in improvements in both size and speed using different compression techniques. These techniques have been used in various research leading to improvements in networking and storage compression. As a standard for interchanging data between heterogeneous applications, XML compression techniques have been widely adopted in networking. In addition, for its ability to represent data structure, the need of compressing XML has extended to storage and database.

General-purpose compressors are not able to achieve the highest level of efficiency due to the lack of local redundancy found in XML files (Augeri et al., 2007; Cheney, 2006b; Ferragina et al., 2006; Ng et al., 2006a; Sakr, 2008, 2009). These techniques are based on algorithms which exploit the predictable nature of XML data to remove unnecessary bits of information. With the introduction of more complex algorithms, general-purpose compressors have been offering a fast and reliable compression with higher encoding rates. Reasonable compression efficiency is achieved when compressing XML data using these techniques.

XML-conscious techniques have been developed to achieve higher compression rates by manipulating the XML structure. This manipulation generally involves similar techniques implemented in general-purpose compression applied to XML data to increase the local redundancy. Therefore, most of the XML-conscious techniques can be classified as front-end applications. This part of the system is aimed at restructuring XML data in a format which is subsequently passed to one or many back-end general-purpose compressors.

There are different compression techniques aimed at reducing the size of XML data. The term *compression* is usually referred to those techniques which implements a general-purpose compression algorithm after manipulating the XML data. These techniques are able to increase the local redundancy and exploit knowledge of the back-end compression algorithm to achieve better compressed formats. Most effective techniques include semantic awareness to categorise data into substructures, which are then compressed using most effective back-end compression algorithms.

The term *encoding* is referred to those XML-conscious techniques which implement their own encoding mechanisms. Encoding rules are used to reduce the redundancy of the structure and data of XML by mapping source symbols and blocks of data to an efficient binary representation. This technique allows the binary format to be decoded using the rules provided by the encoder.

### 1.2.1 Fixed Length Encoding

Compression algorithms try to find the most economical method of writing blocks of data to binary representations. Different techniques exist to achieve efficient representations that allow the least amount of information to be stored in the compressed format. The process of transforming data into a different, more compact format is defined encoding. **Fixed** and **variable** length encoding are the two main techniques to compress data in efficient binary formats. Variable length encoding techniques map source symbols into variable number of bits depending on their frequency. Whereas, fixed length encoding techniques result in a fixed number of bits per source symbols. Table 1.1 compares variable to fixed length binary representation for a simple alphabet.

| | **A** | **B** | **C** | **D** | **E** | **F** |
|---|---|---|---|---|---|---|
| **Frequency** | 32 | 23 | 18 | 13 | 11 | 6 |
| **Fixed Length** | 000 | 001 | 010 | 011 | 100 | 101 |
| **Variable Length** | 1 | 01 | 0010 | 0011 | 0001 | 000010 |

TABLE 1.1: Variable and Fixed length bit mapping

As shown in table 1.1, fixed length encoders require more bits to store the source information when the likelihood of symbols to appear varies. For this reason, the use of variable length encoding has been preferred over fixed length techniques. Based on this observation, extensive research has led to the development of advanced techniques to support variable length encoding to achieve more compact binary representations.

With the information provided in an XML Schema, it is possible to achieve an efficient encoding mechanism by mapping blocks of data to their lowest binary representation. Using this information, fixed length encoders are able to achieve

more efficient compressed formats compared to variable length encoding techniques. Data types can be considered the building blocks for fixed length encoders since they provide the rules to validate and encode data. However, this schema-*informed* compression also allows variable length encoders to achieve higher encoding rates. XML-conscious compressors exploit the XML schema information to compress data semantically and separate data from structure which allows variable length encoders to be more efficient.

### 1.2.2 Motivation

This research is motivated by the need to develop an XML compressor that can be used for a wider range of applications associated with document storage. In addition, the aim is to provide an efficient format which can be used to improve networking and storage. Achieving a higher compressed size is essential for the scalability of applications running on a low-bandwidth network or in need to reduce disk space usage.

Initial studies (Moore, 2010a) have demonstrated the possibility of achieving additional levels of compression using fixed length encoders. These techniques have demonstrated a better management and scalability for low-bandwidth network applications by transmitting smaller payloads (Moore et al., 2010). The advantage of managing a more efficient compressed format has also demonstrated to be beneficial for constrained devices that require to communicate across heterogeneous networks (Moore et al., 2012). However, the XML data sets found in this scenario are optimised for compression and representation purposes. XML allows different structures to be defined depending on the requirement of the application. Some of the data types are not optimised for compression and encoded using variable length encoding techniques. This work extends prior studies in this field by providing support for data types that are usually compressed using a variable length encoder. In conclusion, this research is motivated by the need of achieving better compressed formats for a wider range of XML data sets in order to improve network transmission or data storage applications.

## 1.3 Research Approach

Compression is an important process for all the applications that require to store, process or exchange XML data. The pervasiveness of this language has led to an extensive amount of research and software outputs. A number of tools have laid the basis for XML-conscious techniques by manipulating source data and presenting more compression-ready formats to back-end algorithms. More advanced techniques have been able to utilise the information provided in XML schema files to compress data more efficiently. These are defined as Schema-informed techniques and rely on an additional file to be used for both encoding and decoding routines. Although these techniques have found great success for exchanging XML data amongst homogeneous networks, the need of an external entity does not allow the same benefits for storage purposes. In this scenario the XML Schema has to be compressed with the XML file, decreasing the benefits of applying a Schema-informed compression.

The use of a schema language limits the compression to XML files and other markup languages. This information is mostly needed to validate and provide rules to encode data. However, some XML files are limited by the structure and nature of the document. The use of inconsistent structures and data types leads to poor compression efficiency. For example, HTML documents can be poorly structured and can contain large amount of text per element. In this case, a variable length encoder is able to provide a better compression compared to fixed length encoder. Therefore, analysing the XML file prior to compression is essential to decide what type of compression is best to apply.

Standards such as the Efficient XML Interchange (EXI) format have been developed by the W3C to overcome the limitations of compressing XML (Schneider and Kamiya, 2011). The use of a schema to improve compression has led to encoding techniques similar to those implemented in telecommunication and computer networking. Fixed length encoders are at the basis of these compression techniques derived from Abstract Syntax Notation One (ASN.1) (ITU-T, 2008b). Although current compressors are in favour of compression algorithms based on variable length encoding, schema-informed techniques based on fixed length encoders have demonstrated a compression efficiency beyond the level of any compressor.

### 1.3.1 Research Questions

The main research question is focused on the use of fixed length encoders to compress XML data. This technique can be subsequently applied to other markup languages which can benefit from a descriptive schema language. Data type constraints play a major role in enabling a fixed length encoding technique. This data provides the additional encoding information to compress and decompress the markup language. The research question is described in the following statement.

#### Main Question

**Can a fixed length encoder improve general-purpose compression techniques for markup languages?**
This question focuses on the major role of fixed length encoding techniques applied to markup language compression. XML is used as the main markup language and is one of the most popular and widely used language today. The scope of this research question is to evaluate the efficiency of these techniques applied to a wider range and different nature of XML documents.

A number of questions can be derived based on current knowledge of XML compressors. As these techniques are already applied on a small scale, it is important to extend the use of fixed length encoders to specific data types.

#### Questions

**How can a fixed length encoder be extended to support more domain-specific data types to aid compression?**
Currently most fixed length encoders support a limited range of data types. These data types are based on simple encoding mechanisms using the information provided in the XML Schema. The use of fixed length encoding is explored for supporting a wider range of data types. This depends on the ability to recognise XML data formats from the document content.

**Is it possible to increase compression by adding support for non-domain specific data types?**
Mapping XML data types to specific values is an important process required to apply a fixed length encoding technique. A set of built-in data types are usually defined within the compression tool. Although the aim is to increase the amount of data types compressed using a fixed length encoding technique, it is not possible to define all the possible variation of data types used in real XML data sets. For this reason, mapping data values to their closest data types is essential to achieve the best performance. This research question focuses on compressing data values that would not be otherwise assigned to a fixed length encoder.

### 1.3.2 Aims and Objectives

A number of compression techniques have been developed to address the issues related to the use of XML. These techniques are based on fixed and variable length encoders using schema informed or uninformed compression. However, very little has been done to investigate the use of structured XML data types to aid compression. More efficient formats can be achieved by analysing these data types and applying a fixed length encoding capable of mapping high-level data types to the least number of bits. The aim of this research is to identify the limits of a hybrid approach by extending schema-informed compression tools to general XML documents.
The objectives of this thesis are as follows:

- Develop a model to compress XML data using a hybrid system which implements both fixed and variable length encoding when their best use cases are triggered.
- Devise a system to encode high-level data types using fixed length encoding techniques.
- Identify the best use cases for compressing XML data using the hybrid model.

- Explore the use of fixed length compression techniques to improve the efficiency of XML compression.

### 1.3.3 Approach

Current XML compressors lack semantic knowledge. This information is only provided by a descriptive schema language which defines the structure and data types of XML. Schema-uninformed compression is therefore not able to achieve highly compact binary representation. However, advanced techniques such as EXI are able to achieve high level of compression using both fixed and variable length encoding techniques. This level of compression is achieved using low-level built-in data types and string tables for compact representation of repeated string values. High level data types are therefore compressed using variable length encoding techniques.

A more efficient compression can be achieved using a fixed length encoder to compress most of the data types found in XML. This approach involves the development of a hybrid system capable of separating data from structure using a transparent schema-informed technique. XML data is categorised into set of data types. Basic types are represented as non-empty values which can be encoded and transmitted using a specific encoding rule. Character Strings are defined as a subset of basic types which present a less efficient encoding mechanism (Dubuisson, 2001; Larmouth, 2000). This data type is usually compressed using a variable length encoder while other basic types are processed by a fixed length encoder. Basic data types consist of low-level to high-level formats such as Integer, Enumeration, Bit String, and Date. Character Strings data types instead, consist of formats which cannot be efficiently mapped to encoding rules such as IA5String and UTF8String (ITU-T, 2008a). Compressing high-level basic data types using a fixed length encoder, together with the transparent schema-informed compression, allows the hybrid model to achieve a level of compression beyond current XML compressors. The idea of this thesis is to apply encoding rules extended from ASN.1 to outperform XML-conscious and general-purpose compressors.

## 1.4 Contributions

This research investigates the performance of compression techniques applied to XML data. A hybrid system is developed based on the requirements considered during the study of XML compression techniques. Furthermore, this research can be extended to the compression of other markup languages that can be described by a data definition language. The major contributions of this thesis are as follows:

1. A survey on current compression techniques used to compress XML data. These tools are thoroughly analysed to demonstrate the potential of their compression model compared to others. This survey contributes to existing studies presenting state of the art tools and techniques for XML compression.

2. A study of EXI and Packedobjects compression models and tools, which have only been previously described in their format specification. For each of these tools, the compression model, applicability and limitation are described. In addition, simple examples are provided to illustrate how an optimal compression is achieved.

3. A comparison the compression efficiency and performance of a number of tools analysed in previous survey in the field of network management. The application of XML compression techniques in this domain was selected based on the performance of these tools for small highly-structured XML files. These files have a stronger emphasis on the structure of the document with XML tags used to separate and highlight each data types.

4. A system for compressing high-level data types using a fixed length encoder. Existing solutions focus on using fixed length encoding techniques only when a strict schema-informed technique is applied. This work presents a transparent schema-informed compression which allows data types to be defined in a domain-specific data definition language. This language is used as a protocol to allow low-level and high-level data types to be compressed using a fixed length encoder.

5. A new model for compressing XML data using encoding rules derived from Packedobjects. The model is based on a number of processes followed

by the fixed and variable length encoding mechanisms. The experimental results demonstrate a good performance relative to the compression size achieved by the new model for XML files with a specific size range.

6. A hybrid model, referred to as Hybrid Packedobjects (HPO), that implements a fixed and variable length encoding techniques when their best use cases are triggered. An experimental evaluation of HPO against the most efficient XML and general-purpose compressors is provided. These tools are compared against a corpus composed of several data sets which differ by size, data types and XML structure. HPO demonstrates the ability to achieve a better compression by encoding most data types using the fixed length encoder.

7. The results of the experiments demonstrate the additional amount of compression that can be achieved by encoding high-level data types using fixed length encoders. Furthermore, this research provides an analysis of the XML corpus used to perform the experiments, demonstrating the amount of high-level data types which exists in real XML data sets. These data types are classified using regular expressions providing encoding mechanisms to define these data types in a data definition language.

These results demonstrate a significant improvement in compression for XML documents based on structured data.

## 1.5 Published Material

This research has led to a number of joint publications. Each publication is based on parts of the thesis and presents some of the contributions listed in the previous section. A complete list of publications with the related contribution is provided in Appendix E.

## 1.6 Organisation of the Thesis

The remaining chapters of the thesis are structured as follows. **Chapter 2** provides the technical background necessary for a detailed understanding of

markup languages and data compression. The first part of the chapter discusses and provides examples of markup languages focusing on XML, describing its structure and main components. This part provides a background on the application programming interfaces (API) used to parse, query and transform XML documents. XML validation is discussed based on schema languages and the validity of XML structure based on XML standards. This background information will be used in subsequent chapters for the development of the hybrid model. The second part of Chapter 2 provides the basis of data compression. The difference between fixed length and variable length encoders is discussed thoroughly with examples using different encoding techniques. The chapter continues to describe various techniques to compress XML data and classifying XML-conscious techniques. The chapter concludes with features and classification of XML-conscious techniques.

**Chapter 3** provides a description of the most relevant and effective tools to compress XML data. A more detailed analysis is given for tools that play a major role in the development of the hybrid model. The first part describes these tools based on variable and fixed length encoding, explaining back-end tools and technologies such as *zlib* and ASN.1. The second part of Chapter 3 provides an analysis of XML data sets from the literature review and online publicly available repositories.

**Chapter 4** presents the preliminary experiments performed using the tools described in Chapter 3. This part focuses on XML compression techniques in order to improve network management using a set of small highly-structured XML files collected from network devices such as routers, switches and sensors. Results illustrate the performance of XML compression techniques for this specific data set highlighting the performance of schema-informed techniques over standard XML-conscious approaches. The aim of this chapter is to evaluate the performance of XML compressors for a specific data set which have not been evaluated in other work.

**Chapter 5** describes the architecture of the hybrid model. The first sections provide the motivations and requirements for the development of the hybrid model. These considerations are based on the studies of XML compression tools of Chapter 3 and the results on the performance of schema-informed techniques of Chapter 4. The chapter continues on describing the various processes of the hybrid model during encoding and decoding of XML data. A motivating example

to compare the results of the best XML compressors against the hybrid model is provided to demonstrate the additional compression that can be achieved by the hybrid model. The chapter concludes describing the applicability and limitation of this approach.

**Chapter 6** is divided into several sections. The first section describes the methodology and the environment where the experiments were performed, to allow replicable experiments. This section describes the compression tools to which the hybrid model is compared and the data sets used. The second section provides the results of the experiments divided into synthetic and real XML data sets. A third section provides an analysis to justify the difference in efficiency between the different data sets and discusses the performance evaluation.

Finally **Chapter 7** discusses the main findings and addresses the research questions raised. The hybrid model is classified within the set of compression tools based on the results achieved. This chapter also discusses the limitation of this approach and provides directions for future development and research. A number of appendices support the experiments and the hybrid model processes description.

# Chapter 2

# Technical Background

This chapter provides the basic knowledge of Markup Languages with particular attention to Extensible Markup Language (XML). The structure of XML is discussed together with the basic components required to construct XML documents. In addition, simple examples are provided in order to clarify the difference and the use of these components in particular scenarios. The use of application programming interfaces to obtain information from the XML data are discussed together with the different methods to process information using tree and event based approaches. This chapter introduces standard languages to validate XML data, highlighting the difference between several validation languages and the validity of XML data based on the well-formedness of the structure. Metrics and classification are provided to categorise XML documents based on their content and structure. The second part of this chapter introduces document compression as the domain area where XML is studied. This part focuses on lossless compression applied to XML data and discusses different techniques used in this area. This section provides the background knowledge on compression tools, defined as XML-conscious which are specifically designed to compress XML data. The main features and classifications of the most influential tools which will be evaluated in later chapters are described.

## 2.1 Markup Languages

Markup language is a specific way of annotating a document with syntax distinguishable from the main text. This language contains a syntactically different set of text usually annotated with special characters which are needed to provide specific instructions regarding the nature and scope of the document. Annotations have the dual role of being used to inform users and instruct computers. For this reason, these annotations are represented in a computer and human-readable format.

Markup languages can vary depending on the purpose and area of use. There are three main categories that can be distinguished when defining these languages (Coombs et al., 1987). *Presentational* is a type of language in which the markup is used to inform authors and readers about the structure of the text. As a result, it is possible to set a structure and then define characters that are often hidden from the author to express paragraphs, chapters and other typographic divisions in order to make the text visually readable. *Procedural* can be defined as an explicit version of presentational markup. In a document formatting scenario, the markup will provide a set of instructions defining the typographic divisions that will be visible once the document will be processed by a computer. *Descriptive* markup labels parts of text informing both readers and machines on the use of the concerned text. Unlike procedural, the descriptive markup focuses on encapsulating parts of text to specify its purpose or, more technically, to relate sets of text to a particular class. This can be easily recognised by the use of presentation semantics to represent the markup language in a human-readable and understandable manner. The use of tags to contain text is usually the form in which descriptive markup languages are represented. Examples of Descriptive Markup Languages are HyperText Markup Language (HTML) and Extensible Markup Language (XML). These languages are therefore commonly referred to as “languages of the Web”.

Table 2.1 provides an example of the three languages used to construct a paragraph using LaTeXand HTML as Procedural and Descriptive languages respectively. A common feature of markup languages is the use of a specific set of annotations to define how the data is to be presented. These predefined semantics are used to provide a purpose to the text defined within the tags. In the Descriptive example of table 2.1, <p> and <br> are used in order to describe

TABLE 2.1: Types of Markup Languages

| Presentational | Procedural | Descriptive |
|---|---|---|
| Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.<br><br>Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. | **hspace**{**1em**}Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. **vspace**{**3mm**} Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. | <**p**> Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. <**br**>Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.<**p**> |

a “paragraph” and “break”. However, other markup languages such as XML do not allow the use of predefined semantics. The lack of this feature allows markup languages, with particular attention to XML, to be extended into other areas including document representation, database storage, web services, and user interfaces.

### 2.1.1 XML

XML is a standard for data storage and data exchange over the internet. It has the ability to represent structured data in a human and computer readable format and provide support for Unicode (Bray et al., 1998). Most of the features of this markup language derive from Standard Generalized Markup Language (SGML), an ISO standard used as the basis on which XML was built (Jelliffe, 2006). XML was designed to carry data that needs to be stored or transported over networks. The use of predefined presentational semantics is one of the first features in which HTML can be initially defined differently from XML. However, these languages also differ by having distinctive goals, even though they can be applied within the same domain. XML focuses on the data and is designed to transport and store information while HTML, in conjunction with CSS, focuses on the presentation of the data, how it is displayed and how it looks.

LISTING 2.1: Example of XML

```
<?xml version="1.0" encoding="UTF-8"?>
<email>
  <from>Alyssa P. Hacker</from>
  <to>Eva Lu Ator</to>
  <subject>Subject of the email</subject>
  <body>Content of the email</body>
</email>
```

Code listing 2.1 provides an example of an XML document containing tags such as <mail>, <to> and <from>. These tags are not defined in the W3C standards, instead they are constructed by the author in order to structure the data contained in the email. Code listing 2.1 represents an email document structure with a self-descriptive design. One of the main feature of XML is the ability to allow users to design their own structure in order to wrap information within the markup language. The simplicity of XML is one of the key features that have led to an expansion of this language in different areas of computing. Data stored in an XML file is hardware and software-independent. This allows developers to easily exchange data over the Internet using XML as default format between heterogeneous applications. In addition, due to the extensibility and adaptivity of XML, this language has been successfully adopted in a number of other fields as syntax to develop document formats and as a Domain-Specific language (DSL). The ability to represent data structures has enabled the use of XML in fields such as web services, database storage and network protocols in order to develop applications which rely on a widely used and common language. Extensible HyperText Markup Language (XHTML) is an example of how XML has been successfully used to *extend* HTML and enable a well-formed HTML-like document to be parsed as a XML document.

#### 2.1.1.1 Structure of XML

Developed as a profile of SGML, XML can be defined as a subset of the original language. The primary intention for which XML was developed, was to create a lighter version of SGML for a more robust implementation to be used in the World Wide Web (Jelliffe, 2006). Therefore, most of the main elements of this language are still preserved in XML, except for some restrictions which are allowed in SGML (Bray et al., 1998). The structure and well-formedness of XML are defined by the W3C XML Working Group specifications (Bray et al., 2008).

XML components can be categorised into two different subsets. Some components enclose pure data that will be used to inform on the scope of the document, whereas others are specific to the application that will process the XML. Definitions and examples of main components are listed below.

LISTING 2.2: Simple XML BNF

```
1 document        ::=   prolog element misc*
2 element         ::=   empty_element | start_tag content end_tag
3 empty_element   ::=   '<' element_name (attribute)* '/>'
4 start_tag       ::=   '<' element_name (attribute)*'>'
5 end_tag         ::=   '</' element_name '>'
```

**Element**

Element is the technical name of a component formed by pairing of a start tag and an end tag. Each XML document must contain at least one element, *i.e.* the root, which can contain sub elements or a mix of XML-valid components. Code listing 2.2 is a simplified version of how XML is constructed using Backus–Naur Form (BNF) language, starting from the document to the defined element structure. Element within the root must be strictly nested with opening and closing tags. Element names are case sensitive and can contain a variety of characters with exception of spaces. Some restrictions are applied on the first characters of the element name i.e. it cannot contain the XML letters, numbers or punctuation characters. Empty elements can be used to define a document structure with null data for a less verbose and more robust document.

```
<student> ... </student>
```

**Attribute**

These components are used in order to provide additional information that is not usually related to the data contained within the XML. This information can also be expressed using a child nested element. There are no restrictions on using an attribute over a child element, however, because of the limitations of these components, complex data that require multiple values or a tree structure are best enclosed using element components. Therefore, attributes are used in situations where information is static and usually needed to provide additional information to the software in order to support the element parsing. Attributes are defined inside the starting element and mainly consists of an attribute name and value expressed using an equal sign and double quotes for data value.

```
<student dob=''...''> ... </student>
```

**Processing Instruction**

Processing Instruction (PI) is an XML component aimed at informing the application on the processing of XML. Data contained in PIs will provide information regarding the processing, transformation, or query of XML. As an option available for the `misc*` object defined in code listing 2.2 line 1, PI can appear in any location and nesting level of XML document. Similar to the `prolog`, aimed to inform the application about the version of XML and encoding scheme, PIs are constructed by a single element starting with "<?" and ending with "?>". A language target and content are then provided within the tags in order to inform the application about the language and the instructions respectively. This technique has been widely used in order to embed information within XHTML and provide additional support to the development of web pages.

```
<?php echo variable; ... ?>
```

**Namespaces**

Namespace is an advanced XML component defined in the W3C recommendation. Namespaces are used to solve the ambiguity between elements with identical names in order to differentiate multiple categories. For example, in an XML document containing an employee in multiple departments, the developer utilises the tag `<dept>` to describe a specific department within the system. However, ambiguities can arise when multiple `<dept>` tags can appear in the same nested level. Using element tags with a prefix, it is possible to avoid conflict with elements containing similar name but different data and meaning.

Namespaces must be defined in the parent node or inside the root node if they are intended to be used globally across the entire XML document. The namespace definition is constructed similarly to attributes, with the attribute name defined as `xmlns:prefix` and the value of the attribute to be URL for the namespace to be used as identifier. The prefix is then used as `prefix:dept` to define its category and provide more information regarding the data contained within the XML. Default namespace is a specific type of component where the namespace definition does not define a prefix. In this specific case all elements which do not have a namespace are linked to the URL defined in the default namespace definition. An example of namespaces usage is given in code listing 2.3.

LISTING 2.3: Namespaces usage

```
...
 <volumes xmlns=‘‘http://volumes/ns/1.0’’
          xmlns:eng=‘‘http://volume/eng/1.1’’>
        <volume id=‘‘ ... ’’>
                <title> ... </title>
        </volume>
        <eng:volume id=‘‘ ... ’’>
                <title> ... </title>
        </eng:volume>
</volumes>
...
```

**CDATA**

Software processing XML data are able to traverse and validate documents by tracking starting and ending tags. However, some situations require data containing illegal XML characters to be included into the document. Character Data (CDATA) is the term used to identify an XML component enclosing data to be ignored by the parser. For this reason CDATA is able to contain illegal XML characters such as “<” “>” or “&”. Opposite to Parsed Character Data (PCDATA) which is processed by the parser, CDATA differ from comments as being a fundamental part of the document to be ignored by the parser. For example, within the element ‘‘<student>...</student>’’ only the content of this element will be processed as text. However, by using CDATA component to enclose the student element, all the characters, including the markup, will be processed as text.

```
<![CDATA[<student> ... </student>]]>
```

#### 2.1.1.2 Application Programming Interface

Understanding how an XML document is parsed and the difference between the following parsing techniques is essential to the development of a tool that requires to process XML data. Knowledge of Application Programming Interfaces (APIs) can provide useful insight on the behaviour and performance of a tool. Some of the limitations of this research derive from the use of a specific API. These limitations are considered and discussed in future work.

As mentioned in the previous sections, XML is just a document containing markup language. In order to obtain knowledge of the information contained in the document, the XML file needs to be processed using software that is able to recognise the markup and provide an application interface to handle elements and other declarations. Event- and tree-based interfaces are two main categories to provide APIs for XML processing.

**Event-based**

The Event-based approach is a stream-oriented API that allows the user to access part of the XML tree sequentially. Simple API for XML (SAX) (Megginson et al., 2001) is the standard developed in different languages and used in various projects. This approach provides an API to operate on part of an XML document without constructing an in-memory representation. Therefore, this implementation has very little impact on the memory that is needed to process the XML. Using a SAX API, the user defines a set of callback methods which will be called when a specific event occurs during the XML parsing. Elements that construct XML files are therefore managed as events based on opening and closing tags. A disadvantage of SAX is the lack of a bidirectional parsing technique. Because there is no memory representation of the data, it is not possible to access parsed elements without processing the entire document again. However, creating a memory representation using an event-based parser or parse an in-memory tree are the two techniques that are available in most SAX API. This event-based approach is most suited when the user requires bits of information that can be always retrieved in the same manner. For this reason, SAX can be considered the best approach when parsing an XML document for strings contained in elements or attributes with very little impact on system resources.

**Tree-based**

The Document Object Model (DOM) is a type of tree-based API that allows users to generate a memory representation of an XML document (Nicol et al., 2001). Using this internal tree structure, users are able to navigate the tree and retrieve information contained within the elements and other components. An important aspect of a DOM parser is the presence of nodes which are the memory equivalent of elements. Users are able to create and modify nodes from a tree structure with a bidirectional

parsing technique. Since the XML needs to be loaded in memory, this implementation tends to have an impact on system resources starting from files in the kilobytes range. However, this API is mostly needed in specific situations, where a pragmatic representation of the XML file is needed to access different parts of the tree at different times. A mature DOM API is able to provide a memory representation for all the components available in an XML file presented in a valid or non-valid format. Diagram 2.1 shows an example of an XML tree and the relationship between nested nodes. Using a tree-based API, users are able to navigate from any node and retrieve information for as long as the tree is stored in memory.

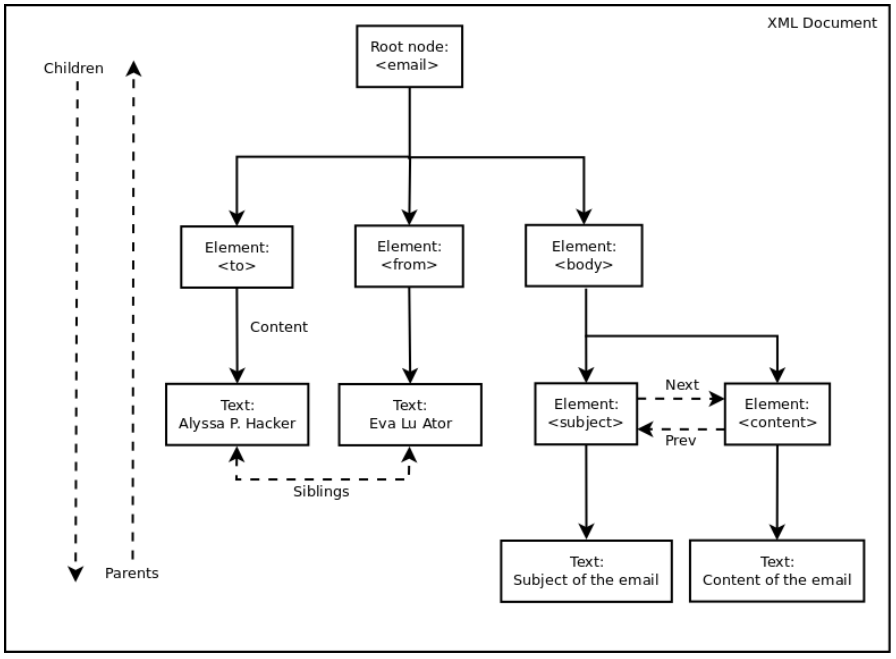


FIGURE 2.1: XML DOM

The difference between a SAX and DOM parser mainly depends on the scenario where these techniques are used. DOM is mostly needed for more complex situations when the document needs to be loaded completely into memory and accessed multiple times. Whereas, a SAX parser is mainly used to retrieve simple information rapidly. The amount of memory needed to use SAX or DOM also depends on the *shape* of the document. A SAX parser needs to store information regarding opening and closing tags in order to keep track of the depth of the XML and report non-valid document. Hence, the memory needed to use a SAX parser is proportional to the depth of the XML document. Instead, for a

DOM parser the amount of memory needed is proportional to the depth and the length of the XML document.

Another significant difference between a SAX and DOM parser is the pre-processing needed to operate the tree. A DOM parser will load the document entirely and return an in-memory representation while a SAX parser operates sequentially without being able to keep track of parsed data. For this reason, a SAX parser tends to outperform a DOM parser in processing efficiency. In most scenarios the full memory representation of an XML document is needed in order to validate the document. For example, programming interfaces for declarative transformation and query languages need to be able to access various parts of the XML tree at different times.

#### 2.1.1.3 XML Query and Transformation

Programming interfaces are also used to render, transform and query XML documents. Extensible Stylesheet Language (XSL) is a family of languages designed to describe how to display a document. The concept of XSL is analogous to Cascading Style Sheets (CSS) applied to HTML web pages. This language provides the basic specifications for XSL transformation and query languages, in most cases, using XML as the declarative language. XSL consists of three main languages, XSL Transformation (XSLT), XML Path Language (XPath) and XSL Formatting Objects (XSL-FO). There are a number of additional XSL family related languages that have been developed for different purposes. XSLT is the main language used to transform XML into other formats such as HTML, XML, XSL-FO, Portable Document Format (PDF), PostScript (PS), Scalable Vector Graphics (SVG) and other formats (Clark, 1999). XPath is used to navigate an XML tree and return information stored in multiple XML components based on the query (Robie et al., 2013).

**XSLT**

XSLT is a Style Sheet Language designed to transform XML document into various formats. Due to the inheritance of XSL, a Style Sheet language aimed to express the presentation of structured XML documents,

XSLT has been mainly design to transform XML into formats with presentational semantics such as HTML. However, this language is able to transform XML data into various formats such as plain text, PDF, PostScript or a different XML structure. XSLT is able to transform documents by removing, adding, rearranging or sorting XML components such as elements, attributes and other declarations. Because of the popularity of XHTML as result of transformed XML, most web browsers support this language according to the W3C recommendation. XML documents can be converted into different formats by defining an external reference to an XSLT document. For example, a structured XML file of a catalog can reference an XSLT document which collects some of the information and displays them in an HTML page. Once opened with a browser, the XML will be converted into the XHTML format and displayed using the HTML presentational semantics.

One of the features available on XSLT is the ability to search for information contained in the XML document. However, this feature is not native to XSLT but is imported from the XPath language. The use of this language allows XSLT to search for information within large XML files adopting XPath as sub-language for information retrieval. One of the issues that has been solved during the development of XSLT is the support for a streaming transformation, a technique to parse an XML document without creating an in-memory representation. As mentioned in the previous sections, a DOM parser loads an entire XML document into memory enabling easy information retrieval with a bidirectional parsing technique. However, due to the implementation of XML in fields such as database storage, constructing large documents in memory requires a large amount of resources. For this reason XSLT 3.0 implemented a streaming transformation and various improvements to handle large documents (Kay, 2012). However, because of the increasing complexity between the three versions, software capable of parsing these documents are mostly based on the XSLT 1.0 specifications[1][2][3].

[1] Veillard, D. The XSLT C library for GNOME - http://xmlsoft.org/XSLT
[2] The Apache Xalan Project - http://xalan.apache.org
[3] Microsoft 2013, XSLT for MSXML - http://msdn.microsoft.com/en-us/library/ms759204.aspx

**XPath**

XPath is a Query Language designed to perform queries on XML documents. This language is used for selecting nodes and returning the information contained within XML components. Because of this important feature, XPath is at the basis of most of the XML-related programming interfaces for easy information retrieval and computation. The expressions used in order to traverse documents are similar to those employed in traditional file systems to navigate between sub-folders. Using these expressions, users are able to navigate between nodes without knowledge of the XML structure. However, knowledge of the document enables users to narrow the query using predicates to select the desired nodes from a list of children. In addition, with a similar DOM implementation, XPath is able to move between XML components, returning the parent node, siblings, children and attributes of a current node. Operators and functions to perform operations on the element data type are enabled on the 1.0 version and most of the programming interface libraries. These features allow XPath to perform queries and additional basic operations on the returned data types.

### 2.1.2 XML Validation

This section provides an introduction to validating XML data. Validation is an important aspect of this research as it allows the compression of XML using the fixed length encoder. Knowledge of the components and structure of validation techniques will be necessary in later chapters when discussing the proposed tool.

This research defines “XML validation” as those techniques that ensure a correct structure and well-formedness of XML documents (Bray et al., 2004). As a verbose language with redundant data, the possibility of users entering erroneous data or using an undefined structure is high. This scenario can also be applied when an XML document is constructed using a specific API by gathering data from users or machines. Additional error checking must be implemented in order to monitor the range of the data that has been passed by a front-end application. Using the components rules defined in the previous section, a valid

XML document should not contain data outside tags or markup characters enclosed without specific components. The document structure must be logically and physically valid in order to meet the XML specifications. XML components explicitly declared using valid markup such as element, PI, comment, declaration etc., form the logical structure of the document. The physical structure instead is constructed by entities such as document entity which defines the starting point of an XML processing.

An XML processor should, at user level, report non-valid XML documents whether the error occurs within the logical structure or the physical structure. It is possible to distinguish between two types of validation processors, validating and non-validating. A validating process will parse both structures reporting errors contained in all the entities and any violations of the specification. A non-validating approach will process both structures, report errors contained in all available entities, however, it will not report a violation of the specification. These approaches are available in most XML programming interfaces providing support to easily enable validation of the XML. While the well-formedness constraints of a document are defined by the XML specifications, validity constraints are defined within additional internal or external structures. The two main approaches to define additional constraints to the data and the structure are Document Type Definition (DTD) and XML Schema (Thompson et al., 2004).

#### 2.1.2.1 DTD

A DTD entity declares the structure of an XML document by defining a list of legal elements. This entity was imported into the XML specification as a subset of the original language SGML where it was used as prologue. A DTD is used to validate the structure of a users XML document or to agree on a specific structure between homogeneous applications over the internet. A DTD can be included in an XML document using an external or internal entity. Using an internal entity, the structure and hierarchy of the document is defined before the root node using the *DOCTYPE* definition. An example of a DTD file to validate the code listing 2.1 is shown below as an internal entity. The external entity defines a link to a physical storage using *DOCTYPE* definition with the path to the external DTD document. Using a Uniform Resource Identifier (URI) the

declaration can link to a DTD on the web that will be downloaded and parsed during the validation process.

LISTING 2.4: Example of DTD

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE email
[
<!ELEMENT email (from,to,subject,body)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT subject (#PCDATA)>
<!ELEMENT body (#PCDATA)>
]>
...
```

From the DTD point of view, the XML document is constructed using a subset of what the document can contain. DTD can recognise and apply constraints on elements, attributes, entities, PCDATA and CDATA. A DTD was primarily designed to provide information to the authors to facilitate the creation and modification of documents. For this reason a DTD file should not be overly complicated so they do not become hard to design and maintain. A DTD provides a set of allowed element type declarations which specifies the possible content of an element node. Structurally, a DTD can apply restrictions to the shape of the XML by defining the allowed sub-elements and their logic. Elements can appear in sequence or choice using comma "," and pipe | characters respectively. Using the +, * and ? characters, it is possible to emulate the logic of BNF document and define when an element can contain one or more, zero or more and one or zero occurrences of elements. These quantifiers can be used next to the element or next to a set of mixed elements. `#PCDATA` is used in order to restrict an element to contain parsed character data. Using `EMPTY` and `ANY` the authors can describe when an element is empty or when it can contain any XML components such as child elements or text.

#### 2.1.2.2 XML Schema

XML Schema is one of the XML schema validation techniques defined as the W3C recommendation (Sperberg-McQueen and Thompson, 2000). Similar to DTD, XML Schema defines the structure and shape of XML documents in order to validate elements and text of the XML. The main difference with DTD is the

language used to define the set of rules to which the XML must conform. While DTD is written in a subset of SGML BNF-like language, XML Schema is XML-based allowing the parser software to use XML parsing techniques. This feature of XML Schema allows users to avoid learning new languages and to use XML editors to manipulate both documents. XML DOM can be used to create an internal tree structure of the XML Schema and to transform it using XSLT. In addition, XML Schema is only available as an external entity, linked to the XML using a URI to a local or web resource. XML Schema is commonly referred to as XML Schema Definition (XSD) in order to avoid confusion related to the use of the same term to define XML schema languages.

The major advantage of XSD is the extensibility that is allowed by the use of this language to validate XML. It is possible to reuse the same Schema to validate multiple documents stored in a database or to reference multiple XSD files in different sections of the XML. Another powerful feature of XML Schema is the ability to define custom data types based on standard types. Users are able to apply additional constraints to standard data types by adding restrictions such as enumeration, total digits number, length of characters, patterns using regular expressions or white space control. For this reason XML Schema is mainly used in those situations where the well-formedness of XML and the element rules defined in DTD are not sufficient. In order to define the structure and data of XML, XSD is composed using simple and complex types.

**Simple Type**

Simple types are used to define those elements that can only contain data in text format. This is the lowest format to which an element can be expressed in by defining the data which is allowed to contain. Simple types can be defined as one of the most powerful features of XML schema as they apply constraints to improve the data format and allow a restricted set of rules. XML Schema contains a set of built-in data types to apply minimum restrictions on string, integer, decimal, boolean, date and time. Using the restriction indicator, users are allowed to apply additional constraints on the data such as enumeration, pattern or length. Elements containing attributes are defined as simple types although the inclusion of attributes defines a complex type. The restrictions available on the element data are also available for attributes and for all the XML declaration requiring data.

**Complex Type**

Complex types are used to define elements containing nested elements, data as text, both nested elements and data, or empty elements. Complex types define how the elements are structured in a nested sequence. XML Schema indicators are used to control the document by defining the order, occurrence and grouping of elements. The indicators `sequence`, `choice` and `all`, are used to set rules on the allowed nested elements and their appearance in the document. For example, the `sequence` indicator allows elements to be displayed in the exact sequence as they are defined in the XML Schema. The `choice` indicator defines a number of elements where only one is to be chosen in the XML. Lastly, the `all` indicator, allows any defined element to be used in any order only once. Occurrence and grouping indicators are used in conjunction with the element to define how often a specific element can appear and group sets of element.

One powerful feature of XSD is the ability of defining global simple and complex types to be later used in one or more instances. However, this feature increases parsing complexity. Because of the verbosity of this language, XSD documents used to validate relatively simple XML files can quickly grow in depth and length due to the redundant markup used to define the various indicators. Using global types it is possible to avoid increasing the depth of the document, in order to reduce maintaining complexity. Documents implementing this design method firstly define simple and complex types and then refer to them using the `ref` attribute within elements. Code listing 2.5 and 2.6 provide an example of XML schema written using local and global types respectively.

Although it is defined as the successor of DTD, XML Schema is still not as popular as its predecessor. The reason for this lack of success is related to the complexity introduced by XML Schema.

LISTING 2.5: Example of XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="email">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="from">
          <xs:simpleType>
            <xs:restriction base="xs:string">
              <xs:maxLength value="40" />
            </xs:restriction>
          </xs:simpleType>
```

```
        </xs:element>
        <xs:element name="to" type="xs:string"/>
        <xs:element name="subject">
          <xs:simpleType>
            <xs:restriction base="xs:string">
              <xs:minLength value="5"/>
              <xs:maxLength value="25"/>
            </xs:restriction>
          </xs:simpleType>
        </xs:element>
        <xs:element name="body" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

LISTING 2.6: Example of XML Schema using global types

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<!-- define global simple types -->
  <xs:simpleType name="nameType">
    <xs:restriction base="xs:string">
      <xs:maxLength value="40" />
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="headingType">
    <xs:restriction base="xs:string">
      <xs:minLength value="5"/>
      <xs:maxLength value="25"/>
    </xs:restriction>
  </xs:simpleType>
<!-- define global complex types -->
  <xs:complexType name="content">
    <xs:sequence>
      <xs:element name="from" type="nameType"/>
      <xs:element name="to" type="nameType"/>
      <xs:element name="subject" type="headingType"/>
      <xs:element name="body" type="headingType"/>
    </xs:sequence>
  </xs:complexType>
<!-- main element-->
  <xs:element name="email" type="content">
  </xs:element>
</xs:schema>
```

#### 2.1.2.3 XML Information Set

The well-formedness and correctness of XML documents is based on a XML Information Set (Infoset). Infoset defines an abstract data model describing a set of rules detailing the properties of XML trees. A pre-defined data model allows API developers to follow one specification, enabling users to switch between different APIs without having to learn new data models. For example, Infoset

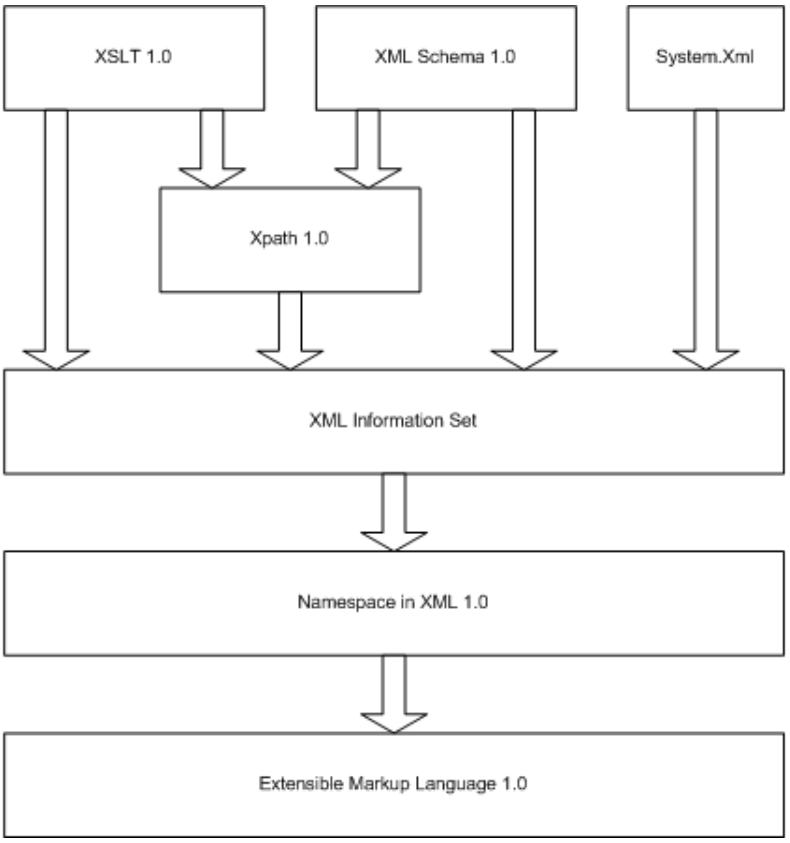


FIGURE 2.2: XML Information Set (Gudgin, 2004)

defines what information is considered relevant in an XML document. Empty elements defined using opening and closing tags or a single self-contained tag, are considered equivalent from the API point of view. The same would apply for the use of different characters, such as the escape codes, which are used for a specific task. Therefore is it possible to visualise an XML processing API and higher-level language specifications such as XSLT, XPath and XML Schema to be based on top of the XML Information Set. The latter will be based on top of XML Namespace specification which is subsequently based on the Extensible Markup Language. Diagram 2.2 provides a graphical hierarchical representation of the various specifications.

### 2.1.3 Metrics and Classifications

The following section classifies XML data into two major categories.

**Document-centric**

Document-centric focuses on the text of XML. Text data is the major part of the document and can appear in any instance of the file. In addition to standard XML components, element tags are mostly used when the document needs to specify part of text, for example when a new paragraph is created. Examples of document-centric XML are documents designed for human reading, i.e. XHTML documents, DocBook, etc.

**Data-centric**

Data-centric is based on XML with a regular structure and an equal balance between data and tags. Element tags are used to divide data types depending on the system requirements. These XML mostly represent structured data and can be used as database representations. In this category, there is a particular type of XML called **structural** documents. These XML are based uniquely on XML structure (element tags) with no data values. These documents are used to define a particular structure and to test the performance of XML-conscious compressors. **Textual** documents, instead, are based on a minimal structure with the XML content ratio consisting mainly of data values.

A further classification can be introduced based on the validity of XML documents. Both **Regular** and **irregular** documents are a challenge for XML-conscious compressors which rely on a schema language. This issue arises when a document does not validate against a schema or standard XML infoset.

## 2.2 Data Compression

The main objective of data compression is to remove unnecessary information by encoding the original data into fewer bits. Data compression is widely used in many areas of computing in order to overcome hardware limitations and improve human interaction. These techniques are now used transparently and are part of our everyday life. Source coding is the original term defined in electrical engineering to describe the process of data compression (Lelewer and Hirschberg, 1987; Shannon, 1948; Wade, 1994). The objective was to remove the inefficient redundancy in order to reduce data transmission time or to reduce the amount of storage required. Overcoming hardware limitations is one

of the challenges of applying data compression. Processing data represented in fewer bits reduces resource usage whether the information needs to be simply processed, stored or transmitted. A distinction between data and information is vital to understanding data compression. Information is represented with data, hence it is possible to convey the same information using different amount of data. A highly efficient compressor can significantly reduce the size of the original data depending on the technique used and the nature of the source. A trade-off usually exists between compression size and compression speed where higher levels of compression consume more processing time. However, the resources required to achieve high rates of compression may not be available on a constrained device. The level of distortion and the resources required to compress and decompress the data are also subject to the same trade-off. For example, compressed video-conferencing data requires high computational resources on both the encoder and decoder sides. If distortion is introduced to improve speed, the quality of the original data is subject to deterioration.

Different fields in computing adopt a specific data compression category. Data compression techniques are divided into two main groups: lossless and lossy. **Lossy** data compression is mainly used in multimedia applications where the quality of the source can be decreased without noticeable differences. These compression techniques are able to achieve the lowest compression size by losing information which can be discarded. Such techniques are usually applied to

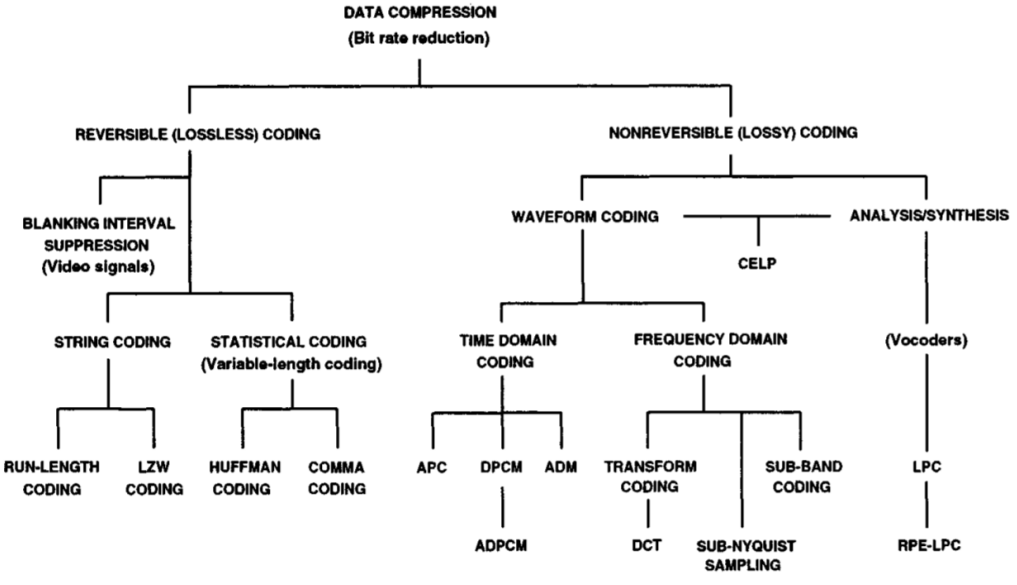


FIGURE 2.3: Data compression techniques (Wade, 1994)

those areas of computing which are highly based on human interaction such as audio, image and video compression. In contrast, **lossless** data compression is applied in scenarios where the decoded data must convey exactly the same meaning as the source. This compression technique is mostly applied to text based information where the decoded version must be the replica of the original. Statistical redundancy is the key to lossless compression techniques. This *inefficient* extra bit of information is exploited in order to form a more concise version of the original data. The statistical nature of the data is also used as a reference to determine the limits of lossless compression. In the field of information theory, this mathematical limit is defined as entropy (Shannon, 1948). Opposite to lossy, the lossless approach is a reversible process which, due to the nature of the compression, cannot achieve a compression size similar to lossy techniques.

Diagram 2.3 illustrates the two branches of data compression named as *reversible* and *non-reversible* coding, each branch expands into various techniques and further into specific algorithms.

### 2.2.1 Fixed and Variable Length Codes

Reversible (lossless) coding techniques shown in figure 2.3 can be categorised into different encoding rules. These rules have been devised to achieve efficient representations using the least amount of information. Fixed and variable length encoding are the two main techniques to compress data in efficient binary formats. Other techniques, for example, are based on generalisation of string coding. Variable length encoding techniques map source symbols into variable number of bits depending on their frequency. Contrary, fixed length coding applies a fixed number of bits for each of the source symbols. This fixed number of bits is directly proportional to the amount of source symbol found in the source data. The following table compares the length codes for the binary representation of alphabet source symbols.

TABLE 2.2: Variable and Fixed length bit mapping

| | **A** | **B** | **C** | **M** | **N** |
|---|---|---|---|---|---|
| **Frequency** | 32 | 23 | 18 | 13 | 11 |
| **Fixed Length** | 000 | 001 | 010 | 011 | 100 |
| **Variable Length** | 0 | 11 | 100 | 1010 | 1011 |

Below there are several examples of fixed and variable length coding techniques to compress the alphabet source symbols below.

``ABACMABACN''

Using the rules defined in table 2.2, it is possible to encode the information above using 3 bits for each of the source symbols. The amount of bits required to create this binary representation is calculated using $\lceil(\log_2(n))\rceil$ function, where $n$ is the total number of unique characters found. To encode 5 different characters a fixed length encoder requires 3 bits of information as shown in table 2.2. The same number of bits would be required to encode up to 8 different characters. Therefore, it is possible to encode the string listed above in 4 octets with 2 additional redundant bits as shown in the following binary representation.

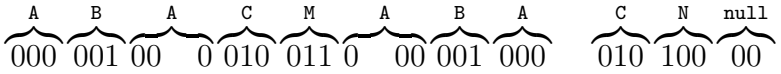


FIGURE 2.4: Fixed length binary representation

Using this approach, very likely and very unlikely characters are represented using the same bits of information. Therefore, the decoding process is a simple operation as depicted in diagram 2.5.

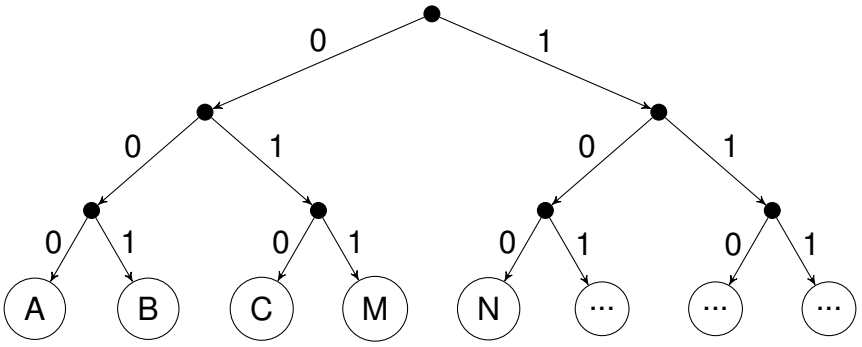


FIGURE 2.5: Fixed Length Coding

Fixed length encoders are suitable to compress data where different source symbols have an equal likelihood of appearing. However, where one or more symbols have a different likelihood of appearing, a variable length coding technique is typically preferred. Using the information of table 2.2, the lowest binary representation can be attributed to the most frequent source symbol or to those more likely to occur. The string listed above can be encoded in 3 octets with 2 additional redundant bits as shown in the following binary representation.

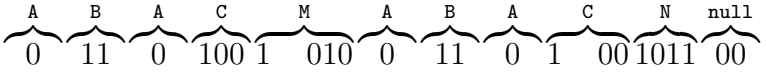


FIGURE 2.6: Variable length binary representation

The main challenge of this encoding mechanism is the reconstruction of the binary representation. Defining source symbols with variable length codes can result in decoding errors. For example, assigning code `01` to symbol `A` and code `0101` to symbol `B`, would result in multiple interpretation for bit stream `010101`. To solve this issue, basic variable length encoders implement prefix codes (Shannon, 1948). With this techniques, no codeword is a prefix of another codeword thus simplifying the decoding process as shown in figure 2.7.

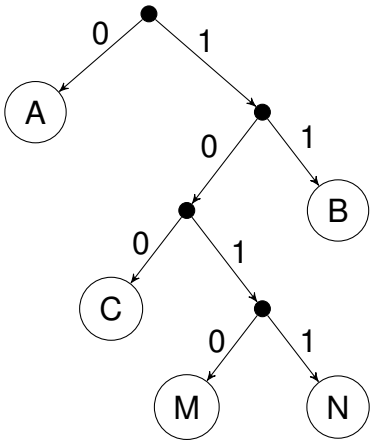


FIGURE 2.7: Variable Length Prefix Coding

The idea around prefix codes is to find the optimal coding tree to minimise the amount of bits required for the most common codewords. Frequent source symbols require shorter codewords in contrast with rare source symbols. Various optimisation techniques have been devised to construct an optimal coding tree. The most popular is the Huffman's coding technique which involves a bottom-up approach to construct the optimal tree (Huffman et al., 1952). To highlight the performance of Huffman coding additional source symbols have been added to the previous string example.

``ABCBABMANBCACDBDEEA''

A Huffman coding tree is created by joining less frequent source symbols as leaves thus assigning shorter codewords to more frequent ones. The frequency table for the above string is provided below.

TABLE 2.3: Frequency Table

| **Source symbols** | A | B | C | D | E | M | N |
|---|---|---|---|---|---|---|---|
| **Frequency** | 0.3 | 0.25 | 0.15 | 0.1 | 0.1 | 0.05 | 0.05 |

The frequency table 2.3 needs to be stored in the compressed format to reconstruct the tree. The following figure analyses the generated binary representation constructed using the Huffman coding.

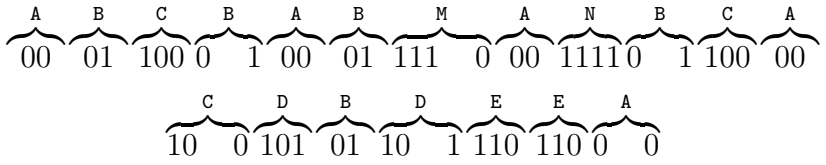


FIGURE 2.8: Huffman binary representation

Figure 2.9 presents the coding tree used to decode the bit stream. During decoding, the tree is reconstructed using the frequency table information returning the original format.

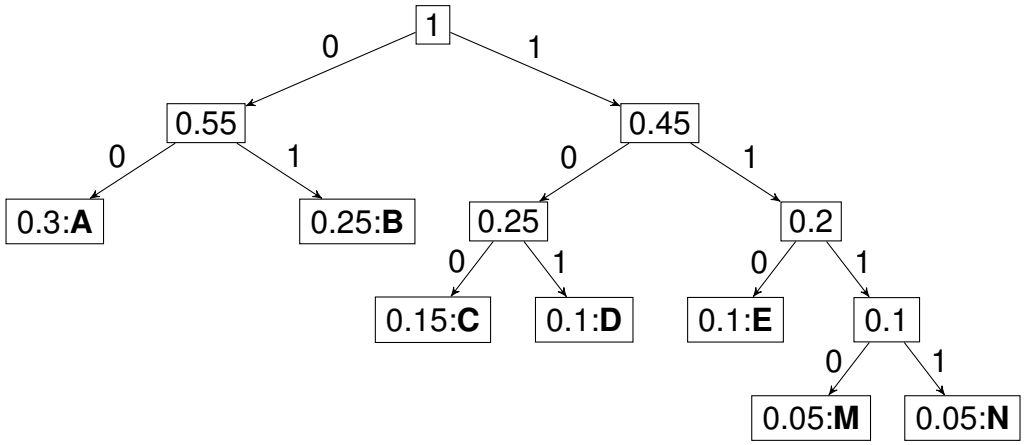


FIGURE 2.9: Huffman Coding

### 2.2.2 XML Compression

Section 2.1 discuss the structure of XML and how it can be used in a range of different fields. As the popularity of this language continues to grow, the demand for processing and storing has improved leading to a number of interesting results (Arion et al., 2007; Boncz et al., 2006; Cheney, 2006b; Ferragina et al., 2006; Liefke and Suciu, 2000; Wang et al., 2007; Zhang et al., 2004). Software capable of handling XML data efficiently has led to many advantages. The most known and exploited are listed below.

- To minimise the size of the data to reduce disk space and network bandwidth required to exchange XML data.

- To minimise the memory consumption required to process XML using efficient DOM parsing.
- To efficiently querying both compressed and uncompressed XML data.

XML compression refers to those techniques and algorithms which take advantage of the verbose and redundant structure of XML to improve the compressed size. Because of the nature of XML as markup language that needs to be processed by machines, most of the available XML compression techniques focus on lossless compression algorithms. However, lossy and near-lossless techniques have also produced interesting results (Cannataro et al., 2001).

Traditional compression techniques described in diagram 2.3 can be successfully used to compress XML data. However, an XML specific compression technique is able to apply more compression beyond a typical lossless technique. Knowledge of the structure of the XML allows techniques based on prediction and statistical algorithms to apply a better compression. Similar work has been conducted in the field of XML parsing. As described in section 2.1 a DOM API needs to load an entire document into memory before being able to traverse the tree. In the case of a large XML document, this process can use more memory than the system resources can afford to allocate. Hence, several researchers have proposed a more efficient parsing technique to reduce the memory needed to create an in-memory representation of XML (Delpratt et al., 2008; Wang et al., 2007).
A growing number of XML compression techniques have been proposed to solve challenges related to compression ratios and computational resources required (Augeri et al., 2007; Buneman et al., 2005; Cannataro et al., 2001; Harrusi et al., 2006; Levene and Wood, 2002; Ng et al., 2006a; Sakr, 2008, 2009; Skibinski and Swacha, 2007; Toman et al., 2004). These techniques can be categorised into three main categories listed in the following sub-sections.

#### 2.2.2.1 General-purpose

As described in section 2.1, XML data is stored as a text file. General-purpose compression techniques are therefore the first basic approach to compress XML by treating the data as a normal text file (Cleary and Witten, 1984; Gailly and Adler, 1999; Seward, 2000). These techniques are able to reduce up to 70% the

size of the original file (Sakr, 2009). The main advantage of these approaches is the large amount of research conducted in the field of data compression, prior to the work applied to XML data. Therefore, although the compression size is not optimised for XML data, tools based on general-purpose compression techniques are able to achieve faster compression times (Sakr, 2009). A number of algorithms have been devised over the last decade to compress text data efficiently. The most popular and efficient algorithms to which XML compressors are usually based and tested against are gzip(Deutsch, 1996b), bzip2(Seward, 2000) and PPM (Cleary and Witten, 1984).

Gzip is based on the DEFLATE lossless algorithms which is a combination of the Abraham Lempel and Jacob Ziv of 1977 (LZ77) and the Huffman coding techniques (Deutsch, 1996a). LZ77 is a dictionary coder which compresses data by replacing multiple occurrences of the data with references to a unique copy that was already found in the uncompressed stream (Ziv and Lempel, 1977). Huffman coding is based on a variable length code table where the shortest code is assigned to the most common source symbol (Huffman et al., 1952). A combination of LZ77 and Huffman coding is used for the DEFLATE algorithm which is at the basis of many software and hardware encoders. The Lempel–Ziv–Markov chain algorithm (LZMA) is based on LZ77 and features a higher encoding ratio compared to other lossless compression algorithms (Pavlov, 2015). LZMA is at the basis of the 7z format used for 7-ZIP[4], a relatively new open source file archiver.

Bzip2 is based on several layers of compression techniques with the Burrows-Wheeler algorithm at its core. The Burrows-Wheeler transforms (BWT) character strings by changing the order of the characters where the result will have several repetitions of single characters in a row. This algorithm allows an increase compression by repeating the number of characters of a string. Bzip2 is often compared to gzip and other variants. The compression size is more efficient than compressors based on DEFLATE algorithm, however bzip2 is slower in compression time (Sakr, 2009).

PPM is an adaptive statistical data compression technique based on context modelling and prediction and it is at the basis of many XML-conscious compressors. This algorithm is able to predict the next symbol in the stream using a set of previous symbols found in the uncompressed stream. PPM is a simple

[4]The 7-zip file archiver - http://www.7-zip.org

and efficient compressor, however due to the nature of the algorithm it is also the most computationally expensive.
Gzip, bzip2 and PPM are three of the most common compression algorithms that are used at the basis of XML compressors. To evaluate the performance, these tools are usually compared against other compression algorithms to test compression size improvements.

#### 2.2.2.2 XML-conscious

This research defines *XML-conscious* techniques as those that consider the structure of XML to achieve a better compression ratio. XML-conscious techniques are optimised to work with XML and take advantages of the redundant data to achieve higher compression ratios. In addition to the language format, these techniques are also able to exploit the awareness of XML Schema languages in order to apply additional compression. Therefore, these methods can be classified as schema informed and schema uninformed according to their ability to access a schema language.

**Schema uninformed**

Schema uninformed refers to those techniques which consider the XML language when applying traditional compression techniques. Based on the structure and the simple design of the language it is possible to achieve a better compression compared to text compression techniques (Cheney, 2001; Liefke and Suciu, 2000). This is possible due to the tree structure non-local redundancy which, in the case of a structured XML document is static and highly predictable.

**Schema informed**

A more advanced technique is able to access a predefined XML schema language and apply more compression based on the knowledge of structure and data types of the XML. Several studies (Cheney, 2005; Girardot and Sundaresan, 2000; League and Eng, 2007b; Subramanian and Shankar, 2006) have developed techniques to apply additional compression using Schema languages such as DTD and recently XML Schema. For example, by defining the types of XML elements that can be used in

a specific node, it is possible to predict the structure and avoid encoding data that is defined in the schema language. In the case of an XML schema, it is also possible to define constraints to the data type and recognise data such as a MAC address which can be then treated as a group of two hexadecimal digits. Some issues are related to the availability of the schema language when compression is used across heterogeneous networks. Both the sender/encoder and the receiver/decoder must have the same schema language in order to exchange the document.

#### 2.2.2.3 Queriable

Queriable techniques refer to a branch of XML-conscious compression capable of performing queries over the compressed format (Buneman et al., 2003; Ferragina et al., 2006; Ng et al., 2006b; Tolani and Haritsa, 2002). This ability is vital for those resource-restricted applications and constrained devices which cannot afford to decompress the entire document. Because of their nature, compression ratios are worse compared to XML-conscious and general-purpose text compression. However, the main purpose of these techniques is to avoid compressing entire documents in order to perform queries at compression speed and time costs. These categories are used in accordance to the environment where they have been applied. As mentioned in the previous paragraph, some resource-limited systems might not be able to afford to decompress an entire document and perform queries because a DOM parser might require an excessive amount of memory. At the cost of additional processing time, Schema-informed techniques are able to achieve higher compressed formats which are highly desirable when network bandwidth is the true bottleneck of the system. Alternatively, general-purpose techniques are used for their fast memory-efficient encoding mechanisms. However, as it will be discussed further towards the end of this chapter, the nature of the document can have a major impact on the efficiency of XML compression.

### 2.2.3 Features and Classification

Previous section categorises several compression techniques based on their ability to support XML queries and schema languages. The main objectives of

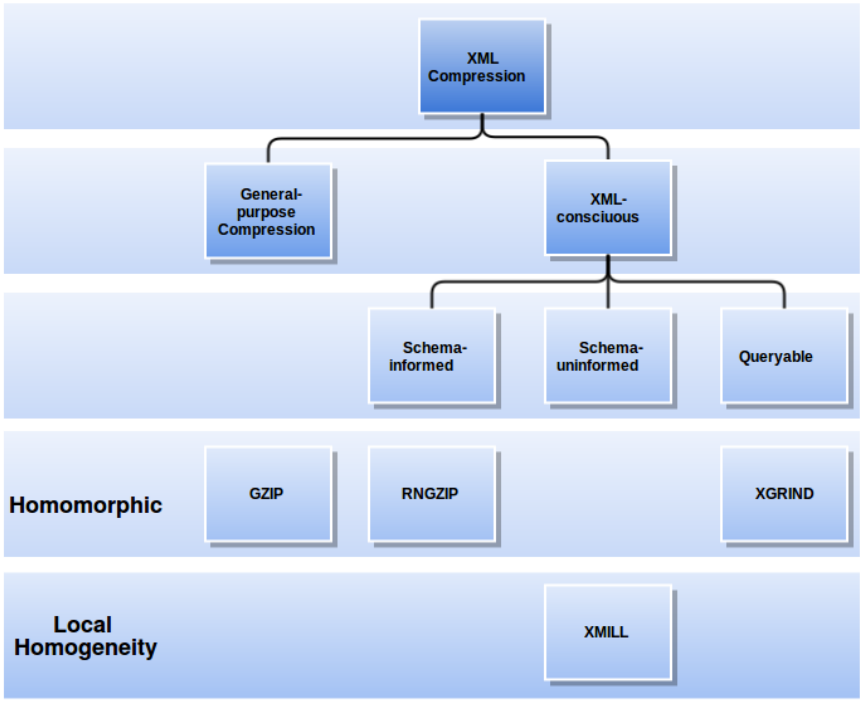


FIGURE 2.10: Features and classification of XML Compression

these techniques have been identified together with the areas which are currently being exploited. Further features and classifications can be introduced based on software and its ability to process XML, considering data and structure as two separate components. As discussed in section 2.1, markup is highly distinguishable from the main text. In addition, due to the simple and efficient structure of valid XML documents, it is possible to concisely represent XML into different formats. This led to the idea of separating XML data from the structure. This idea has produced interesting results particularly with the use of the XML schema language to support compression.

#### 2.2.3.1 Homogeneity and Homomorphism

Diagram 2.10 illustrates the various classifications of XML compression techniques with the corresponding software implementations. The diagram includes two further classifications for specific software. These classifications are based on the software ability to treat XML data and structure as two separate components. Practically, each XML-conscious technique can be further classified based on how XML data is processed. Two main classifications are listed below.

**Local homogeneity**

Techniques based on local homogeneity properties treat XML data and structure separately. Based on their path and data type, data values are stored into semantically related *containers* which are then compressed separately. Diagram 2.11 shows an example of using the local homogeneity implementation to compress an XML document. A binary codeword is assigned to the XML structure such as start tags and attributes. This powerful feature was firstly introduced by XMill (Liefke and Suciu, 2000) to compress XML documents more efficiently. This idea has been later expanded to improve compression by introducing a container language to apply custom policies to separate data values into multiple components. However, this aspect of the system highly relies on human intervention and its document's data type knowledge to increase compression. Storing data values into semantically related containers is the major advantage of local homogeneous applications. General-purpose compression is applied to the set of containers based on the nature of the data. Each container can be compressed using different compression schemes. This allows compression to be applied to a group of semantically related data values and increase the overall compression. Furthermore, more specialised compression can be applied to specific data types such as date, MAC address, IP or URL formats.

**Homomorphic**

Homomorphic compression techniques retain the original structure of an XML document. Differently from homogeneous techniques, the final data format of this approach preserves the document structure and data values. The structure preservation allows this technique to perform additional operations that can be executed on the compressed format. This feature was

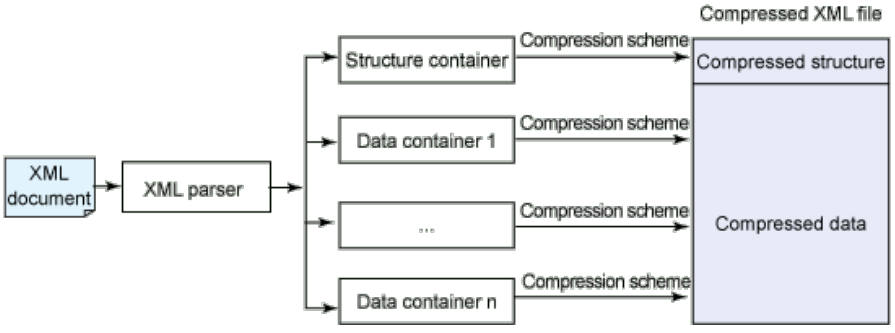


FIGURE 2.11: Local Homogeneity of XMill compression (Sakr, 2011)

initially introduced by the XGrind compressor (Tolani and Haritsa, 2002), which also allows queries to be performed on the compressed format. In addition to queriable application, homomorphic compression techniques allow indexing and updating on the compressed format. Code listing 2.7 and 2.8 are examples of how a homomorphic application encodes XML. The XML document in code 2.7 is compressed using code 2.8 format, where the structure of XML is replaced with binary codewords and the data values are encoded using the application default compression algorithms. A powerful feature for applications with homomorphic properties is the ability to use a schema language to validate the XML documents. This allows applications that are required to perform validation and queries on XML to compress and extract data more efficiently. Because of the nature of these techniques, compression ratios for techniques with homomorphic properties are lower compared to different approaches.

LISTING 2.7: Example of XML document

```
<?xml version="1.0" encoding="UTF-8"?>
<email date="08/10/13">
  <from>Alyssa P. Hacker</from>
  <to>Eva Lu Ator</to>
  <subject>Subject of the email</subject>
  <body>Content of the email</body>
</email>
```

LISTING 2.8: Example of Homomorphic compression

```
T0
  A0 encode(08/10/13)
  T1 encode(Alyssa P. Hacker) /
  T2 encode(Eva Lu Ator) /
  T3 encode(Subject of the email) /
  T4 encode(Content of the email) /
/
```

#### 2.2.3.2 Online and Offline Compression

A final classification is based on the ability to operate in an online or offline manner. **Online** compressors are able to stream the compressed format to the decoder, which is capable of decoding the stream without the need of receiving the entire file. On the other hand, **Offline** compressors do not operate on a stream basis. The decoder must receive the full compressed format in order to be able to decompress the document. A number of researchers (Muldner et al., 2012; Müldner et al., 2005) have suggested the use of online and offline compression for specific areas. In more detail, the use of online compressors was found effective to decrease the network latency and improve scalability. The use

of offline Schema-informed compressors, instead, has proven to achieve compression ratios beyond Schema-uninformed approaches. Therefore, this techniques have been adopted in scenarios where minimising the size of the data that needs to be communicated or stored is crucial. Offline compressors are also used in scenarios where the data needs to be validated against a schema language. In conclusion, use of one compressor over another highly depends on the scenario where XML compression is applied.

## 2.3 Summary

This chapter provides the technical background on markup languages and data compression for XML. The first part discussed how XML became a standard for data storage and exchange over the Internet thanks to its extensibility properties and human-readable format. Various components of XML have been discussed by providing simple examples to specify the use of one particular component for a specific task. Basic components such as element and attributes have been covered together with more advanced ones such as PI and namespaces. This knowledge provides a good understanding of how XML components work to form structured documents with hierarchical properties. This chapter discussed the use of application programming interfaces API to extract knowledge from the XML document which are simply presented as standard text files.

Event-based and tree-based APIs are the two main approaches for XML processing used in different fields depending on the requirements of the application. Issues related to processing times and system resources required to process XML documents have been highlighted for both event and tree-based APIs. Extensible Stylesheet Language XSL is introduced as a family of languages designed to describe how XML information is presented. Two major derivatives of XSL, implemented in later chapters, are discussed as additional APIs. The first section on the chapter also focused on XML validation techniques. A number of techniques used to ensure a correct structure and well-formedness of XML documents have been defined. Examples for DTD and XML Schema are provided to demonstrate the difference between these two major validation techniques and the features of each language. This section discussed two types of validation techniques and how XML processors work

on reporting non-valid XML documents with errors on the logical and physical structure.

The second part of this chapter introduced data compression with particular emphasis on lossless compression algorithms. Examples of fixed and variable length coding techniques are provided in order to demonstrate the encoding/decoding processes. Different techniques which have been used to compress XML documents are discussed. General-purpose techniques are the first basic approach to compress XML by treating data as normal text file. By investing more processing time, XML-conscious techniques exploit knowledge of XML structure and data types to increase the compactness of the compressed format. Finally, this chapter concludes by defining various features and classifications of XML-conscious compressors which will be evaluated in later chapters.

# Chapter 3

# XML Compressors and Analysis of XML Data

This chapter is divided into two main sections. The first section investigates the architecture and implementations of various XML compression tools. The advantages and disadvantages of each major compressor are discussed using features and classifications described in Chapter 2. Special attention is given to tools with higher complexity and deployed implementations. This section describes the most influential technologies from which current compressors have evolved, identifying the additional benefit that each tool has introduced. A number of additional XML compressors, classified as query-friendly and back-end encoders, are discussed. Furthermore, a summary of the work analysed in this section is provided. Each tool is classified according to the specification described in the previous chapter and thoroughly analysed to highlight the software complexity. Finally, this section revisits the research goals and objectives of the thesis.

The second section of the chapter focuses on the analysis of XML data. A number of research papers are evaluated to understand the statistics for different types of XML documents. Based on these results, this section presents an analysis of current structures for XML documents and their validation languages. The chapter concludes with an analysis of XML compressors based on knowledge of XML data.

## 3.1 XML Compressors

A number of compressors have been developed over the last decade to overcome the issues related to the verbosity and redundancy of XML. This section describes a number of prominent XML compressors including work based on the extension of compression algorithms and serialisation formats. This work focuses on tools written in native languages with code publicly available. The scope of this research is related to the ability of running an application over multiple machines including low-powered devices based on reduced instruction set computing (RISC) and advanced RISC machines (ARM) to more complex instruction set architectures. Applications evaluated in this section are general-purpose, XML-conscious and schema-informed using one or many back-end algorithms to tackle specific issues related to the use of XML. For each tool this study describes the domain where the application is introduced and the compression ratio improvements achieved on its data set. Finally, the findings are summarised by revisiting the properties and results of each tool.

### 3.1.1 XMLPPM

XMLPPM (Cheney, 2006c) is one of the most influential statistical approaches to compress XML data. XMLPPM is a streaming compression tool that uses Prediction by Partial Matching (PPM)(Cleary and Witten, 1984), a data compression technique based on prediction (Cheney, 2006b). This model is based on an adaptive statistical technique defined as Multiplexed Hierarchical Modelling (MHM). Using a local homogeneous format, XMLPPM is able to achieve higher compression rates with noticeable improvements in processing time compared to other statistical compression techniques. However, although it has been shown that PPM back-end compressors are efficient, these tools usually use more computation resources (Augeri et al., 2007; Sakr, 2008, 2009). PPM can compress between 10-25% more than dictionary based compression tools (Sakr, 2009). Using the *eXpat*[1] XML parser, this tool generates a stream of events which are subsequently encoded using a byte-code representation. XMLPPM is based on two main concepts: Encoded SAX (ESAX) and the use of separate PPM models. ESAX is a more succinct representation of a SAX

[1] The Expat XML Parser - http://expat.sourceforge.net

encoding which unlike other representation is able to perform encoding and decoding *online* in order to process data incrementally. Four PPM models are maintained for elements, attributes, characters and miscellaneous data respectively. Due to model splitting used to aid prediction, accuracy and the local homogeneous property of XMLPPM are lost. To recover, element context symbols are injected into the corresponding modules.

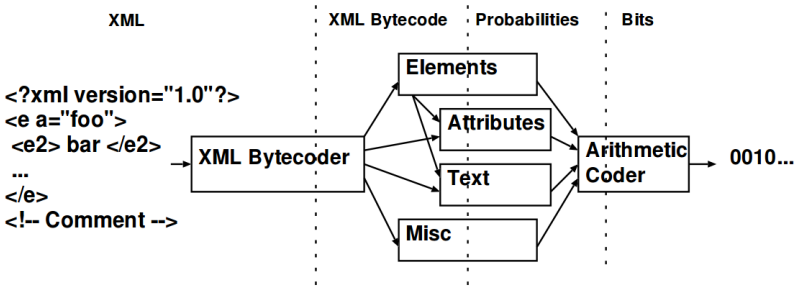


FIGURE 3.1: XMLPPM Architecture (Cheney, 2005)

Figure 3.1 illustrates the architecture of XMLPPM divided into the three main stages. XMLPPM demonstrate that the knowledge of the XML is essential to achieve higher compression rates over PPM compressors. First of all, element and attribute names are tokenized, improving speed for the low-level compressor. In addition, the use of models and XML knowledge to differentiate statistical characteristics is found effective to improve compression. However, this level of optimisation to apply knowledge of XML comes at a cost. XMLPPM is considerably slower compared to *gzip* and *bzip2* compressors.

In summary, XMLPPM applies ESAX stream and prediction by partial matching using different models. Using a specific encoding scheme, it is possible to compress XML data efficiently. However other encoders can also be used to produce similar results(Cheney, 2001). Based on the experiments performed, this technique is able to improve compression between 5% to 30% more compared to other tools with homomorphic properties.

### 3.1.2 DTDPPM

DTDPPM is an experimental extension of the work conducted on XMLPPM which offers compression improvements for relatively small and highly-structured XML files by incorporating extra knowledge from a DTD (Cheney, 2005). This work focuses on the use of DTD to improve compression based on the structure knowledge of XML documents. DTD was chosen over other schema languages because of its simplicity and wider use during the tool development. As discussed in the previous chapter, DTD is still widely used today due to its simplicity and the complexity introduced by its successors i.e. XML Schema. In addition, these tools highly depend on XML parsing libraries which offer full built-in support for DTD.
DTDPPM introduces four levels of optimisation on top of XMLPPM due to the use of a schema language and the possibility to validate XML during compression. These optimisations are listed below.

**Ignorable whitespace stripping**

Whitespaces are often irrelevant for XML files. A structured XML file will contain a number of whitespace outside the element tags used for indentation purposes and to understand the document's hierarchical structure. DTDPPM defines these as ignorable whitespaces. This feature is allowed thanks to the use of a DTD language which provides knowledge of XML. Whitespace removal is essential to the compressor in order to prevent losing track of the context. Additional stripping is performed in the presence of an element containing whitespaces which are not mentioned in the DTD. Using the parser library, whitespaces and newlines are then recreated during decompression. This optimisation is optional, enabling users to preserve structural and content whitespaces.

**Symbol table reuse**

This second level of optimisation is specifically aimed for structured documents. Small XML documents are usually those that suffer poor compression ratios in most general-purpose and XML-conscious techniques. This is due to the additional information, such as frequency table, general purpose algorithms include into the compressed format. The symbol table reference needed in XMLPPM to be dynamically built by the encoder and

decoder can be referred from the DTD. Using this feature, it is possible to avoid transmitting the symbols inline and make full use of the DTD.

**Element symbol prediction**

This feature is based on the idea of omitting symbols that can be predicted from the context. In the case of highly-structured data, parent elements have a specific list of children allowed sequentially. In this case the byte code of the predicted child element is omitted as the decoder can extract it from the context. Although this technique may seem a natural way of exploiting the DTD knowledge, DTDPPM fails to present significant results for this optimisation feature.

**Bitmap-based attribute list encoding**

The last optimisation introduced by DTDPPM is aimed at the use of the DTD file to improve compression for attribute components. Based on the idea that attribute value pairs are irrelevant to the presentation of XML, attribute lists can be rearranged to improve compression. Using DTD declaration of attributes, DTDPPM demonstrated the possibility to use attribute type and specifications such as **#FIXED**, **#REQUIRED** and **#IMPLIED** to encode attribute lists indicating which attributes are present and subsequently send the corresponding values.

The performance of DTDPPM is evaluated on five corpora differing by size and structural content (Cheney, 2005). The results demonstrate compression improvement for small, highly-structured XML files. However, for large documents DTDPPM does not show improvements over XMLPPM. Similar to XMLPPM, this schema-informed approach is considerably slower compared to general-purpose compressors.

#### 3.1.2.1 XMLPPM Extension

Variants of XMLPPM and DTDPPM have been developed, however, their efficiency varies according to the scenario where these tools have been applied (Cheney, 2006b; Skibinski and Swacha, 2007). Most of these tools are aimed at document compression and do not provide an API for further use. SCMPPM

(Adiego et al., 2004) is a variant of XMLPPM based on Structural Context Modelling (SCM). None of the work based on XMLPPM manages to show a significant level of improvement over XMLPPM which remains the best approach to compress XML using multiplexed hierarchical modelling.

### 3.1.3 XMILL

XMill (Liefke and Suciu, 2000) is the first tool to introduce an XML-conscious compressor with local homogeneity properties discussed in Chapter 2. Similar to XMLPPM, XMill is based on existing general-purpose compressors in order to encode data efficiently using the best available compression algorithms. In addition, XMill introduces the ability to compress data types with specific compressors based on user knowledge of XML. Results demonstrate a substantial increase in compression with almost half the size of general-purpose compressors without drastic effect on compression speed. XMill successfully demonstrate the possibility of including XML-consciousness on top of general-purpose compressors without a negative impact to overall speed. This level of abstraction is achieved thanks to the extensive knowledge of the back-end compressors and their ability to achieve higher compression rates when presented with homogeneous data. Exploiting the self-describing nature of XML, XMill decides which compression algorithm to apply using XML tags. Results demonstrate that by using knowledge of the data types, it is possible to translate raw data into XML structures and improve compression. The unusual idea of expanding data to a structured markup language has demonstrated effective compared to general-purpose compressors. For example, instead of being treated as strings, data such as IP, date and fixed content types can be compressed more efficiently. IP can be stored as four bytes, and fixed content types can be factored out using enumeration.

XMill compression is based on three principles. The first principle is the **separation of structure from data**, structure such as opening, closing tags and attributes are separated from the data which consists of the elements and attribute values. After separating structure from data, XMill **groups related data items** into containers which are compressed individually. For example, all the data values from the `<sender>` element are grouped into a single container and

compressed. A **semantic compression** is finally applied to each of the container individually. This specialised compression based on the nature of the data values stored in the container, can be optimised with the aid of a *container expression language*. The user is able to group data based on its knowledge of the XML data type and apply semantic compression.

Compared to other compressors, XMill has a wider range of application. The tool is targeted for two different areas: data exchange and archiving. It is possible to improve network bandwidth and reduce the space required to store data by compressing XML. However, the clear limitation of the XMill approach is the lack of positive results for small data sets. Results demonstrate lack of improvement for XML files with sizes below 20KB due to the bookkeeping overhead and the lack of an efficient back-end compressor to encode small containers efficiently. For this reason, XMill can be categorised as a tool more suitable for data archiving rather than data exchange.
Figure 3.2 illustrates the architecture of XMill. The XML file is parsed with a SAX parser which streams tokens to the core of XMill. The *path processor* is the main component which determinates where to send each token according to its nature. Elements and attribute tags are stored in the *structure container*,while data values are stored in others according to the nature of the data. Users can apply knowledge of the data types and specify a compression algorithm using the *container expression*. When a container reaches the size of 8MB, the data values are compressed and stored on disk. Code listing 3.1 and 3.2 show an

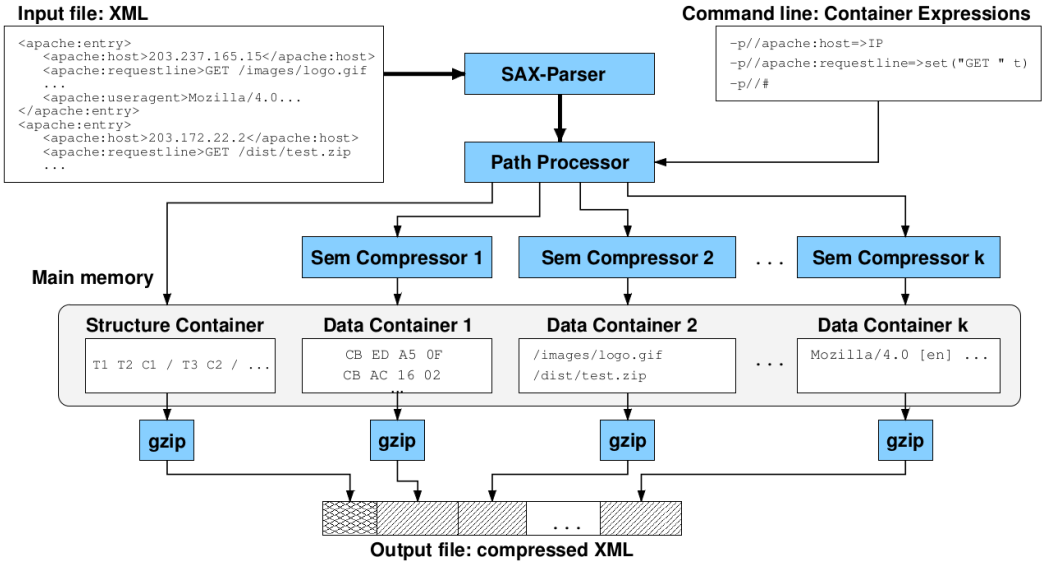


FIGURE 3.2: XMILL Architecture (Liefke and Suciu, 2000)

example of compressing XML data and how the separation between structure and content is maintained. The typical XML example used in this chapter has been modified to demonstrate the use of the same container to compress the *message* element.

LISTING 3.1: XML document

```
<?xml version="1.0" encoding="UTF-8"?>
<email date="08/10/13">
  <from>Alyssa P. Hacker</from>
  <to>Eva Lu Ator</to>
  <message>Subject of the email</message>
  <message>Content of the email</message>
</email>
```

LISTING 3.2: XMill Compression

```
T1 T2 C2 /
     T3 C3 /
     T4 C4 /
     T5 C5 /
     T5 C5 /
 /
```

In summary, XMill is a powerful tool which applies existing compression algorithms to XML. Using the *zlib* library as the main engine, it combines knowledge of the XML data with semantic compression to achieve higher compression rates.

### 3.1.4 WBXML

Wireless binary XML (WBXML) was developed to improve performance over wireless networks (Alliance, 2001; Martin and Jano, 1999) by reducing bandwidth and the resources needed to process XML messages (Augeri et al., 2007; Pak and Park, 2012; Werner and Buschmann, 2004). WBXML specification defines a compact binary representation of XML data. This allows narrowband communication channels to transmit data effectively by reducing the size of the original data. Differently from XMill, WBXML is aimed specifically for data exchange with multiple software implementations (Jehanne, 2009) and libraries (Bell and Jehanne, 2006) developed.

In contrast with compressor tools evaluated so far, WBXML does not depend on a general-purpose compression library. The Wireless Application Protocol (WAP) forum designed a specific algorithm to parse XML into a tree structure, extract elements data and transmit it in accordance to a state machine at both ends. WBXML is a schema-informed approach to compress XML. It highly depends on the knowledge provided by the DTD file to encode data efficiently. In

addition, due to the common token string table in the preamble, it cannot be streamed or pipelined. In summary, two major drawbacks can be highlighted with the WBXML encoding technique. First of all, the algorithms only encode the element tags, attributes and attribute values and it does not compress data between element tags. The second disadvantage of WBXML is the common string table in the preamble which interfere with online compression as both the encoder and the decoder must share the same string table. For this reason WBXML cannot perform well for highly-structured documents.

### 3.1.5 *zlib*

*Zlib* (Deutsch and Gailly, 1996) is one of the most popular general-purpose compressors available today. This library is widely used in a variety of software for commercial application and open-source projects. Zlib is the first approach to compress XML data using a general-purpose tool. This library allows to build applications without major dependencies to be used across different platforms. Zlib is the basis of various *high-level* XML-conscious and archival compressors which apply knowledge of the library and the data to enhance compression (i.e. XMill). *gzip* (Deutsch, 1996b) is a well-known compressor based on the zlib library. Because of its pervasiveness, either zlib or gzip are the standard tools to which proposed compressors are compared and evaluated. Zlib is able to compress files such as documents but also multimedia files such as video and images (*libpng* depends on zlib for data encode and decode routines (Schalnat et al., 2002)). This pervasive feature of zlib is possible due to the way it compresses files. It treats data as a stream of bytes without associating semantics with the content. Light-weight implementations of this *XML-blind* compressor (Adler, 2005) are capable of achieving excellent compression ratios. Compared to XML-conscious compressors, zlib is able to achieve the highest encode and decode speeds in most of the experiments described in various works (Sakr, 2009). Depending on the nature and size of the XML, these tools are able to achieve similar encoding and decoding times.

The major drawback of using zlib to compress XML data is the lack of document semantics. Although zlib is able to compress data efficiently, it is not able to manage non-local duplication, which is highly frequent in XML. This property is based on the presence of redundant data in different parts of the document

such as the opening and closing tags of the root element. In addition, XML data is not interpreted as a markup language with a range of data types, but is compressed as strings. This is the issue XMill tried to solve by introducing semantic compression. It demonstrates the efficiency of sorting data into various containers and compress it separately based on the relational database *column-wise* compression idea (Iyer and Wilhite, 1994). In addition, with the introduction of atomic semantic compressors, it is possible to recognise data types such as integer and apply binary encoding.

### 3.1.6 EXI

Efficient XML Interchange (EXI) is the approach recommended by the World Wide Web Consortium (W3C) (Fablet and Peintner, 2012; Kamiya and Bournez, 2012; Schneider and Kamiya, 2011) to compress XML data. Using data type information and constraints defined in the Schema file, EXI can significantly reduce the size of an XML file using an algorithm that can determine what will occur at any point of an XML file. This schema-informed compression utilises either XML Schema, RELAX NG schema or DTD to optimize compression.

#### 3.1.6.1 Design principles

EXI is designed on a basic algorithm to encode and compress XML data efficiently. The generalised application of this algorithm is theoretically able to encode any language that can be described by grammar. EXI is optimised to work with XML using its own built-in grammar with the possible aid of a schema language. Using the features and classifications of XML compressors described in the previous chapter, EXI can be described as both Schema-informed and uninformed compressor, based on the availability of the XML schema. Since this implementation requires the full compressed format in order to perform decompression, EXI is only capable of performing as an offline compressor. Using a schema-uninformed compression, XML data is encoded using a homomorphic compression technique. Both structure and data are compressed into byte codes and optimised using string tables.

The main objective of EXI is to generalise the number of applications that can communicate with XML data using an efficient encoding. The idea is to have a minimal system and an elegant approach to expand the use of EXI to low-powered embedded devices. Efficiency and flexibility are important design principles to allow this level of optimisation. This flexibility allows any XML document to be compressed without the risk of corrupting data. This feature is designed to work for fragments of XML and documents containing schema language deviations and user-defined data types for efficient encoding. EXI enables XML to be efficiently used in a variety of environments. The light-weight implementation of EXI has been used to solve issues related to network bandwidth (Castellani et al., 2011). EXI Schema-informed approach managed to encode up to 97% the size of the original data, providing a reasonable performance for the constrained payloads available on 6LoWPAN and similar networks (Shelby, 2010).

The basic concept of EXI is based on a hybrid approach using formal language and information theory to devise a relatively simple algorithm. Here, a grammar-driven approach is used to determine the likelihood of XML components to appear at any given point. Subsequently, the EXI stream encoder maps these components to a stream of events using a set of event-codes (EC). To increase compression, EXI events can be passed to the compression algorithm. This algorithm is able to reduce the overhead, creating a compressed stream by combining smaller channels.

#### 3.1.6.2 Architecture

**Data type Representations**

EXI defines a set of built-in data types to represent data sets ranging from low to high-level formats. The use of these data types is triggered only in combination with a *strict* schema-informed approach. XML Schema data types provide information regarding the value of each element enabling a better encoding mechanism. High-level formats are converted into the lowest encoding scheme. For example, the Boolean data type is represented using an n-bit unsigned integer where “1” represents true and “0” false. Higher level-data types are also represented using a mixture of lower types such as unsigned integers and boolean. The same approach of breaking down high-level formats to multiple lower types

is used to represent strings. The string value is converted into an unsigned integer representing the string-length and n-bit of unsigned integers representing the Unicode for each single character. A string table is implemented to avoid duplicating encoded strings and improve efficiency. Unique strings are stored in the string table, duplicated strings are instead assigned a compact identifier.

Simple types can be mapped to a lower form constructed by boolean and unsigned integers. The length required to encode an unsigned integer *m* in the range of 4096 or smaller is represented as an *n*-bit unsigned integer where *n* is equal to $\lceil(\log_2(m))\rceil$. For example, the integer 324 can be encoded into an *n*-bit unsigned integer where $n = \lceil(\log_2(234))\rceil = 9$. A similar encoding rule is applied to high level simple types which are encoded efficiently into an unsigned integer form. Enumeration is a powerful simple type which is encoded using the same equation. Enumerated values are encoded into an *n*-bit unsigned integer with *n* equal to $\lceil(\log_2(m))\rceil$ where m is the number of enumerated values listed in the schema. For example, given 4 enumerated values, the length of the unsigned integer in bits required to represent enumeration is equal to $\lceil(\log_2(4))\rceil = 2$. The unsigned integer will represent the order or the items listed in the schema, `00`, `01` and `10` will represent the first, second and third enumerated item respectively.

**Encoding Example**

Code listing below provides an example of a schema-informed encoding mechanisms using the XML and XSD files of Appending A code listing A.7 and A.8. XML data A.7 is listed below to provide visual clues during the analysis of the encoded format. XML data types are fully constrained using appropriate simple types provided in code listing A.8.

LISTING 3.3: XML data

```
<?xml version="1.0" encoding="UTF-8"?>
<student>
  <module>FuncProg</module>
  <hours>48</hours>
  <courses>CS</courses>
  <ref>AABBCCDDEE</ref>
</student>
```

XML data of code listing 3.3 is compressed using an EXI recommended library available in (Garrett, 2012). The schema-informed compression provides

the following format listed in hexadecimal form. In addition, an additional bit-analysis of the hexadecimal form is provided to demonstrate the encoding mechanism of EXI.

```
80 01 48 CE AD CC 6A 0E 4D EC E2 40 00 B5 57 79 9B BD C0
```

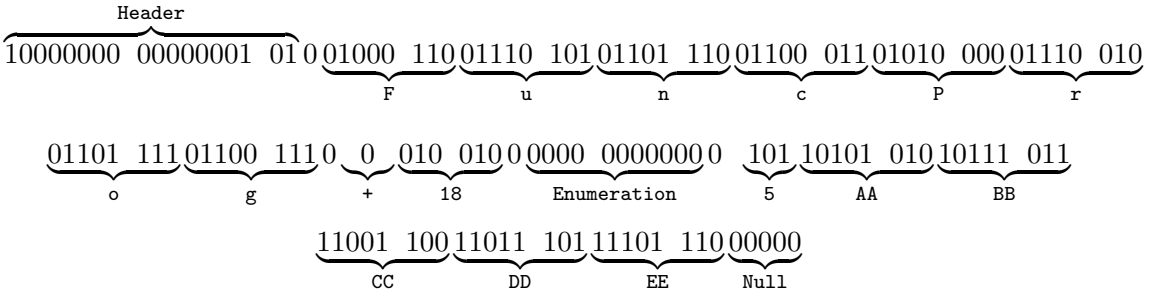


FIGURE 3.3: EXI binary representation analysis

From the analysis of the unaligned format presented in figure 3.3 it is possible to highlight some of the features described in the previous sections. Two octets are used to provide header information and to describe a `sequence` complex type. Each data type is separated by a single bit represented using `0`. String characters are encoded using their 8-bit binary form. Encoding mechanisms for enumeration are not consistent with the rules described in previous sections. This inconsistent encoding can be related with issues to the EXIficient EXI library. Hexadecimal strings are encoded using one nibble, a 4-bit unit of digital information, for each character instead of using an octet needed to represent standard strings.

**Compression**

EXI is able to achieve high compression rates by encoding event streams using a compression algorithm. As result of a sequential encoding technique, the event stream is a mixture of heterogeneous data. By leveraging knowledge of both the XML and the compression algorithm, EXI multiplexes the heterogeneous stream into channels which can then be compressed more efficiently. Similarly to the XMill idea of grouping homogeneous data into containers to be compressed individually, EXI groups the structure information as event-code and values of the XML into different channels. In addition, to achieve higher compression, XML values are grouped according to the element names.

Smaller channels are then combined with larger channels to keep a low overhead. For example channels with few different element names are combined into a single larger channel. This local homogeneous property of EXI is based on knowledge of the DEFLATE compression algorithm. By creating a byte-aligned representation of the event stream, the algorithm is more likely to identify redundancies in the octets compared to an unaligned format.

#### 3.1.6.3 Limitations

Based on the specifications provided in the EXI format 1.0 documentation, it is possible to highlight some of the limitations of EXI. Contrarily to research that considers EXI to be a light-weight approach to achieve higher compression (Castellani et al., 2011), full implementation lacks simplicity. Most of the XML aware compressors can be categorised as standard binary encoder such as ASN.1 and WBXML or encoder based on general-purpose algorithms with support for XML. To achieve compression sizes higher than existing tools, EXI can implement both binary encoding and general-purpose compression. This is achieved using a fixed and variable length encoder at different stages. This level of complexity is therefore not suitable for constrained devices with limited resource capabilities. A second misconception is the idea of using the binary format to avoid parsing time and resource consumption. This feature is only available for a schema-uninformed compression, which does not leverage knowledge of XML data types. If a schema-informed approach is used, the XSD data types and user-defined custom types have to be recognised. To achieving higher compression, the *strict* mode of EXI depends on the validity of these data types. This requires the XML to be validated against the XSD. This process intensive operation impacts the performance of the tool and its ability to perform in a constrained environment based on current specifications.

When an XML value is mapped to a specific type, the value is transformed into a lower format which is then converted into a sequence of octets. However, this feature is only available for schema-informed techniques and it is not supported for the uninformed approach. As an encoder, EXI is not able to detect XML data types without the knowledge provided by the XSD. Therefore, in absence of a schema language which specifically defines data types, all XML values are by

default mapped to their best matching data type, which, in most cases, does not provide the highest level of compression.

### 3.1.7 Abstract Syntax Notation One

Abstract Syntax Notation One (ASN.1) is a standard to describe a set of rules and structure for encoding/decoding, transmission and data representation (Steedman, 1993). Primarily used for telecommunication, ASN.1 has expanded in the field of computer networking as the basis of most current technologies. In network environments, data generated by one machine needs to be communicated to a single or a number of different machines. Depending on machine architecture, the data encoding mechanism of a particular machine might be different from the decoder. ASN.1 has the fundamental role of providing abstract syntax to consistently encode, transmit and decode data. Depending on the machine architecture, messages carried across the network are specified as binary values formed by a sequence of octets. Independently from their binary representation, the sequence of octets needs to be mapped to a number of data types. ASN.1 is the notation used to specify these data types using a set of algorithms named *encoding rules* that allow to determine the value of the octets. These encoding rules enable the use of data types between machines with different encoding techniques by automating the data type's validation process.

#### 3.1.7.1 Encoding Rules

Basic Encoding Rules (BER), Packed Encoding Rules (PER), and XML Encoding Rules (XER), are the three main families that determine the value of an octet (ITU-T, 2008a). Although ASN.1 provides the structure, it does not restrict the encoding mechanism used to generate the transfer syntax. BER, PER and XER, together to variations of these rules, provide the basis to encode data independently from machine architecture or language. Encoding rules of ASN.1 are designed to provide benefits to users based on different circumstances. For example, a subset of BER, Distinguished Encoding Rules (DER), provides a unique mechanism to encode values enabling the use of this encoding rule to digital certificates. PER is aimed at achieving more compact transfer syntax in order to tackle issues related to low-bandwidth networks. The main objective

of XER, instead, it to enable ASN.1 data to be displayed in a human-readable format and to process it using common XML tools such as parsers or browsers.

**PER**

The term *Packed* Encoding Rules, PER, is used to highlight the idea of achieving the minimum representation size for ASN.1 data. Compared to DER, this format is able to achieve a lower size by omitting type tags encoding. However, in order to decode the message, the decoder must be aware of the protocol used to encode the data. Basic and Canonical PER are the two encoding rules with bit aligned and unaligned variant (ITU-T, 2008c). Aligned PER encodes the length and the data of the value consistently as shown in figure 3.4.

```
02 22 04 4A 6F 68 6E
```

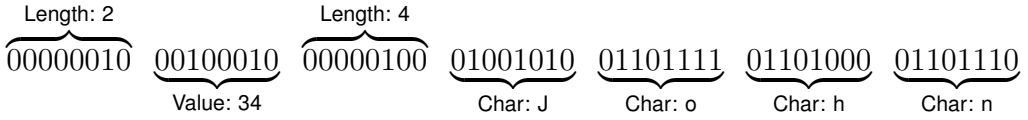


FIGURE 3.4: ASN.1 aligned PER binary representation

The process of encoding data from code listing A.10 of Appendix A using aligned PER is based on the direct byte addressing process similar to DER with the omission of the type tag value triplet. Unaligned PER manages to achieve a smaller transfer syntax by encoding data more efficiently as shown below.

```
02 22 04 95 BF 46 E0
```

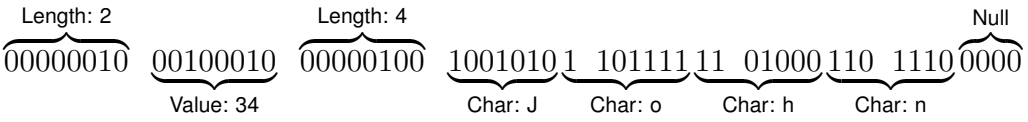


FIGURE 3.5: ASN.1 unaligned PER binary representation

The difference between aligned and unaligned PER is mainly shown in the `IA5String` type encoding. The encoder is aware that valid `IA5String` only requires 7 bits instead of 8 bits to be represented. By removing the most significant bit, the string octets are formed using the bits of the next encoded values.

In this unaligned example, the last octet is padded with 4 redundant bits to form a new octet $\mathrm{E0} \rightarrow 1110\mathbf{0000}$.

#### 3.1.7.2 Compression Comparison

Although ASN.1 is not a XML-conscious compressor, it is at the basis of many current technologies. Most of the well-known protocols and mechanisms to define data have adopted ASN.1 as the fundamental encoding mechanism. XML Schema also is heavily influenced by the idea of constructing simple and complex types based on the specifications previously described. The ability of XER to be mapped to XML schema languages has enabled the use of ASN.1 protocols to be adopted as a schema language. Therefore, ASN.1 schemas can be used to validate XML similar to the W3C schema languages.

The encoding size and performance of ASN.1 has been tested against EXI and Gzip (Bournez, 2009). EXI is able to achieve higher encoding rates due to the additional complexity and compression applied to the encoded stream. Since based on a similar encoding mechanism as PER, EXI shows similar encoding rates to ASN.1. Scenarios with a poor encoding are the result of schema language deviations and lack of simple data types. As based on similar encodings, PER can achieve the same encoding as EXI when compared to the schema-informed approach. Therefore, it is possible to assume that ASN.1 is able to achieve similar encoding sizes to EXI without the DEFLATE compression option enabled. One of the drawbacks of ASN.1 is the inability to recognise additional elements within the XML which are not specified in the protocol/schema language. Although EXI is able to encode a wider range of XML documents, the performance of EXI against Gzip or ASN.1 was not evaluated. A clear justification was provided based on the lack of native languages implementations such as C/C++ (White et al., 2006). The Java implementation of EXI cannot be compared to a native language due to the overhead associated with the Java Virtual Machine (JVM). The EXI working group has considered a native implementation comparison as future work. Therefore, based on the knowledge of both encoders, it is possible to estimate ASN.1 PER encoders to perform better than EXI. This is achieved using the ability of ASN.1 to be compiled into native code, increasing the integration with data structures of native languages.

### 3.1.8 Packedobjects

Packedobjects (PO) efficiently encodes XML data using the information provided in a corresponding XML Schema (Moore, 2012). The schema-informed approach used by PO is based on the idea of mapping XML data values to a corresponding protocol to achieve higher encoding rates. Using a set of built-in data types, PO is able to compress data efficiently by applying encoding rules extended from ASN.1. As schema-informed compressor, PO is only capable of performing as an offline compressor. XML data is encoded using a homomorphic compression technique. The decompressed data can only be processed sequentially using the information provided in the schema. The following sections highlight the key feature of PO, its ability to successfully operate in constrained networks, the architecture and limitations of the tool.

#### 3.1.8.1 Design principles

The main objective of PO is to achieve the highest encoding rates for structured data across the network. The justification for using XML in contrast to other formats is given by a wide range of network applications that can support this language. The use of an XML Schema language instead of a DTD is needed to provide additional data type in formation.

**Efficiency**
PO is defined as a data encoding/serialisation tool that provides high-level bit-packing to support network applications. Hence, the use of PO can be justified to a specific application of XML compression, low-bandwidth networks. The efficient encoding size is the first key feature presented by PO. As a format based on ASN.1 PER, PO is able to support data types that can be easily mapped to data structures of native languages using the minimum amount of bits.

**Extensibility**
The use of an abstract syntax has been subject to various research over the last decades to overcome issues related to byte ordering between heterogeneous platforms. The extensibility feature of PO allows protocols based on ASN.1 PER to be extended by integrating more high-level data types such as Date and

IPv4. PO offers the ability to map high-level simple data types to lower formats preserving the original transfer syntax.

**Integrability**
The application of PO is found in a number projects, ranging from devices running over Internet of Things (IoT) to more common networks based on publisher-subscriber models (Moore et al., 2013a, 2012, 2010). Heterogeneous systems are based on devices with different architectures and processing capabilities. PO is able to run across different platforms supporting the same binary protocol for machines based on *MIPS* and *ARM* instruction set architectures (ISA) up to more complex CPUs running on *x86*. This allows network topologies based on a centralised unit to exchange transfer syntax between sub tree or mesh networks running on different hardware.

#### 3.1.8.2 Architecture

Domain knowledge allows users to deal with data in a different perspective. Traditional general-purpose text compressors are not able to recognise the structure or the data type values of an XML file. For example, an hexadecimal string is processed as a sequence of characters and encoded using a specific compression algorithm. XML-conscious compressors are able to recognise the structure of XML and, in some cases, compress data values using semantically-aware techniques. However, knowledge of data types provided by the schema language, allows the compressor to encode an hexadecimal character as a nibble (sequence of four bits) instead of an octet. Domain knowledge allows PO to outperform general-purpose and less semantically-aware compressors mainly in the field of networking. Data types processed by a network management application are usually repetitive and can be easily mapped to a schema language for validation purposes. These applications exchange information such as IP addresses, CPU temperatures, sensor data and vendor specific information.

The schema language plays an important role in helping PO to identify data types of XML. Using the knowledge provided by data types, PO is able to recognise each value and apply the most efficient encoding technique. Similar to EXI, PO transforms high-level data formats into a lower format using signed and unsigned integers. For example, *bit-string* is based on the *xs:string* restriction, however, each character can be either `0` or `1` and therefore encoded using 1 bit

instead of 8 bits. The *unix-timestamp* data type is used to represent date/time using the RFC 3339 format such as `2014-09-12T04:38:36Z`. This format can be converted to POSIX time which is represented as the signed integer `1410493116` and encoded by PO in merely 4 octets.

A complex type is the term used to identify a set of data types as they appear in the XML. As an abstract syntax extended from ASN.1, PO provides a rich syntax to describe network protocols. Complex types of ASN.1 are at the basis of current protocols and have inspired new languages such as XML Schema. However, to enable the use of this language to a wider range of applications, these complex types have been extended to allow accurate and general sets of data types. For example, XML Schema allows indicators such as “sequence”, “choice” and “all” enabling XML data to be presented freely. While this feature allows the use of valid XML for a wider range of applications, schema-informed encoders are required to store additional data in order to keep track of the value in an unordered sequence of data types. Schema-informed compressors such as EXI and PO avoid encoding the structure of XML based on the knowledge provided by the schema. However, if a schema does not present an accurate complex type representation, it is not possible to directly map *n*-bits to a specific data type. While supporting unconstrained complex types allows the compressor to be extended to different fields, this feature increases software complexity and worsen the compactness of the transfer syntax. As a tool aimed to improve network efficiency, PO does not integrate the use of complex types derived from ASN.1 that require additional processing. The lack of support for these complex types allows PO to be implemented easily, to perform faster and to keep the transfer syntax to its minimum size.

Figure 3.6 summarises the design of PO during the compression and decompression stages. An initialisation function is used to validate and transform XML schema into an internal schema format for efficient compression and decompression of XML data. The internal schema can be called upon multiple times avoiding the parsing overhead. Furthermore, this internal structure can be cached to speed up program start-up. *Libxml2*[2] is the library used to parse the XML DOM and the schema language. During the `encode()` function the XML DOM is validated against the memory representation of the schema. XPath is used to map XML values to the schema which are subsequently encoded using

[2]The XML C parser and toolkit of Gnome - http://www.xmlsoft.org

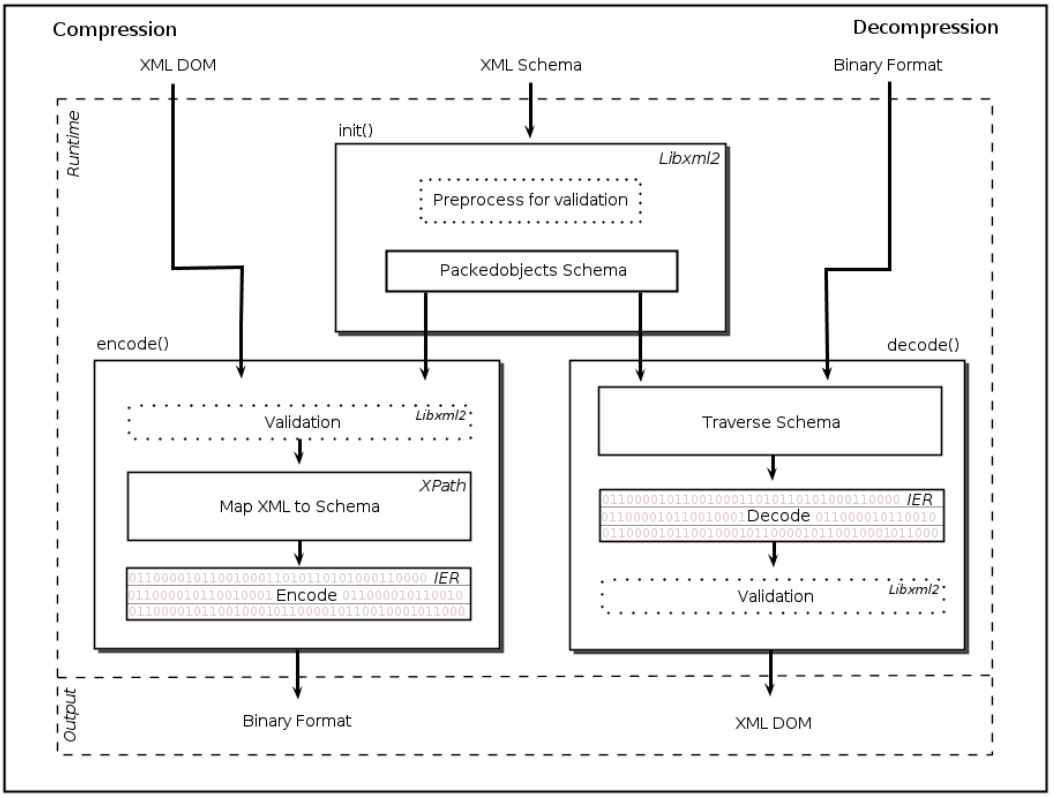


FIGURE 3.6: Packedobjects Architecture

the PO Integer Encoding Rules (IER). Depending on the application, the output of the encoder can be stored on disk or allocated to a memory buffer. The `decode()` function requires the schema used in compression to call the correct decoding routines. After traversing the schema, IER maps bits to their values creating a memory representation of XML which is subsequently validated to check for possible errors. The XML DOM can then be stored locally or passed to a front-end application.

#### 3.1.8.3 Integer Encoding Rules

PO can be divided into two main components, front-end and back-end. The front-end part of the application is designed to convert syntax notation into a low-level format which is then subsequently passed to the back-end of the system. This last part is based on Integer Encoding Rules IER, a variant of ASN.1 PER to encode and decode data. The back-end provides a compact transfer syntax while operating efficiently thanks to its native C implementation. Similar to the unaligned PER transfer syntax, the result of the encoding is a sequence

of octets representing the length and value of each data type (Moore, 2009, 2010a, 2011, 2010b). As analysed in the previous chapter, the use of PER is specifically designed to create a compact transfer syntax by relying on the knowledge of the ASN.1 protocol. For this reason PER is preferred over BER and DER and other encoding rules variants.

The design philosophy of PER is to encode a value using the minimum amount of bits required to represent the specified range. Provided lower and upper bounds, the bits required to encode an integer can be calculated using $\lfloor(\log_2(n))+1\rfloor$, where *n* is defined as the difference between the upper and lower bound. Given *a* and *b* to be lower and upper bound respectively, where *n* is equal to *b* - *a*, the amount of bits required to encode a value in the range of *a* and *b* inclusive, is equal to $\lfloor(\log_2(n))+1\rfloor$ . For example, for *a*=100 and *b*=500 (upper and lower bound), *n* = 400, $\lfloor(\log_2(400))+1\rfloor$ = *9*. Using this equation, IER maps data type's values to a sequence of signed and unsigned integers. The range provided by the protocol is essential to provide additional information to the encoder. Upper and lower bounds provides enough information to encode the constrained integer without providing the length information. Semi-constrained and unconstrained integers, instead, require the length of the value to be encoded, decreasing the compactness of the transfer syntax.
The front-end of PO is designed to transform high-level syntax into a lower form, where data types are presented using a more suitable form to be passed to the encoder. The transformation sequence can be broken down into various stages using XML and XSD examples provided in code listing A.7 and A.8 of Appendix A. Using s-expressions it is possible to represent data and protocol in a concise format which can then be easily mapped to the IER. The first stage is to create a combined form by merging information from both data and protocol.

LISTING 3.4: PO Normal form

```
;; Normal form
((student sequence)
 (module string (size 0 10) "FuncProg")
 (hours integer (range 30 60) 48)
 (courses enumerated 0 2)
 (ref hex-string (size 1 64) "AABBCCDDEE"))
```

XML data is *assisted* by the constrains provided by the XSD protocol. This example provides four simple types: `string`, `integer`, `enumerated`, `hex-string`

and one complex type `sequence`. This complex type does not affect how information is encoded, therefore, it does not increase the size of the compressed format. The second and third stage of the encoding process are described in code listing A.11 of Appendix A.

The output of the core form is an unaligned sequence of octets analysed below.

```
136 221 119 99 161 203 126 121 4 213 93 230 110 247 0
```

The first sequence is the decimal representation of the transfer syntax using decimal values which maps to a 8-bit binary. The value of each group of bits is shown in the bit analysis listed in figure 3.7. The over brace and under brace are used to specify the length and value of data types.

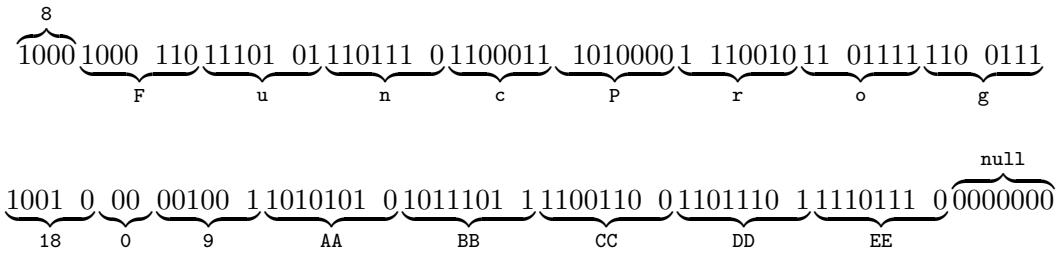


FIGURE 3.7: PO binary representation

#### 3.1.8.4 Applications and Limitations

The following section describes the application and limitations of PO for its use as low-level encoder. This information will be used to construct an alternative model capable of addressing the limitations highlighted in this research area.

**Applications**

Packedobjects is a library which can be used to efficiently compress an XML DOM by using the information provided by a corresponding XML Schema. The level of compression achieved is very similar to EXI but unlike EXI, Packedobjects is designed to be light-weight and simple to implement. Therefore it is suited for embedded systems and mobile devices. The software has been designed for writing network protocols which strive to minimise the amount of data communicated. In addition to compression, all data is validated by the schema

during the encode and decode process. Packedobjects is not a general purpose document compression tool. It is intended to be used in an application that communicates over a network. As such it provides a simple DOM-based API for encoding and decoding structured data. Similar to EXI, the compression technique used is based on applying knowledge of the data types specified in a schema to provide better performance over statistical compression techniques.

**Binary Logarithms**

The schema-informed compression of PO and EXI can be described as fixed length encoding. Here, the probability of each bit of information is equally likely to appear. Both EXI and PO are implemented on similar mechanisms based on binary logarithms $\log_2(n)$. This function is used in order to calculate the number of bits required to store different data types. For example the result of the binary logarithm for the unsigned integer 4, means that it can be stored in 2 bits, 8 into 3 bits, and 32 into 5 bits. Binary logarithm is commonly used in the field of data compression due to its connection to the binary system. EXI and PO highly rely on this system to encode data types efficiently and calculate the required number of bits needed to store a value. Sections 3.1.6 and 3.1.8 have described the used of $\lfloor(\log_2(n))+1\rfloor$ and $\lceil(\log_2(n))\rceil$ for PO and EXI respectively. The standard approach to obtain the number of bits required to encode a value is calculated by adding 1 to the integral part of the binary logarithm. The integral part is obtained using the function *floor* $\lfloor x\rfloor$ to return the largest previous integer not greater than $x$. This binary logarithm function is able to specify an encoding mechanism for values of length equal to 1. On the other size, EXI calculate the length of the value using mathematical function *ceiling* $\lceil x\rceil$ to return the smallest following integer not less than $x$. Using function $\lceil(\log_2(n))\rceil$, where n is 1 would return 0 instead of 1 and 32 would return 5 instead of 6. Therefore, additional complexity needs to be included in the EXI implementation in order to handle situations where the binary logarithm does not return the correct length.

**String Support**

Based on encoding rules optimised for network transmission, the performance of PO for string value compression is poor compared to string-optimised tools. Compressing strings requires multiple encoder and decoder calls resulting in lack of performance and representation of the compressed format. Using the unaligned variant, PO is able to reduce 1 bit for every 8-bit characters represented in the first 127 ASCII printable characters. The 7-bit encoded string

representation does not provide a sufficient compression compared to other semantic-aware and statistical tools. This lack of support for string compression is justified by the use of PO in the field of network transmissions.

**Additional Processing**

The second version of PO is based on XML and XML Schema to define data and protocols. One of the advantages of PO over most of the XML-aware compressors is the use of XML Schema to behave as a protocol. However, the advantages of this language are minimal compared to the drawback and the additional complexity introduced by XSD documents. Due to the lack of namespaces and other XML components compression, it would be possible to design a PO protocol using a DTD. Similarly to XSD, the DTD can validate multiple XML documents as an external entity. The cost of using XML Schema is justified by the ability of mapping ASN.1 protocols to XML, essential for the IER encoder of PO. The major drawbacks of XSD protocols are the computational resources required and the additional time lost for handling an additional file. A speed-up mechanism is provided by the PO API which allows the schema file to be parsed on software start-up and cached in memory. This feature avoids the need of parsing the XSD on each encode/decode call, reducing the processing resources and time needed to compress data.

### 3.1.9 Other Compressors

A number of additional XML compressors have been developed over the last few years. This section summarises the results and the major contribution for each of the following compressors discussed in the literature reviews.

**Fast Infoset**

Other binary XML formats exist such as Fast Infoset (FI) which specifies how to encode XML into binary using Abstract Syntax Notation One (ASN.1) (Sandoz et al., 2004; Steedman, 1993). Differently from general text compression tools, FI aims to improve processing time and performance. FI provides an alternative syntax to represent instances of the XML information set discussed in the previous chapter. Using a binary encoding mechanism, this implementation is usually faster and provides a smaller syntax compared to XML. In order to increase the encoding size, FI utilises vocabulary tables (similar to EXI) to map

elements, attribute names and other character strings to small integer values. The tool is also supported by the use of additional algorithms to efficiently encode integers, floating points and arrays of these numbers. For example, an integer in the range –32768 to 32767 is recognised by the tool and encoded using two octets representing a signed integer instead of five octets representing the string. FI is based on Encoding Control Notation (ECN) of ASN.1. However, this rule is not necessary when the encoding mechanism is provided (ITU-T, 2005). As a binary format based on ASN.1 encoding rules, FI can be compared to EXI which provides similar encoding mechanisms. However, due to the additional compression and complexity introduced by EXI, FI is not able to achieve similar encoding sizes (Jaiswal and Mishra, 2013).

**XGrind**

XGrind is another approach of compressing XML using properties designed by XMill (Tolani and Haritsa, 2002). This tools aims at increasing the compression size by using information provided by a DTD file, for example enumerated attributes are encoded more efficiently. The XGrind format supports queries to the compressed format which is represented in a semi-structured query-friendly format. Results demonstrate how XGrind is able to achieve similar compression sizes to XMill, although the main advantage of XGrind is the ability to perform queries on the semi-compressed format.

**XPress**

XPress is an efficient XML compression tool optimised to perform direct and efficient queries to the compressed format (Min et al., 2003). Similar to XGrind, the homomorphic properties of XPress preserve the structure of XML documents. The novelty of XPress is the introduction of a specific algorithms, Reverse Arithmetic Encoding (REA) to encode label path as distinct intervals. One of the advantages of XPress is the ability to apply different encoding algorithms to diverse data types. Results demonstrate a better compression ration over XMill and XGrind with negative results in compression speed when compared to XMill and gzip.

**XComp**

XComp is a compression tool based on the local homogeneity properties of XMill: separating content from structure and grouping data values into semantic

containers (Li, 2003). The novelty introduced by XComp is the ability to confine the maximum memory usage for each of the containers. In addition, data is not only grouped based on elements and attribute names but also based on the document levels. The back-end compressor used to compress each container is uniquely based on *zlib*. In conclusion, XComp can be categorised as an extension of the work based on XMill that provides similar performance and a better encoding size.

**XAust**

XML Compression with Automata and Stack (XAust) is a Schema-informed compression tool based on arithmetic coding (Subramanian and Shankar, 2006). Information provided by the DTD is transformed into a Deterministic Finite Automata (DFA) and used to track the structure of the XML with accurate prediction on the expected symbol. Data values for each simple type are encoded into specific containers which are incrementally compressed using arithmetic coding. XAust results demonstrate small benefits in compression ratio over tools such as XMLPPM and *bzip2*. The Java implementation of XAust is compared to other tools developed using native languages such as C++ of XMLPPM with some benefits in virtual memory used and time required to compress XML.

**RNgzip**

Rngzip is a schema-informed XML compression tool based on gzip (League and Eng, 2007a). Using a more clean and light-weight XML Schema language, called Relax NG, this tool is one of the first to implement a more advance schema. In comparison to other compression tools such as bzip2, XMill, XMLPPM, DTDPPM and XAust, rngzip does not always provide the best compression size depending on the nature and size of the XML. In addition, because of the implementation written in Java, rngzip does not provide run-time performance metrics in comparison to other tools.

**Millau**

Millau is another Schema-informed XML compression tool to efficiently represent and exchange XML data over the Web (Girardot and Sundaresan, 2000). The encoding process of Millau does not strictly depend on the schema. A DTD can be provided to optimise a token dictionary providing better performance and compression size. This tool can be described as an extension on WBXML,

which by applying additional knowledge from a schema is able to achieve higher compression sizes. Millau is one of the few tools designed for the specific purpose of managing XML data over the web. However, although this tool focuses on a specific domain, it does not restrict the data set to a specific domain. Results demonstrate the performance of Millau when presented with XML of different sizes. Millau manages to achieve smaller compression sizes for files below 5KB. Over this range, gzip takes advantage of the redundancy of the data to achieve a better compression.

**Protocol Buffers**

Protocol Buffers (PB) is an high-level language syntax used to represent structured data concisely (Varda, 2011). Using a light-weight syntax with a concise structure, data is mapped to a protocol which provides information about data types and structure of the document. The advantage of PB is the use of the Interface Description Language (IDL) used to describe both data and protocols. Compared to XML, PB syntax is simpler to write and provides a more clear format similar to ASN.1. This binary tool is able to encode data 20 to 100 times faster compared to XML for serialising structured data. PB is included between the tools studied because of its similarity with PO and its resemblance with ASN.1. With knowledge of PO it is possible to describe PB as a binary data serialisation tool with similar encoding rules but different syntax.

### 3.1.10 Summary of Related Works

Section 3.1 reviewed a number of XML compressors and binary tools to efficiently represent structured data. This study focused on compressors which have been considered in a number of research papers and have achieved significant results in comparison to other novel approaches. Most of the tools studied are developed using native language implementations. This measurement was considered based on the ability to benchmark multiple tools without considering the language implications. Although it is believed that an application written in Java can achieve similar results to a native implementation, due to the improvements of virtual machines, the residual systematic effect of a JVM cannot be neglected. This includes byte code interpretation and the overhead tasks required to manage memory. Memory management and garbage collection are

important features for native applications which allow tools to have more control over the exact amount of memory that can be allocated and freed at different times.

#### 3.1.10.1 Tools Categorisation

An important component that needs to be considered when comparing the processing efficiency of different tools is the library needed to transform XML into a lower form which can be then evaluated by the tool. First of all, the choice between tree-based and event-based API depends on the ability of each tool to operate on a stream basis or using an internal memory representation. Based on its domain a tools can justify the need for a DOM over a SAX API which can be fundamental for the functionality of the system. This API is provided by an XML parser library which is used by most schema-informed compressors to map data type constraints to element values. Multiple XML parsers exist and have been implemented in different languages. The efficiency of a tool also depends on the ability of an XML parser to manage XML data efficiently. Binary tools such as EXI and FI do not require a XML parser and therefore should theoretically achieve faster encoding speeds. Processing efficiency and memory consumption should not be evaluated for non-isomorphic applications which have been designed to operated in different domains. Based on these properties, it is possible to compare different tools on their software complexity and the ability to operate in constrained devices.

| Symbol | Description |
|---|---|
| P | General-purpose Compressor |
| C | XML-conscious Compressor |
| I | Online XML Compressor |
| O | Offline XML Compressor |
| N | Schema-informed Compressor |
| U | Schema-uninformed Compressor |
| Q | Queriable XML Compressor |
| A | Archival XML Compressor |
| H | Homomorphic XML Compressor |
| L | Homogeneous XML Compressor |

TABLE 3.1: Features and Classification of XML Compressors

Table 3.1 provides a number of symbols for different properties which can be found in XML compressors and serialisation tools. These features are categorised based on the following properties:

- General-purpose or XML-aware compressors
- Tools that provides a streaming or tree-based API
- Ability to take advantage of a schema language
- Tools designed for archival or queriable formats
- Local homogeneity and homomorphic compressors

Table 3.2 lists the compressors that have been evaluated in section 3.1 using the feature and classifications described in table 3.1. A link to the online repository is provided for each of the compressors that have been evaluated and tested for the scope of this research. This study evaluated compressors which are developed using mainly a native language such as C and C++. PO is available in two different versions, an abstract syntax in XML or s-expressions using Guile (an implementation of the Scheme programming language). The back-end low-level encoder for both version is written in C. Although developed in a different language, EXI was included in the evaluation as the standard recommended by the W3C.

| Compressor | Features and Classification | Code Availability | Language Implementation | Software Complexity |
|---|---|---|---|---|
| XMLPPM | CIUAH | (Cheney, 2006c) | C | **Medium** |
| DTDPPM | CINAH | (Cheney, 2006a) | C++ | **High** |
| WBXML | CONAH | (Jehanne, 2009) | C | **Medium** |
| XMILL | COUAL | (Colver, 2004) | C++ | **High** |
| ZLIB | POUAH | (Adler, 2005) | C | **Low** |
| Packedobjects | CONAH | (Moore, 2012) | C / C - Guile | **Low** |
| EXI | CONAL | (Garrett, 2012) | Java | **High** |

TABLE 3.2: List of XML Compressors

The software complexity column is based on an analysis of each compressor on various factors. This information was grouped based on the ability to provide developer-friendly API which enables the use of the tool to be imported in other platforms. The control flow was examined together with the use of operation such as IF, DO, and SELECT statements. Software maintainability is also considered based on how feasible it is to extend the front-end part of the

system. Finally, the code structure, design and function refactoring was considered. From the table it is possible to identify a number of compressors with a high level of complexity. For example, DTDPPM is a beta version based on top of XMLPPM, mainly experimental and designed for academic purposes only. EXI complexity is both accidental and essential. Although the selected library provides a good API, the Java implementation cannot be easily ported to low-powered devices. In addition, the essential complexity of EXI design yields a larger amount of complexity compared to other tools.

#### 3.1.10.2 Limitations

XML compressors described in section 3.1 have been tested in a number of research papers presenting similar results. These compressors have been developed to solve issues related to at least one of the areas where XML is heavily used: XML Document storage and transmission, binary format for XML messages processing and transmission, and XML database storage and query processing. EXI is the only implementation that has been designed with the intent to formalise a general-purpose XML-conscious compression. However, the results provided in the experiments are scattered and inconsistent (Augeri et al., 2007; Sakr, 2009). For example, tools designed for a specific purpose are tested and evaluated using XML documents that do not belong to that specific domain. This issue is also related to the lack of a formal XML corpus which has been so far ignored. A number of few well-known XML documents have been used to test new tools. In addition, XML compressors listed in section 3.1 are evaluated with large data sets with exception for specific tools such as PO and DTDPPM.

EXI defines an XML compression tool using a schema-informed approach applying additional compression to the aligned stream. Few such implementations capable of running in different domains exist. A good compression ratio can be achieved only when all the options of EXI such as schema-informed, bit alignment and compression, are enabled. In practice, EXI is able to achieve a higher compression ratio by using multiple compression schemes. From the following observation it is possible to conclude that a general XML compressor is directly proportional to the complexity of the software which has a negative impact on the system resources. For example, the highest compression achievable by EXI

requires the use of an XML parser to analyse the XML Schema patterns and facets and return the restricted character set, a pre-compression event grouping, and a DEFLATE compression applied to the aligned stream.

A good compression scheme needs to be implemented based on the XML data presented and the fields where the compression is applied. It is important to understand why certain tools fail to achieve a reasonable compression ratio when presented with different XML data. A statistical analysis of XML is necessary to understand the requirements of a compression tool depending on the domain. For example, a network management domain would share XML messages based on sensor data and timestamps and vendor specific information. An XML document used in database compression, instead, is less structured and contains higher amounts of string data types. Some XML compression research outputs have presented studies on the nature of XML. However, these studies only concern the structure of XML such as the amount of element tags, nesting length and amount of data values. It is important to consider also simple and complex types for XML data, in order to understand which compression algorithms can be applied over another. XMill introduced the first approach using semantic knowledge to increase compression. Raw data converted into XML increases the size by doubling the length of a file but demonstrates a better compression using user's knowledge of XML data types. Schema-informed compression manages to achieve similar results using the knowledge provided by the Schema language. EXI and PO are able to exploit this principle and convert XML into a binary formats using ASN.1-like approaches.

#### 3.1.10.3 Revisiting Research Goals

Section 3.1 analysed and discussed a number of compressors and back-end algorithms to efficiently manage XML data. This section categorised each compressor based on its features and software complexity. Many XML compressors are able to achieve higher levels of compression compared to other tools by increasing the level of complexity. The additional level of compression achieved using a schema language allows schema-informed compressors to be more efficient compared to standard XML-conscious techniques. The information provided in the schema, allows compressors based on variable and fixed length

encoders to achieve higher levels of compression via different techniques. However, the use of this data definition language is shown to be effective only when both encoder and decoder share the schema file, for example when exchanging data between homogeneous networks.
Each tool analysed in this chapter evaluates the performance of their compression techniques against specific XML data sets. Tools based on general-purpose compressors have shown to be highly effective for large XML data sets. However, little work has been done to evaluate their performance when presented with small highly-structured XML files, typically found in network environments. More experiments need to be conducted to evaluate the performance of these tools against a wider range of XML files.

From the analysis of XML compressors discussed in this chapter, this study concludes that only a few tools apply fixed length encoders, both presenting significant results compared to variable length encoders. However, these tools are mainly restricted to the use of this technique only to map basic data types. This is achieved using a data definition language, in case of Packedobjects tool, or using a list built-in data types for EXI.
The Packedobjects tool is a fixed length encoder designed to exchange highly-structured XML messages based on low-level data types such as Integers, Enumeration and IP data. This tool is not able to efficiently manage data such as strings due to the poor performance of its IER encoder for this specific data type. Contrary, EXI presents a more advanced approach to manage string data types using a variable length encoder and a fixed length encoder for basic data types. This binary representation together with a standard DEFLATE algorithm compression applied over the aligned bit stream, allows EXI to challenge most XML-conscious and general-purpose compressors.
The aim of this research is to investigate the use of fixed length encoders to compress XML data. The work presented by these tools can be considered as an initial direction to address the research question raised in Chapter 1. The amount of data types managed by the fixed length encoder is essential to understand the performance of this technique to compress markup languages. Additional levels of compression can be achieve by encoding data types which are usually passed to a variable length encoder. Here, the validity of a specific data type can be ignored as long as it can be mapped to a built-in data type. Compressing these data types using a fixed length encoder can increase the efficiency of the compressor to a wider set of XML data.

## 3.2 Analysis of XML Data

For decades researchers have explored and improved the field of lossless data compression. Sophisticated tools are able to reduce the redundancy of files by applying compression algorithms based on dictionary and probabilistic models. These results have been achieved thanks to algorithms capable of exploiting the knowledge of textual files. For example, dictionary coders match text with a set of data strings contained in their dictionary table. Variable length coders are able to increase compression by mapping most recurring source symbols with the least numbers of bits. These encoders are able to achieve optimal compression rates using some of the knowledge obtained by the textual file. Additional compression can be achieved with a better knowledge of the source. However, it would require almost infinite information to achieve the lowest compression size.

XML compression is subject to the same debate. Due to the simplicity of the XML structure, it is possible create a finite dictionary table to store XML components and then treat XML elements as pure text. Most XML compression tools have tried to improve compression by focusing uniquely on the structure of XML without considering XML data values. Recently, with the introduction of validation languages, which are able of describe the structure and data values of XML, there is the possibility to consider the data types and exploit the information provided within. XMill was the first tool to demonstrate how compressing textual data can benefit from an XML structure by applying knowledge of data types. General-purpose compressors are not able to perform well when confronted with random machine generated data such as IP, MAC address and integers. This issue can be solved using knowledge of these data types and treating data in a different perspective. The issue of handling XML structure concisely has been solved using XML Schema languages together with encoding rules similar those derived from ASN.1. Structure of the XML is not included in the encoded format and length of the value can be omitted when fully constrained values are provided.

In conclusion, knowledge of the structure and data of XML is essential to increase compression and improve current tools. This section provides a summary of the related work that has been conducted in this field. Furthermore, this section discusses the issue related to the experiments of various XML compressors and presents an analysis of XML data sets.

### 3.2.1 Analysis and Current Results

Compression tools analysed in section 3.1 do not present an accurate analysis of the XML data sets used to test compression ratios. The performance of these tools depends highly on the XML document and should not be restricted to the size and the number of tags. More information needs to be provided on the data types which are contained in the XML elements. Structure and data types of XML are both important components that need to be considered to improve compression. Encoders based on extensions of ASN.1 PER, are known to be efficient in representing basic types such as `bit-string`, `boolean` and `integer` but less efficient with string types such as `IA5String` or `UTF8String`. Conversely, dictionary compressors are known to perform well with string types and lack efficient support for integers. Therefore, there are compression size issues when XML data contains string types. Knowledge of the XML allows users to predict the efficiency of a specific compressor and operate accordingly. It is important to understand the nature of XML data not just using the structure and size but also considering element data types.

The following sections describes the results of XML studies in the field of XML Compression, Data Management and Data Quality research areas. Two themes can be identified from the literature. Studies of XML files and studies on XML schema languages. The first is extended to include studies of XHTML and other markup languages of the Web. Various XML schema languages are also discussed.

#### 3.2.1.1 XML Corpora

A number of research has analysed XML data sets used for compression comparisons results (Augeri et al., 2007; Delpratt, 2009; Liefke and Suciu, 2000; Sakr, 2009). Most data sets include synthetic XML data created using XML generators. XML corpora have been analysed based on size, number of nodes, depth and data ratio, using popular XML documents such as `lineitem.xml`, `swissprot.xml`, `nasa.xml`. These files are available from public repositories (Grijzenhout, 2010; Miklau, 2014). These repositories provide a simple analysis on the number of components and structure of XML documents and are included in a number of studies for XML compression.

**Composition**

Researchers performed a statistical analysis of real XML data collected using automatic crawling and manually from government sites, document repositories and XML exports (Mlynkova et al., 2006). Results from the analysis of the 16'000 XML files collected demonstrate that the average size of a file is relatively low, only 1.3MB, for files ranging from 60 bytes to 1'971 megabytes. Small XML files were found for most categories of XML data. Document and report files are the only categories to show larger sizes for XML files. A number of statistical analysis are performed on both XML files and XML Schemas. Relevant to this research are the number of XML components found on the entire data set. For all XML categories, the most common components are elements and attributes. Empty elements are mainly dominant for XML categories such as Semantic Web and Reports. Furthermore, the percentage of text found in the XML data set demonstrates that tagging dominates the size of documents with exception of document file category, where the portion of text is around 80%. Overall, contrary to previous analysis in this research field, this study demonstrates that real XML data shows different pattern usages and these are not always complex as expected.

A second study used a total 200'000 XML files collected using an advanced crawler named Xyleme (Barbosa et al., 2005). This tool was able to discard replicas of documents previously collected using fingerprinting techniques. Various types of XML files were collected from the Web and categorised based on files extensions. This work distinguished between documents from the semantic web, with `.rdf` and `.rss` extensions, wireless application protocols `.wml`, form-accessible documents, and indistinguishable XML files `.xml`. The distribution highlights the high percentage of `.xml`,`.wml` and form-accessible files. In line with previous research, the distribution on the average document size for the entire data set is relatively low, 4,641 bytes in size. XML documents with size greater than 10KB only compose 17% of the data set. An analysis was performed on the content of this data set to identify structural and textual content. It was found that structural information is in fact dominant over textual content. Documents with highest percentage of textual information are part of the 17% of the data set with size greater than 10KB.

**Validity**

XML data has been studied in the field of data quality to assess its ability to be used by XML technologies (Grijzenhout and Marx, 2013). The results of these

research provide useful information on the well-formedness and validity of XML data collected from the Web. Experiments conducted in this fields allow research in XML compressors to be aware of the validity of XML documents and increase support for common problems and errors. A recent study (Grijzenhout and Marx, 2013) collected a total of 180'000 unique XML documents, including files with references to schema languages such as DTD, XML Schema and Relax NG. Well-formed documents compose 85% of the entire data set. A total of 24% reference to a downloadable DTD or XSD file, however, only 16% of these are well-formed documents. For XML Schema validation, it was found that a higher portion of well-formed XML documents could not validate with a syntactically correct XSD. Well-formed documents that validate with their schema composed only 10% of the total data set.
An additional analysis was performed to highlights issues related to character encoding, well-formedness and validity. While character encoding was correct for over 99% of the data set, it was found that a considerable portion of the collection had at least one error related to the well-formedness of the XML. These errors were found using a modified version of the *libxml2* XML parser. Around 15% of the collection had one to multiple fatal errors such as opening and ending tag mismatch, premature end of data in tag, and attributes construct error. Further analysis was performed to identify between recoverable errors and warnings. In regards to the validity of XML documents, it was found that files referencing an XSD are more reliable that those referencing a DTD. For those files which fails to validate, the most common errors are related to unexpected elements. In conclusion, this study demonstrates a positive change in the validity of XML documents with a growing number of files referencing an XSD.

From the results of these analyses it is possible to provide the following additional observations: many XML documents contain a large amount of text nodes which have a relatively low average length; element names are very repetitive and most of them contain large attribute nodes.

**Data types**

Research in the field of XML compression, data management and data quality presents interesting results regarding the composition of a variety of XML files. These studies provide a detailed analysis of the components and structure of XML documents, with a clear difference between structural and textual content

(Barbosa et al., 2005; Mlynkova et al., 2006). However, these works do not analyse XML data types. So far, XML data has only been analysed based on the amount of structural and textual information. A further analysis on the composition of the latter can provide useful information on the types of data found in real XML. This can be used to improve tools such as XML compressors to focus on the particular data types found in their respective domain.

#### 3.2.1.2 Schema languages

Other research have focused uniquely on the statistics of XML and XML Schema languages and the differences in document nodes, attributes and other components (Bex et al., 2004; McDowell et al., 2004; Mlynkova et al., 2006). Studies of the schema languages inform us on the amount of restriction which are applied on simple types (73% of the schema evaluated) and the data types used to represent data values. Information on the data types can be extracted either from the schema language or from the XML file. However, these studies do not provide an analysis on the composition of simple and complex data types. Further analysis on the data set shows that only 7.4% of XML do not have a schema language, DTD is found in 74.6% of the data set while XSD in the remaining 38.2%. However, it was found that based on previous research, the number of XML files with reference to XSD is gradually increasing (Grijzenhout and Marx, 2013).

## 3.3 Conclusions

This chapter discussed and evaluated various XML compressors and back-end technologies. Most of the compressors do not benefit from the schema language which is provided for many document-centric and data-centric XML documents. However, higher software complexity is generally required for compressors based on DTD or XSD schema languages. All the compressors analysed are based on one to multiple layers of general-purpose compression algorithms with the exception of binary XML representation. The key feature of each compressor is based on the management of XML data prior to applying

general-purpose compression or specific encoding rules. Compression size results of each compressor vary depending on XML documents. XML-conscious techniques can be applied to most documents which belong to the regular and irregular XML category.

The second section of this chapter focused on the statistics of XML data. This section analysed XML and XML Schema languages evaluated in XML Compression and XML Data Management research areas. Based on the knowledge of data it is possible to deduce the optimal compressor for different types of XML. Using a schema-informed approach compression can be improved mainly for structured data using special data types which can be mapped to a lower form. This approach is also important to reduce the redundancy of non-unique XML tags which occur in both data-centric and document-centric XML files. Results on XML data sets inform us that the number of schema languages linked to XML files is dominant in most cases. This information can be exploited to achieve better compression ratios. However, the analysis of XML compressors demonstrates an exponential increase in complexity when using a schema informed approach. Binary representations, known for their fast encoding mechanisms, are increasing in complexity due to the additional parsing required to analyse the XML schema restrictions. A trade-off between compression size and time exists when considering XML-conscious and schema-informed compressors respectively.

In conclusion, an optimal compression size can only be achieved in presence of external knowledge provided by a schema language or by the user. Documents and compressors, which do not benefit from a Schema-informed approach, are not able to achieve significant difference in compression compared to traditional general-purpose compressors. The ability to achieve compression ratios beyond general-purpose compressors is given by the use of the XML structure information provided by the schema. This ability can be categorised as the key feature to achieve a higher ratio when compressing highly structured data-centric documents.

# Chapter 4

# XML compression techniques for efficient network management

This chapter investigates the use of compression tools and their application in the field of network management. This study focuses on native languages implementations of different XML-conscious and schema-informed compressors to improve the overall performance of the system. The scenario of network management is introduced together with the application of XML compression techniques for a set of relatively small, highly-structured XML data. The methodology provides details on the XML corpus and the compressors execution performed to obtain the results. This study discusses the results of the experiments and provides an explanation for the performance of different tools in compression size and speed. Finally, the chapter concludes by highlighting the research gaps in XML compression and provides future directions to improve the design of XML-conscious tools.

## 4.1 Introduction

The ability to monitor environmental behaviour and obtain sensor information from embedded devices has extended the field of network management. There has been a shift towards processing data using low-powered devices where computational power, memory and storage capacity are restricted. These devices are capable of collecting data from their immediate environment such as

temperature, motion, sound and even levels of radiation and seismic activity (Estrin et al., 1999). Data is gathered and then processed from a variety of devices which range from network switches to single-board computers equipped with sensors for industrial, civil, or military purposes (Lifton et al., 2007; Winkler et al., 2008).

Several issues arise when these devices are connected autonomously. Processing power, battery life and the design of the network protocol are the key limitations to overcome in order to increase the capacity of these devices and overcome their operational limitations. Battery-friendly processors exist which can accommodate the software demands and at the same time avoid having a negative impact on battery life. However, in this environment, the communications network still poses a challenge. Even if the battery life barrier can be overcome by using other sources of energy e.g. solar panels or kinetic chargers, these devices cannot afford to connect to high-bandwidth networks.

The amount of data shared on these networks varies depending on the field where the technology is applied, however, having highly structured data is key to supporting applications such as network management. As the amount of data transmitted over the network may be limited, an efficient representation of structured data is essential.

Messages represented in XML are widely used to interchange data over the Internet. Depending on the scenario in which the technology is applied, messages might be sent in a semi-continuous stream, or triggered in response to an event or threshold being reached. For example, medical applications will have different needs compared to environmental recording systems. Sending XML via a constrained network is a challenge. The verboseness and redundancy present within a message can result in issues of scalability. Several studies (Augeri et al., 2007; Cheney, 2001, 2005; Sakr, 2008, 2009) have developed methods to compress XML. Although these techniques have mainly focused on document compression for database storage, they could also be applied in the field of emerging wireless networks. General text compression techniques can reduce up to 70% the size of the original file (Sakr, 2009). However, there are drawbacks with these approaches as the back-end compressors are usually not XML-aware. In addition they do not employ any high-level validation which is easily applied to XML using a schema. Validation of real-time data is a significant advantage when managing a network of embedded devices.

## 4.2 Background

### 4.2.1 SNMP

For decades, this field has been mainly dominated by a popular protocol for managing networks systems and devices on IP networks. The Simple Network Management Protocol (SNMP) has been widely used to manage medium scale network systems equipped with devices such as routers, switches, modem and sensors (Case et al., 1990). This internet-standard protocol is controlled by the Internet Engineering Task Force (IETF) working group which has continued working on this protocol for several years providing improvements and more functionalities (Frye et al., 2003). The simple design of this protocol has been one of the main advantages which allowed users to define variables to monitor and further expand their system. This led to an extensive use enabling interoperability and the implementation across the major hardware vendors. However, because of its simplicity, SNMP is not capable of supporting a growing network due to its constraints in scalability (Corrente and Tura, 2004). Several implementations have demonstrated how the major drawbacks of SNMP can be overcome by using XML-based network management replacing SNMP components with XML to both manage the data and define the structure (Shin and Shim, 2005).

### 4.2.2 Related work

Extensive research has been conducted to improve network management in constrained networks (Marrón et al., 2005; Yoon et al., 2003). Wireless Sensor Networks (WSNs) is one area where XML compression has been used to improve data management. Sensor node hardware restrictions have been addressed using an XPath engine on updateable compressed XML data, reducing memory and energy consumption (Hoeller et al., 2009). Furthermore, this work has been extended to support larger sets of data within the sensor network using stream-oriented XML compression in order to have a dynamic queriable system (Hoeller et al., 2010).
This chapter investigates the use of XML compression techniques to improve network management over a wireless embedded internet, however, results can

be extended to general networks. The results of this chapter provide further contributions to the research carried out so far in this area by presenting tools that achieve higher compression ratio across data sets typically found in network management applications.

### 4.2.3 Motivation

Software complexity is not usually considered an issue when discussing XML compression techniques. However, complex compression software results in slower compression speeds. Both run-time parsing and data encoding influence the speed of the entire compression tool. When these procedures are executed at run-time they can have a considerable impact on an embedded system. Therefore, traditional document compression tools must be revised and adapted to satisfy these requirements. Connected to this issue is the type of programming language used. Several studies have focused their attention on the use of virtual machines (VMs) for embedded devices using ahead-of-time compiler (AOTC) or just-in-time compiler (JITC) resulting in performance and memory consumption improvements (Álvarez Gutiérrez and Soler, 2008; Badea et al., 2007). However, these approaches are constrained to particular platforms where these VMs have been ported. This study focuses on languages such as C/C++ which generate native code and are not dependant on a virtual machine. In addition, even if light weight VM-based languages can perform close to low-level languages such as C/C++ (Pizlo et al., 2010), C/C++ is dominant in the embedded computing domain. Apart from performance, other factors must be considered. For example, most compression techniques do not focus on the importance of compressing valid XML data. The assumption that the data is valid during compression is a risk that should be eliminated in a network management application. Therefore, there is an interested in techniques that employ a schema to provide validation. Other non-compression uses for a schema include the ability to provide light-weight security by employing the schema as a key to encrypt and decrypt the XML data.

The areas where this research applies mainly focus on improving network management across a wireless embedded internet. The acronym FCAPS refers to the five subcategories of network management, Fault, Configuration, Accounting, Performance and Security management, initially introduced in the 1980 by

the OSI System Management Overview ISO. The results of this chapter aim at improving the performance management of networks where main factors of this subcategory are severely threatened. Network capacity of wireless embedded internet systems is constrained due to the latency, packet loss and throughput limitation. Due to the constrained throughput, sending uncompressed data is not feasible as it could increase the packet loss resulting in retransmission delays. By reducing the size of data that needs to be communicated, it is possible to increase the robustness of the system and ensure a higher performance for low-bandwidth networks. In addition to performance management, the ability of validating data is a real improvement for the configuration management category.

### 4.2.4 Network Challenges

The IEEE 802.15.4 is a standard for low-rate wireless personal area networks LR-WPANs with low-power transmitters and power consumption. This standard is at the basis of the wireless communication between small devices which are usually related to technologies such as ZigBee, WirelessHART, MiWi, and other non-IP based protocols. ZigBee in particular is usually confused with the 802.15.4 standard, however 802.15.4 defines a physical PHY and Media Access Control MAC layer protocols whereas ZigBee is a network layer which is based on top of the 802.15.4 standard. Wireless sensor network WSN defines a technology which is mainly based and heavily influenced by the 802.15.4 standard. WSN consists of small devices capable of communicating environmental data using built-in sensors typically distributed in a mesh network with sensor and gateway nodes.

Some of these networks based on the 802.15.4 standard rely on a network layer, OSI layer 3, which does not provide the TCP/IP protocol. However, as the technology is moving towards the idea that even the smallest device should be able to communicate to the internet, IPv6 over Low-power Wireless Personal Area Network 6LoWPAN has been designed to send and receive IPv6 packets over the IEEE 802.15.4 standard. The main advantage of this is implementation is the ability to communicate to the smallest device using an IP, hence “Internet of Things”, however the cost of this implementation is considerable. The data rate is the first limitation found in this environment. The IEEE 802.15.4

standard provides a maximum data rate of 250 kb/s for the 2450 MHz PHY and a lower data rate of 20-40 kb/s for the 868/915 MHz PHY. Data rates can be employed based on user preferences and regulations. However, in order to minimise power consumption, a lower data rate is usually preferred. Another limitation is found in the maximum transmission unit MTU. The MTU for the IEEE 802.15.4 standard is 127 bytes of which 25 and 21 bytes are reserved for the frame overhead and the link-layer security respectively. In addition, from the remaining 81 bytes, 40 are allocated for the IPv6 and either 8 or 20 for the UDP and TCP headers, leaving 21 bytes for the actual data. If considering data rates, packet loss, latency and the low throughput, sending data over such networks becomes a serious challenge. Some of these issues have been addressed with the RFC 4944 (Montenegro et al., 2007) by applying an adaptation layer mainly for header compression in order to provide more space for the payload.
In this environment the network still poses severe challenges which influence the performance management of these devices. The processing capabilities are not a major issue when compared to the networks limitations. Investing more processing power to improve the compression of the data that needs to be communicated would result in a better performance of the network, reducing delays and possible packet loss. Discussions have been raised whether is it necessary for these networks to have a TCP/IP protocol stack, however, the aim is to improve network management by focusing on the data and software development, which can be applied on top of OSI layer 3 with both IP and non-IP networks.

## 4.3 Methodology

### 4.3.1 XML Corpus

There are several XML test corpus that have been used over the years to test compression tools. The W3C developed a test suite to define the conformance of XML and XML Schema (Thompson et al., 2011). However, this area is fragmented and lacks data sets that fit each specific application domain. In a network of limited bandwidth, sending large amounts of data is not feasible. For this reason, this experiment excludes large documents which are common in compression experiments and mainly focuses on small highly structured XML data.

In addition, sensor data, or data collected from managed devices, is mostly highly structured without being dominated by string data.
The test corpus used for the experiment is not only based on the system requirements but also collected from IP network devices that used SNMP and the equivalent XML data. This Internet standard protocol manages devices such as routers, switches, IP telephones and sensors in a structured and restrained format which is highly relevant to this scenario. The test corpus used for the experiments was also assembled based on the recommendations from W3C.

For a fair experiment different data sets that contains both numerical and string data within a constrained size are used as shown in table 4.1. Using the XML metrics and classifications described in Chapter 2 is it possible to classify this data set as a collection of data-centric structural XML files. Each XML file is highly-structured with a maximum document depth of 6 levels of nesting. The average size of the data set is around 800 bytes. The data set can be categorised as single XML messages streamed across a low-bandwidth network and as a structured sequence of XML data which are grouped and sent at specific times. The Numeric and String columns of table 4.1 informs us on the variety of data types which are contained in the data set. The Numeric column contains sets of *basic* data types which belong to simple and complex forms that can be easily represented with an n-bit unsigned integer using bit-string, integers, IPv4, etc. The String column is used to represent *String* data types such as UTF8-Strings. This distinction is important to identify encoding differences for compressors based on derivates of ASN.1 PER encoders.

| **Name** | **Bytes** | **Tags** | **Depth** | **Numeric** | **String** |
|---|---|---|---|---|---|
| iptel-devinfo | 702 | 36 | 2 | 5 | 12 |
| iptel-ethinfo | 692 | 36 | 2 | 16 | 1 |
| personnel | 903 | 53 | 5 | 4 | 16 |
| purchaseorder | 704 | 50 | 4 | 9 | 9 |
| router-addnet | 566 | 32 | 4 | 3 | 8 |
| router-disc | 834 | 42 | 3 | 7 | 9 |
| router-qos | 870 | 50 | 4 | 0 | 18 |
| sensor | 591 | 52 | 6 | 14 | 2 |
| switch-config | 731 | 40 | 3 | 6 | 11 |
| temp-sens | 1355 | 94 | 5 | 17 | 23 |

TABLE 4.1: XML Data Sets

Listing 4.1 displays a typical structure of an XML message which is exchanged in network management environments. The following is part of the *temp-sens*

XSD file which defines the structure of a temperature sensors for monitoring of server rooms and infrastructures. The XSD example shows the use of enumerated and boolean simple type restrictions to apply additional constraints on the information that can be shared across the network.

LISTING 4.1: Sensor Data Structure

```
1
2   <xs:complexType name="Rtype">
3     <xs:sequence>
4       <xs:element name="Name">
5         <xs:simpleType>
6           <xs:restriction base="enumerated">
7             <xs:enumeration value="Binary1"/>
8             <xs:enumeration value="Binary2"/>
9             <xs:enumeration value="Binary3"/>
10          </xs:restriction>
11        </xs:simpleType>
12      </xs:element>
13      <xs:element name="Number">
14        <xs:simpleType>
15          <xs:restriction base="enumerated">
16            <xs:enumeration value="Input1"/>
17            <xs:enumeration value="Input2"/>
18            <xs:enumeration value="Input3"/>
19          </xs:restriction>
20        </xs:simpleType>
21      </xs:element>
22      <xs:element name="Value"type="boolean"/>
23      <xs:element name="Alarm">
24        <xs:simpleType>
25          <xs:restriction base="enumerated">
26            <xs:enumeration value="Yes"/>
27            <xs:enumeration value="No"/>
28          </xs:restriction>
29        </xs:simpleType>
30      </xs:element>
31      <xs:element name="State"type="boolean"/>
32    </xs:sequence>
33  </xs:complexType>
```

### 4.3.2 Compressor Execution

Table 4.2 lists the compressors used for the experiments along with code repositories. The table shows how the focus of this work is on XML-aware compressors with particular attention to DTD or schema-informed implementations. A *zlib* implementation (Adler, 2005) is used in the experiments in order to understand the extent to which it is advisable to use a general text compressor. As mentioned in the previous section, the focus of this study is on languages with a minimal level of abstraction from the machine. This means that only XML compressors that are developed in C or C++ and are portable to variety of embedded platforms are considered. This study excludes direct comparison against other languages as the programming language choice can influence the performance results. Application commands used to compress the data set and calculate compression speeds are listed in Appendix C. Hardware used to perform the experiments is specified in table B.2 of Appendix C. The software was compiled using specific flags to specify the machine architecture and the highest level of optimisation provided by the GCC compiler (Jones, 2005) in order to achieve the best performance for each of the XML compressors listed in table 4.2.

| **Compressor** | **Type** | **Schema-aware** | **Language** | **Parser** |
|---|---|---|---|---|
| XMLPPM(Cheney, 2006c) | XML-aware | - | C | SAX |
| DTDPPM(Cheney, 2006a) | XML-aware | DTD | C++ | SAX |
| WBXML(Jehanne, 2009) | XML-aware | DTD | C | SAX |
| XMILL(Colver, 2004) | XML-aware | - | C++ | SAX |
| ZLIB(Adler, 2005) | General | - | C | - |
| PO(Moore, 2012) | XML-aware | Schema | C | DOM |

TABLE 4.2: XML Compressors List

## 4.4 Results

### 4.4.1 Compression Size

To compare the performance of XML compression tools a number of tests were executed to calculate the compression size, speed and a size/speed ratio. A test was performed using a script to loop the compress and decompress function

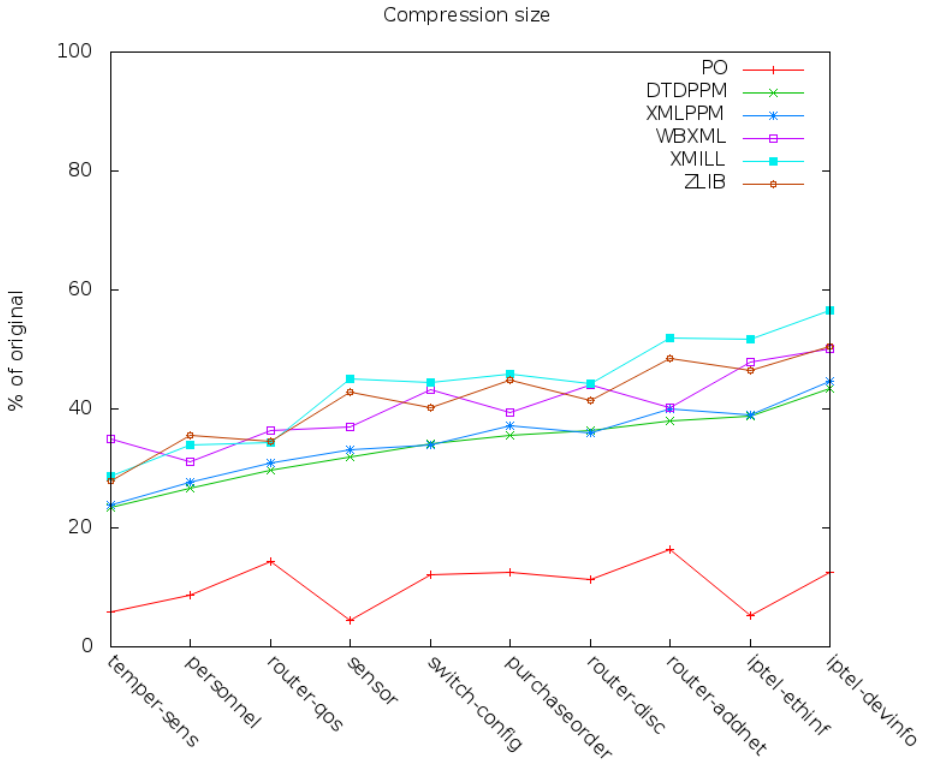


FIGURE 4.1: Compression Size Results

*100* times. In addition, each test was performed *15* times to calculate the mean of the compression/decompression times. Results have been plotted using line graphs with the x-axis sorted by *size* of the XML file in relation to the *numeric* and *string* data types.

Figure 4.1 shows the compression size as a percentage of the original data. Most of the compression tools achieve a level of compression between 20% to 60% of their original size, whereas, a schema-informed compression tool such as PO is able to reduce the size of the XML to between 2% to 20%. The compression tools which are based on statistical and dictionary techniques are influenced by the size and the structure of the XML data where the highest level of compression is achieved for larger data sets. PO, however, is not adversely affected by the size of the XML. The level of compression is influenced by how highly structured the data is, as defined by the data types available in the schema language. For example, the highest level of compression is achieved for data sets such as *sensor* and *iptel-ethinfo* which contain numeric data types. Contrarily, *router-qos* and *router-addnet* cannot achieve similar results as the data types are mostly string based.

The performance of PO can be predicted based on data types presented in the

schema. String types do not benefit from the higher encoding mechanism available for basic data types. The IER back-end of PO is not able to apply a statistical compression, therefore, each string is fully encoded in a 7-bit compressed format. The compression size difference is the result of the schema-informed approach of PO, which allows the tool to encode only data types sequentially without XML structure information. DTDPPM is not able to achieve a significant difference from XMLPPM using information provided in the DTD, it can be assumed that the XML structure information is included in the compressed format and that the advantages of DTDPPM are only triggered in few specific cases. An interesting result is provided by compression patterns between XMill and zlib. The advantage of using the XML-conscious approach of XMill is highly based on the knowledge that needs to be provided by the user. This additional information is essential in order to achieve additional compression results.

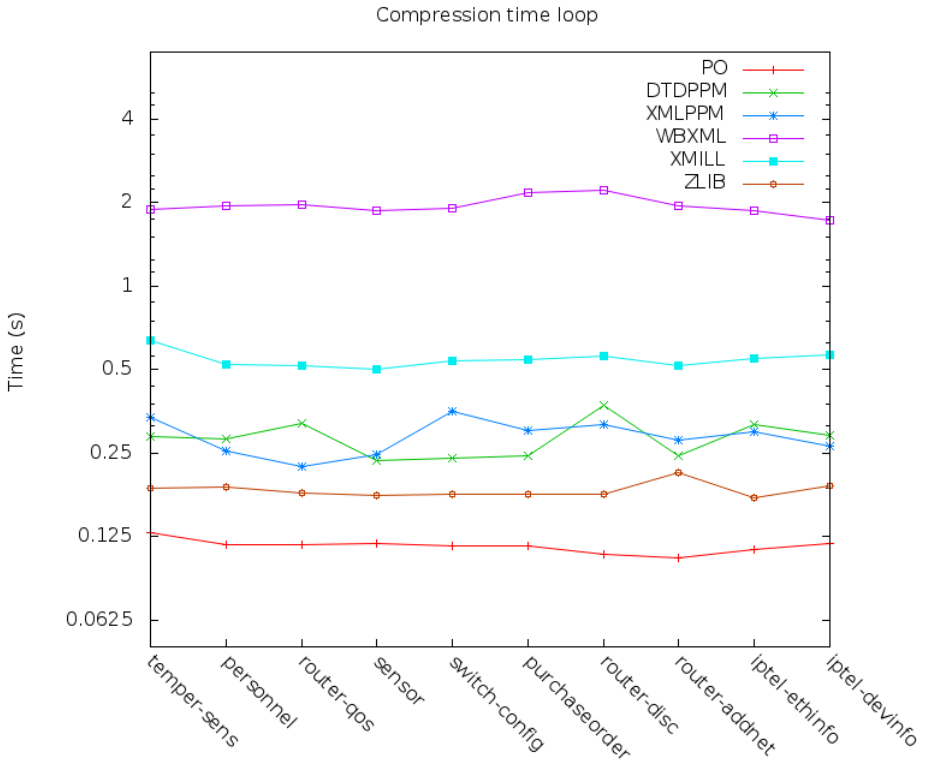


FIGURE 4.2: Compression Time Results

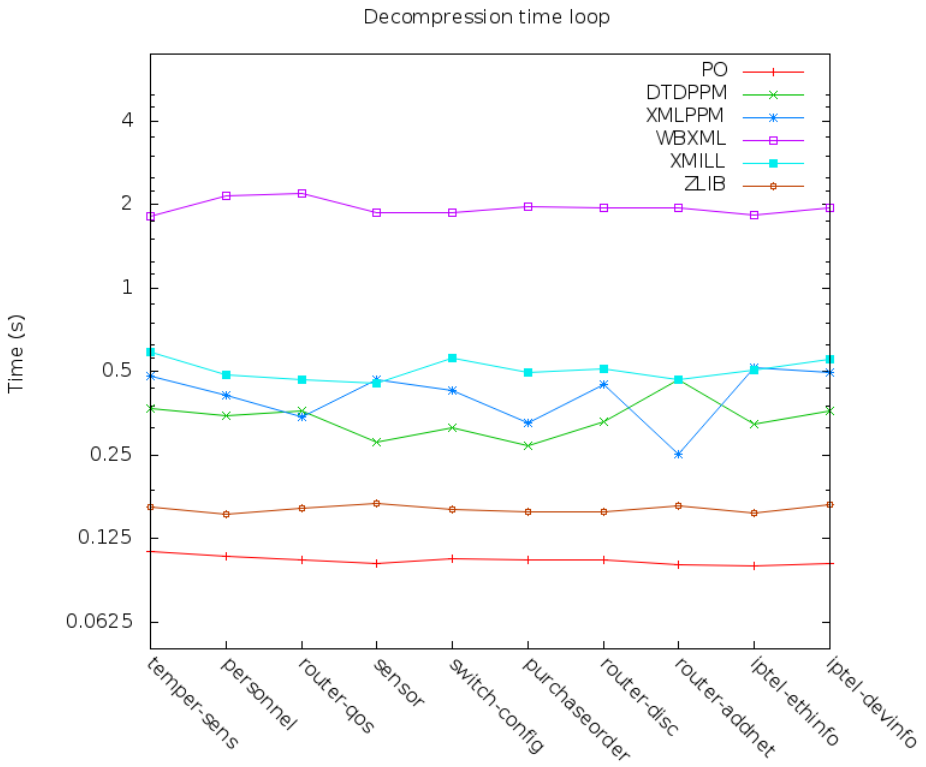


FIGURE 4.3: Decompression Time Results

### 4.4.2 Compression Time

The second experiment is based on the time required to compress and decompress XML data. Figures 4.2 and 4.3 show the results for compression and decompression for each compressor on a logarithmic time scale. The speed of the tool is affected by several factors. The back-end compressor together with the parser implementation is usually considered the main factor that determines performance. The experiments focus on small highly-structured XML messages with a particular emphasis on the ability to provide validation. Instead of parsing only the XML data, compression tools that perform validation have to parse and extract information from a DTD or Schema. However the system architecture plays an important role when dealing with this additional burden. The experiment was performed calling the compress/decompress function on a loop of 100 for 15 times. Each result is the average of 1500 single compress/decompress calls. Using this metrics it is possible to calculate the average time required by the tool to compress and decompress the XML data set.

### 4.4.3 Speed/Size Ratio

The ratio was calculated using equation 4.1 to contrast different tools where $size^{1st}$ and $time^{1st}$ refer a particular tool and $size^{2nd}$ and $time^{2nd}$ to the other the ratio is comparing against. This ratio is used to examine the significance of both encoding speed and encoding size when contrasting tools. Thus, any score above 1 would indicate a tool that outperforms the analysed one.

$$\textit{Speed/Size ratio} = \left(\frac{size^{1st}}{size^{2nd}}\right) \times \left(\frac{time^{1st}}{time^{2nd}}\right) \tag{4.1}$$

As shown in figure 4.4, all compressors underperform against PO, particularly data sets with numeric data types such as *sensor* and *iptel-ethinfo*. PO performs better compared to ZLIB, XMILL and WBXML for all the data sets. XMLPPM and DTDPPM are more efficient compared to PO only for few data sets which mostly contain string-based data. PO is more efficient for numeric data sets such as *sensor* and *iptel-ethinfo*. Although it achieves consistent results when compared to other tools, WBXML presents the worst ratio due to the poor speed of the tool. Overall best results are achieved by the schema-informed compressor PO and the general-purpose compression library zlib. Tables B.3 to B.8 of Appendix C provide the details of compression ratios to highlight the performance of different tools against the data set.

### 4.4.4 EXI format

Another experiment was performed to highlight the differences between the level of compression and speed achieved by PO and an EXI implementation. This experiment was separated from the previous one as the programming language differs. There are currently only Java implementations of EXI compressors that are open source and in a functional state. The level of compression achieved using EXI is similar to that achieved by PO. The experiment was conducted using *EXIficient* library with *EXIprocessor* (Garrett, 2012) tool. Figure 4.5 shows the similarities between the two compressors as both tools use a schema-informed compression with similar encoding. Both encoders are based on ASN.1 PER encoding. Using similar encoding mechanisms to map complex simple types to bit level, EXI and PO compression sizes differ only by few

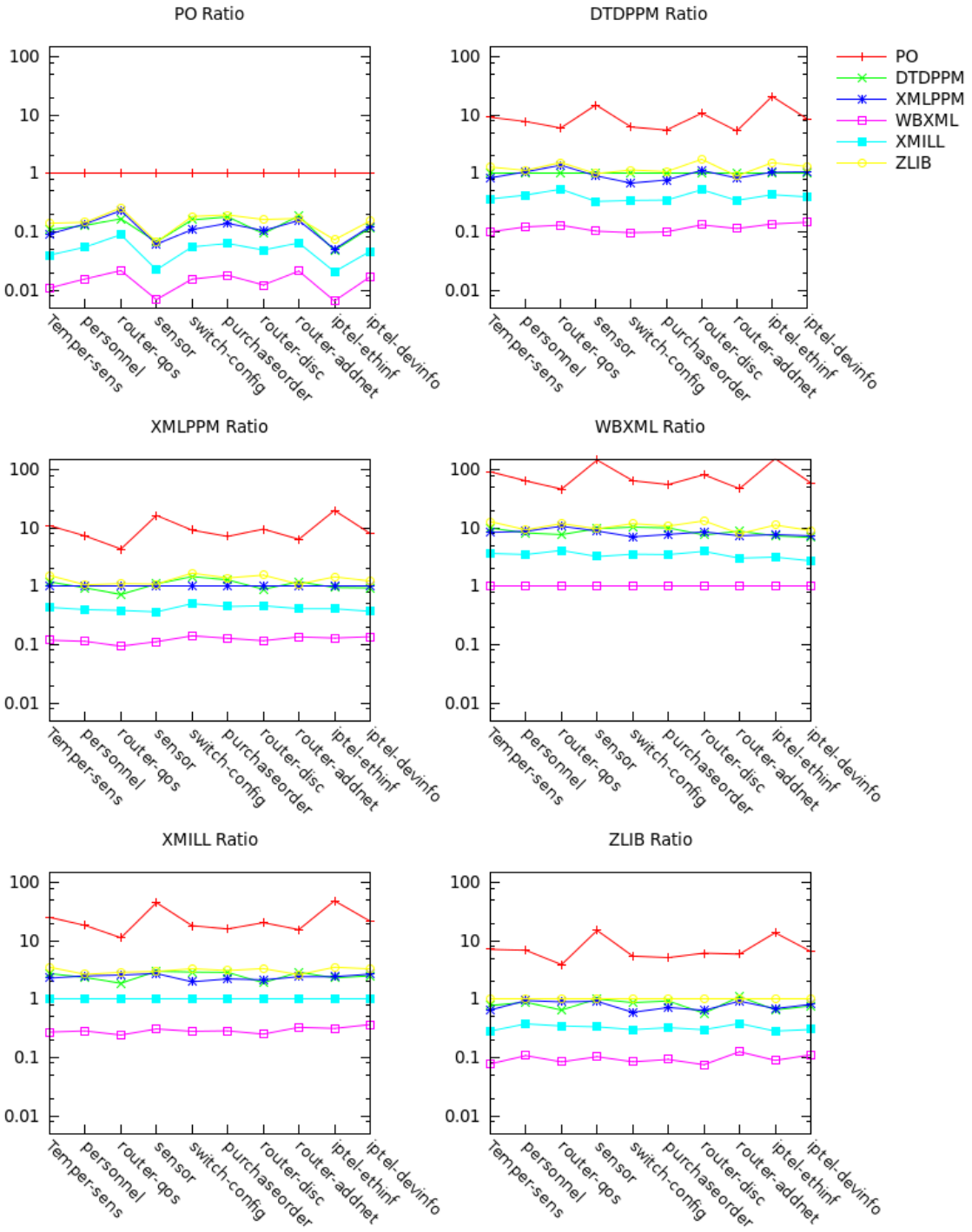


FIGURE 4.4: Compression Ratios

bytes (EXI can use up to two bytes for the header). The significant difference between these tools is the speed at which they operate. Although EXIficient is Java based this should not account for the vast difference in performance.

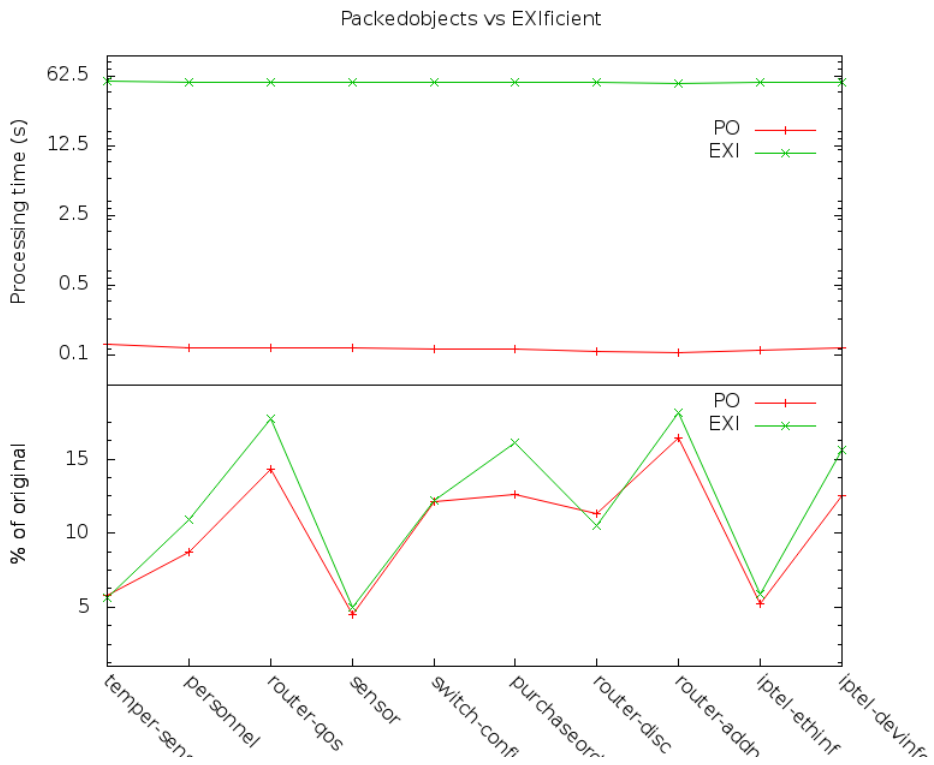


FIGURE 4.5: PO vs EXI Format

## 4.5 Observation

The results of the experiments described in section 4.4 demonstrate the performance of compression tools for an XML data set of relatively small, highly-structured XML files. The use of a schema for defining data types provides information which can be used to improve the compression. A key disadvantage of using a schema is related to the added complexity of parsing an extra XML structure, however this can be managed by a one-time fixed cost within implementations such as PO. In a wireless embedded internet the true bottleneck is network capacity where the size of the data to be communicated has a significant impact on overall performance and the ability to scale usage of the network. In addition to performance, validation is an important feature for XML compression tools. The results demonstrate how schema-informed compression can reduce the size of XML data yet still outperform other general purpose XML compressors.

Apart from zlib, all the tools used to compare against PO used a SAX parser. This technique usually performs better than DOM parsing in terms of speed depending on the application. With SAX there is no requirement to create a

memory representation of the data and therefore it can be more efficient for large data sets. A DOM parser, however, creates an internal tree structure in memory enabling more high-level and developer friendly functionality you would expect when creating a network management tool.
These experiments were performed in order to provide useful information to developers and researchers on how to select a compression tool for this particular domain. The results show that PO is the most effective solution for efficiently handling XML-based network management data across restricted communication networks. It was demonstrated how size is the most important factor to be considered when compression is applied to a low bandwidth network if the difference in processing time between tools is minimal. The results can provide the following observation:

- PO can perform better than other XML compression tools studied.
- Statistical and dictionary-based compressors are more suited to large document compression.
- Schema-aware compressors provide the best compression size.
- Schema-aware compressors can provide validation which is an important addition to applications collecting data across a wireless embedded internet.
- As there is no use case to support large XML data sets, developers can benefit from the advantages of using DOM over a SAX parser.

The results of the experiments demonstrate a substantial reduction of the size of the data that needs to be communicated over the network. In addition to this reduction, another advantage is the compression time required to compress and decompress the XML data. Using a schema-aware compression it is possible to reduce the size of the data beyond the limits achieved by conventional compression tools. In addition, the software parser is developed to provide an application programming interface to further improve network management. These features are the key components for the network performance improvements over a low-bandwidth network. The extra validation and parser functionalities provided by the software also enhance the configuration management of these networks. Devices based on the IEEE 802.15.4 standard are still undergoing

an exponential growth in processing capabilities yet the network performance is constrained mainly by the physical layer. For this reason this study mainly focused on the compression of XML data in order to apply these results not only to a wireless embedded internet but also to general networks.

## 4.6 Conclusion

This chapter compared XML compression tools for their potential to support network management applications over a wireless embedded network. The efficiency of several compressors are examined in a specific network environment where the physical layer is the main bottleneck of the system. Results are based on the ability of each compressor to handle highly structured data with a range of string and basic data types. The main difference highlighted in the results is the compression size achieved by schema-informed techniques compared to standard XML-conscious. It was noticed how to performance speed is highly related to the complexity of the application even when schema-informed tools are burdened with the additional schema file.

A number of XML compressors have been evaluated particularly in the field of network management in order to obtain knowledge on the performance of tools listed in Chapter 3. With few exceptions, experiments conducted in most previous research do not focus on a wider range of data sets but on few well-known XML files. Therefore, the performance of most tools when presented with relatively small highly-structured data sets was unknown. The results of this chapter fill this research gap presenting the performance in compression size and speed for this data set which is usually found in the field of network management.

The results of this chapter show the state-of-the-art of XML-conscious compressors by focusing on a number of academic and industry applications. This study compared several tools with general-purpose back-end algorithms and various XML binary representations based on ASN.1 encoding rules. Based on these results it is possible to conclude that schema-informed approaches are the most efficient techniques to compress XML documents. The ability of these techniques to compress XML data without including XML element information is the key feature to minimise the redundancy of XML. This is possible due to the robust structure of XML which can be thoroughly described in a DTD/XSD file. In addition, data type information provided in the schema language such

as element data types, restrictions and enumeration play an important role by providing the back-end compressor enough information to compress data using the lowest number of bits.

Disadvantages of XML compression techniques are related to the complexity required to parse XML and the schema. While binary representation based on SAX can be considerably faster compared to a DOM API, lowest compression sizes can only be achieved using a schema-informed approach. This technique requires additional complexity due to the XML Schema patterns and restrictions mapping process. In addition, Schema-informed approaches are always limited by the information provided to describe XML data. These techniques in particular are forced to comply with the information of DTD/XSD files. This limitation can be triggered by using different data types which are not recognised by the compressor. Without information of a schema, compression techniques can only rely on the front-end part of the system to manage XML elements using algorithms such as those described in XMILL and XMLPPM.

# Chapter 5

# Hybrid XML Document Compression

This chapter investigates how to best compress XML data using a hybrid compression system developed to improve the efficiency of current technologies. Based on XML compressors evaluated in Chapter 3 and the comparison results of Chapter 4 this research proposes a method that incorporates two compression systems which are executed when their best use cases are triggered. The first section provides the motivation based on knowledge of XML data sets and the experiments performed in previous chapters. This part discusses the requirements of a compression tool capable of improving XML compression with possible extension to other markup languages. Part of this chapter is dedicated to the system architecture of the hybrid compression system to describe various components required to achieve higher compressed formats. A motivating example based on a typical XML document is provided to demonstrate how each component performs and the possible variations that can improve the final compression size. The chapter concludes by describing the applicability of the hybrid model and the limitations of this approach.

## 5.1 Motivation

The design of the hybrid model is based on knowledge of current compressors and information provided by XML data sets analysis. The aim is to improve the design of XML compressors in order to achieve the best compression size with a wider range of data sets. In addition, this work aims at achieving substantial results with different data types which may not be suitable for XML compression techniques. Chapter 3 discussed a number of XML compressors later evaluated in Chapter 4. The compression size and performance for small data sets was tested in the field of network management to fill research gaps in compressing small highly-structured data sets. Results demonstrate the efficiency of schema-informed compression techniques compared to standard XML-conscious. The disadvantages of implementing these techniques have been identified and related to software complexity and availability of the schema language. Simple XML-conscious techniques lack knowledge of both structure and data types of XML. Therefore, without the knowledge provided by a schema language, the XML structure has to be encoded in the compressed format. For example, a schema-uninformed technique will have to store data such opening, closing and empty tags. Furthermore, a schema language also provides information related to XML data types. Without incurring additional processing required to recognise data types, the compressor is able to handle data more efficiently using knowledge provided in the schema language.

The design of the hybrid compressor is based on statistical analysis performed in XML and Schemas compression and data management research areas. The analysis of XML data has shown how schema languages are common only for a specific type of XML documents where simple type restrictions are not designed to aid compression. This intention of this research is to improve compression for documents of various sizes which do not benefit from a descriptive schema language. For example, a schema language can be generated and used to compress documents with identical structure but different data. Different types of XML documents such as structural and textual can achieve better compression when informed with a schema language. As analysed by the XMill compressor, textual XML documents can take advantage of a structured XML form to separate various data types and increase compression with semantic containers. Using a schema language it is possible to minimise compression even for expanded XML documents. This work aims at improving compression

for data sets of irregular XML documents that cannot be validated. This can be achieved by transforming documents into more suitable forms.
The design of the hybrid compressor aims at improving compression for a wider range of XML data without restricting application to a particular area. Most XML compressors do not achieve efficient results for different types of XML when considering structure, size, data types and schema availability. The design of this tool is aimed at improving compression for XML, however, it can be extended to all markup languages with minimal changes to the front-end of the system that transforms high-level syntax into a lower format.

## 5.2 System Requirements

A set of requirements are defined for the hybrid compression model to efficiently compress XML data. These requirements are based on studies of XML compressors in order to improve the compression size.

**R1: Support wider range of XML data sets**

The first requirement is to efficiently support a wide range of data sets constructed from document-centric and data-centric XML files. Each tool described in Chapter 3 is best suited for a particular type of XML, based on the area where these compressors are applied. For example, XMLPPM and DTDPPM are designed to solve problems relevant to web data management focusing mainly on minimum-length coding for efficient XML storage and transmission. Queriable XML compressors, instead, focus on storage techniques for efficient XML database query and processing. Few tools have been designed with the intent to support a wider range of applications. This is due to the difficulties in designing a tool capable of supporting various XML data sets and scenarios. The aim of this requirement is to provide the best performance for a wider range of XML documents while matching the performance of general-purpose compressors for unsupported documents.

**R2: Manage Schema-uninformed compression efficiently**

A severe disadvantage of XML compression is related to the performance of schema-uninformed techniques. The results of Chapter 4 illustrate

a clear difference in compression between schema-informed and uninformed techniques with around 20% additional compression achieved by the first. Although tools such as XMill have introduced additional levels of compression using atomic types applied to XML values with required patterns, the lack of a schema to define the structure of the document is the major drawback. This requirement focuses on the development of an efficient schema-uninformed compression technique to provide the ability of defining the structure of XML. This feature will reduce the information stored in the compressed format, essential for compressing large data sets with repetitive XML tags. An important advantage gained from the development of this requirement is the ability to apply the best data types to specific values. Mapping basic data types to an efficient representation is essential to achieve a higher compressed format. For example, bit strings can be converted into a sequence of octets using only 1 bit per character data, saving 7 bits for each symbol.

**R3: Separate Structure from Data**

Separating structure from data is an essential requirement for XML schema-informed compression techniques. Few compressors are based on the local homogeneity property to enhance compression. Implementing this feature together with a schema-informed approach can increase the compression size. It is possible to highlight some similarities between the homogeneous compression of XMill and the schema-informed approach of PO. XMill streams data into semantic containers based on XML element and user-defined options. A number of containers are created based on the amount of different data types of the XML. Here, the structure container is used to compress XML structure into a more concise format exploiting the data redundancy and similarities. The schema-informed approach can be visualised as a similar local homogeneous version, where the XML structure information is stored in the Schema language therefore reducing the compressed format from the structure container. This requirement aims at incorporating features of schema-informed and local homogeneous approaches.

Table 5.1 illustrates where the requirements listed above are implemented in current compressors. Most of the tools satisfy the requirement R1 with exception for schema-informed DTDPPM and PO which are focused on a particular

| Requirements | XMLPPM | DTDPPM | WBXML | XMILL | PO | EXI |
|---|---|---|---|---|---|---|
| **R1** | ✓ | × | × | ✓ | × | ✓ |
| **R2** | × | × | × | ✓* | × | × |
| **R3** | × | × | × | ✓ | × | ✓ |

TABLE 5.1: Requirements on XML Compressors

domain. Due to the additional complexity, EXI is able to satisfy R1 using both schema-informed and uninformed approaches. An efficient schema-uninformed compression can be found in XMill compressor which is able to achieve higher compression sizes compared to general-purpose compressors. However, requirement R2 can only be achieved with the user's intervention exploiting data type knowledge. Requirement R3 is found in XMill, the first local homogeneous compressor, and to some extent in EXI. The latter implements R3 by aligning event into a semantic bit-aligned stream more prone to compression.
The hybrid compression system described in this chapter aims at implementing all requirements based on the knowledge and performance of current compressors. From a high-level point of view, the requirements can be achieved using aspects and properties of the best compressors. The following model describes the architecture of a system capable of handling various XML files and different scenarios.

## 5.3 Hybrid Compression Model

This section describes a compression model based on the techniques evaluated in Chapter 4 and the requirements described in section 5.2. Diagram 5.1 illustrates the stages required to compress an XML document using different techniques to enable a schema-informed approach with local homogeneity properties. The diagram introduces canonical (canon) XML and XSD which are generated from the stages of the model in order to convert XML into a PO subset and allow a schema-informed compression. Both forms will be described in more detail in the next sections.

The main concept of the model is the separation between basic and character string types, which are compressed using their best encoding technique. The first stage, Document Transformation, is required to transform an XML document into a subset of XML which can be accepted by the PO encoder. The

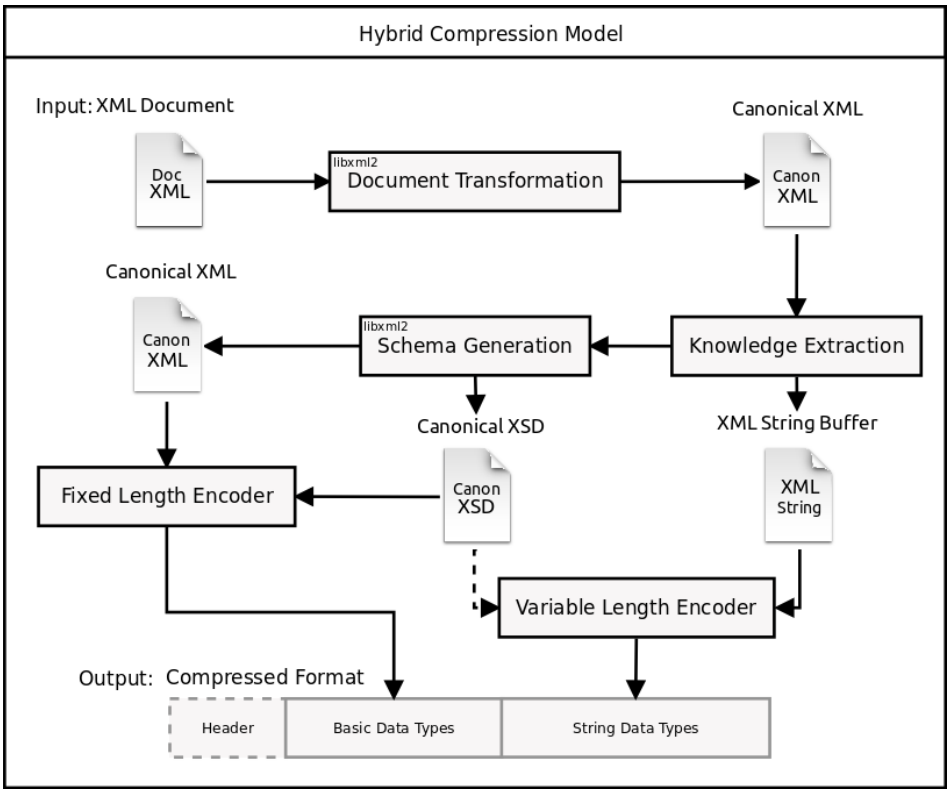


FIGURE 5.1: Hybrid Compression Model

second stage, Knowledge Extraction process, is linked to the Schema Generation. To benefit from local homogeneity properties, the model extracts large string types from the XML data and replaces them with a place holder integer for decompression purposes. A buffer containing string types is created together with a Schema language to validate the XML data. The result of the first three stages is an XML document with integer values pointing to string types stored in a memory buffer. A schema language is also automatically generated based on the information collected from the Knowledge Extraction parsing process. The latter is used to provide a schema-informed compression using a fixed length encoder. Depending on the scenario, the XSD can be stored locally or compressed with the string buffer using a variable length encoder. The result of the compression model is a binary format containing basic types compressed efficiently using a fixed length encoder and the string buffer compressed using a variable length encoder. The approach described in this model is based on the best compression techniques applied to character string types using *zlib* and basic types using PO. The differences between the proposed model and PO are based on the additional stages required before the XML document is passed to the low-level encoders. While PO requires users to provide the XML

Schema, the hybrid model automatically generates an optimal schema that can be subsequently stored with the compressed format. In addition, by separating character string data types from the basic data types, it is possible to incorporate local homogeneous compression properties. The next sections describes the encoding and decoding process for each of the stages described in the model diagram.

### 5.3.1 Document Transformation

The first part of the model focuses on transforming XML documents to a subset of XML standard. This transformation process is needed to allow various XML components to be recognised by the low-level encoder. As discussed in section 3.1.8, PO XML is restricted by an additional schema which does not allow the use of attributes, comments and other non-structural components. Therefore, since PO can only accept a subset of XML, a transformation process is required to enable non-structural components to be recognised and validated by the schema-informed compressor. The term *canonical* (canon) is used to define this specific subset of XML which should not be confused with the standard definition of Canonical XML.

Since data is only available in element components and nested hierarchically, this form can be described as a highly-structured document. Code listing 5.1 presents an example of EBNF to describe the canonical XML accepted by PO. XML documents are based on a prolog and a single root element. Compared to the specification of XML described in (Bray et al., 2008), only two components are allowed while the use of miscellaneous is restricted. Elements can only contain character data or nested elements. This subset allows standard definition for element name, thus the use of namespaces, which are treated as standard element names by the encoder. In summary, compared to the standard EBNF of XML 1.0, this subset is based on minimal components mainly restricted to the use of elements.

LISTING 5.1: Canonical PO XML EBNF definition

```
1 document        ::=     prolog element
2
3 prolog          ::=     XMLDecl
4 XMLDecl         ::=     '<?xml' VersionInfo EncodingDecl '?>'
5 VersionInfo     ::=     'version' Eq (' VersionNum ' | " VersionNum ")
6 VersionNum      ::=     ([a-zA-Z0-9_.:] | '-')+
```

```
7  EncodingDecl    ::=     'encoding' Eq ('"' EncName  '"' |  "'" EncName "'" )
8  EncName         ::=     [A-Za-z] ([A-Za-z0-9._] | '-')*
9  Eq              ::=     '='
10
11 element         ::=     EmptyElemTag | STag content ETag
12 EmptyElemTag    ::=     '<' Name '/>'
13
14 STag            ::=     '<' Name '>'
15 ETag            ::=     '</' Name '>'
16 content         ::=     (element)* | CharData
17
18 CharData        ::=     [^<&]* - ([^<&]* ']]>' [^<&]*)
19 Name            ::=     NameChar
20 NameChar        ::=     NameStartChar | "-" | "." | [0-9] | #xB7 | [#x0300-
21                         #x036F] | [#x203F-#x2040]
22 NameStartChar   ::=     ":" |[A-Z]|"_" | [a-z] | [#xC0-#xD6] | [#xD8-#xF6]
23                         | [#xF8-#x2FF] | [#x370-#x37D] | [#x37F-#x1FFF] |
24                        [#x200C-#x200D] | [#x2070-#x218F] | [#x2C00-#x2FEF] |
25                        [#x3001-#xD7FF] | [#xF900-#xFDCF] | [#xFDF0-#xFFFD] |
26                        [#x10000-#xEFFFF]
```

This part of the system allows XML documents containing unstructured data to be recognised by the low-level encoder. To enable this process, non-structural components are transformed into standard XML elements. For example, attributes are transformed into nested elements in a fixed structure to be later recognised and processed back to their original form. In addition to simple types, the transformation process also requires to transform complex type sequences into suitable forms. Chapter 3 discussed how PO only accepts a subset of complex types derived from ASN.1. However, XML and Schema languages allow documents containing unordered complex types such as `<xs:any>`. Therefore, the transformation process has to deal with the restructuring of complex types such as `sequence` and `sequence-of`.

In summary, the transformation process converts non-structural components and unordered sequences into a highly structured form. This form is constructed on pre-defined nested elements which can be recognised by the transformation process and reverted to the original form. The following sections describe this process in two parts, XML components and structure transformation.

#### 5.3.1.1 XML Components Transformation

Unstructured data is transformed into highly-structured element components nested within the parent node. Attributes, comments, PIs and DTDs are transformed into elements following a general rule: *"Every component is transformed*

*into a sequence of nested elements"*. As highlighted by XMill compressor, converting data into a highly-structured form with the intent of improving compression, increases the size of the original file. However, combining this feature with a schema-informed technique, means that the data will benefit from a well-defined structure without the burden of XML tags redundancy. Although XML transformation appears to drastically increase the size of the original data, the compress format will not be affected. The XML structure will be defined in the schema language. Based on this design, it may appear that the additional nested elements will increase the size of the schema language. This drawback is avoided using repetitive nested elements to define the transformed components. The following code listings provide a clear example on how XML components transformation expands XML documents.

LISTING 5.2: XML Document containing attributes

```
<?xml version="1.0" encoding="UTF-8"?>
<foo bar="Male" baz="1976">foobar</foo>
```

LISTING 5.3: XML Document after transformation process

```
<?xml version="1.0" encoding="UTF-8"?>
<foo>
  <a>
    <a>
      <a>bar</a>
      <v>Male</v>
    </a>
    <a>
      <a>baz</a>
      <v>1976</v>
    </a>
  </a>
  <v>foobar</v>
</foo>
```

In code listing 5.2 and 5.3, the "foo" element contains two attributes, "bar" and "baz" and the value of the element "foobar". In the expanded format, the element "foo" contains an attribute `sequence` "a" which contains a `sequence-of` "a". The attribute `sequence-of` contains "a" and "v" elements which are the name and value of the attribute respectively. All the attributes of the "foo" element are contained inside the attributes sequence. The value of "foo" is placed inside a "v" element after the attribute sequence.

LISTING 5.4: XML Document containing comments

```
<?xml version="1.0" encoding="UTF-8"?>
<foo>
  <!--relevant comment -->
  <bar>bar</bar>
  <!--another comment -->
</foo>
```

LISTING 5.5: XML Document after transformation process

```
<?xml version="1.0" encoding="UTF-8"?>
<foo>
  <c>relevant comment</c>
  <bar>bar</bar>
  <c>another comment</c>
</foo>
```

In code listing 5.4 and 5.5, comments inside the root node are transformed into standard elements of the node. PIs, DTDs and other components follow the same rule applied for comments transformation. The following design is based on the ability to transform XML documents and enable a structured and valid representation of XML. The additional level of nesting for attribute transformation is created to easily generate a valid XML schema based on a `sequence` containing `sequence-of` attributes elements. In addition, the repetitive element tags generated by the transformation process have the dual role of being recognised when performing the reverse process, and achieve better compression size for the generated schema. This is because the automatically generated schema is encoded using a dictionary compression technique.

**Algorithm 1** XML Components Transformation

```
 1: function EXPAND COMPONENTS(Root Node)
 2:     for Current node = Root Node; Current Node; Current Node→Next do
 3:         if Current node = XML ELEMENT then
 4:             for Attr = Node→Properties; NULL != Attr ; Attr→Next do
 5:                 Create Sequence-of Attribute Elements
 6:                 if ∃ Node Value then
 7:                     Create Element Value Node
 8:                 end if
 9:             end for
10:             if Child Element Count > 0 then
11:                 EXPAND COMPONENTS(Current Node)
12:             end if
13:         else if Current node = XML COMMENT then
14:             Create Comment Element
15:         else if Current node = XML PI then
16:             Create PI Element
17:         else if Current node = XML DTD then
18:             Create DTD Element
19:         end if
20:     end for
21:     return Root Node
22: end function
```

Algorithm 1 presents the pseudo code for the XML component transformation. The main function `expand components` analyses each node of the document with different conditional statements depending on the nature of the node. For each element node the algorithms search for attributes and namespaces declarations (which are treated as attributes). In presence of these, the element component is restructured to create a `sequence-of` attribute element with node

value moved to a nested <v> element. The function works recursively to transform elements with nested children. For nodes equal to comments, PIs, or DTDs, a single element is created to store this information.

#### 5.3.1.2 XML Structure Transformation

The structure transformation process applies to unordered sequence of elements. This process can be defined as a structure reordering as each nested element is grouped in a `sequence` or `sequence-of` complex type with its next sibling. Code listing 5.6 and 5.7 provide an example of the sequence reordering applied to an unordered `sequence-of` elements. The structure for nested elements is similar to the one implemented for attributes transformation using <s> tags. Algorithm 2 presents the pseudo code of the sequence transformation process. While parsing current nodes, the main function `expand sequence` triggers `analyse node` to check for unordered sequence of elements. This function compares sibling names increasing the count for repetitions. The return boolean is changed to true when count is greater than 1 and next sibling is not equal to the current node. The return of this function triggers the restructure sequence operation.
More information and examples for the XML components and structure transformation process are provided in Appendix D.

LISTING 5.6: XML Document containing unordered complex type

```
<?xml version="1.0" encoding="UTF-8"?>
<foo>
  <bar>bar</bar>
  <bar>bar</bar>
  <foobar>foobar</foobar>
  <foobar>foobar</foobar>
  <foobar>foobar</foobar>
  <bar>bar</bar>
  <bar>bar</bar>
</foo>
```

LISTING 5.7: XML Document after transformation process

```
<?xml version="1.0" encoding="UTF-8"?>
<foo>
  <s>
    <s>
      <bar>bar</bar>
      <bar>bar</bar>
    </s>
    <s>
      <foobar>foobar</foobar>
      <foobar>foobar</foobar>
      <foobar>foobar</foobar>
    </s>
    <s>
      <bar>bar</bar>
      <bar>bar</bar>
    </s>
  </s>
</foo>
```

**Algorithm 2** XML Sequence Transformation

```
1: function ANALYSE NODE(Node, Count)
2:     boolean = False
3:     if Node→Name = NEXT ELEMENT SIBLING(Node)→Name then
4:         Count + +
5:         ANALYSE NODE(Current Node → Next, Count)
6:     else
7:         if Count > 1 then
8:             boolean = True
9:         end if
10:    end if
11:    return boolean
12: end function
13:
14: function EXPAND SEQUENCE(Root Node)
15:    for Current node = Root Node; Current Node; Current Node→Next do
16:        if Current node = XML ELEMENT then
17:            boolean ← ANALYSE NODE(Current Node, 0)
18:            if boolean → True then
19:                RestructureSequence
20:            end if
21:        end if
22:    end for
23:    return Root Node
24: end function
```

### 5.3.2 Knowledge Extraction

The result of the transformation process is a canon XML compatible with the PO encoder. The second part of the system presented in figure 5.1 is referred as "Knowledge Extraction". During this process, canon XML is analysed in order to extract knowledge required for the subsequent processes. Information regarding complex and simple types is extracted from the highly-structured XML to decide the best compression technique to compress data. With this knowledge the model is capable to automatically generate a schema language to enable a schema-informed compression. Therefore, this process is required in order to identify complex and simple types used in the schema. The knowledge extraction process is involved in detecting basic types such as `integers`, `decimal`, `bit-strings` and others which can be mapped to a lower signed/unsigned integer form. This information is used to construct the schema and improve compression using the PO encoder. For example, detecting IPv4 or hexadecimal data allows the encoder to compress this information using fewer bits compared

to a standard string encoding mechanisms. However, because of its derivation from ASN.1 PER encoder, PO is not able to compress string efficiently, saving only 1 bit for each character of a `xs:string` type. For this reason the model handles string data types using a different encoder based on dictionary compression algorithm such as DEFLATE. Thus, string types are recognised and sequentially stored into a memory buffer which will be compressed using *zlib* library. String types are replaced with an integer identifier which is required to locate the string from the compressed/decompressed buffer. The knowledge extraction process will assign a special data type for these integers in order to restore the string buffer on decompression.
PO EBNF listed in 5.1 only accepts a minimal subset of the original language. The full EBNF allows to freely create elements and miscellaneous components ignoring the risk of validation errors. For example, in a standard XML it is possible to define multiple root elements or embed information outside the root node such as PI, DTD, or comments. As the model needs to keep track of this information, an additional root element is created in order to validate against the XML subset. The root element is removed on decompression and the transformed components are restored to their original format.

### 5.3.3 Schema Generation

A schema is subsequently generated based on knowledge gathered from the canon XML. The schema will define complex and simple types, consisting mainly of data types which can be efficiently handled by PO. Code listing 5.9 provides an example of a schema generated from XML data listed in 5.8.

LISTING 5.8: Example of XML data

```
<?xml version="1.0" encoding="UTF-8"?>
<student>
  <module>FuncProg</module>
  <hours>48</hours>
  <courses>CS</courses>
  <ref>AABBCCDDEE</ref>
  <description>
    Functional programming has
    its roots in lambda calculus
  </description>
</student>
```

LISTING 5.9: Automatically generated PO Schema language

```
<?xml version="1.0" encoding="UTF-8"?>
<_r type="sequence">
  <student type="sequence">
    <module type="integer" variant="constrained" minInclusive="8"
            maxInclusive="8" zlib="1"/>
    <hours type="integer" variant="constrained" minInclusive="48"
           maxInclusive="48"/>
    <courses type="integer" variant="constrained" minInclusive="2"
             maxInclusive="2" zlib="1"/>
    <ref type="hex-string" variant="fixed-length" length="10"/>
    <description type="integer" variant="constrained" minInclusive="68"
                 maxInclusive="68" zlib="1"/>
  </student>
</_r>
```

The schema will be loaded in memory and, depending on the API used, stored locally or concatenated to the zlib string buffer. An important aspect of this process is the lack of validation. PO schema will be used uniquely for compression purposes since the data type information and constraints have already been parsed during the knowledge extraction process. This allows the model to save additional processing time needed to validate XML against the schema. Therefore, the automatically generated schema will act uniquely as a protocol to encode data types into a lower form. In addition, since the schema will not be visible to the end users, it is possible to use an abstract syntax which is closer to the IER of PO. The schema syntax listed in 5.9 can be defined as a concise XML Schema language which can be easily mapped to the XML. In the following HPO schema, string types `module`,`courses` and `description` have been replaced by integers referring to the string buffer. To keep track of these changes, the attribute `zlib=‘‘1’’` is added to the schema definition.

An important aspect of the schema generation process is the ability to define data type constraints to increase the compactness of the compressed format. PO IER does not enforce the user to encode the length of a basic data types if lower and upper boundaries are provided in the schema. The upper and lower bounds are calculated using the data value of elements with identical names. Therefore, these constrains are adjusted to the lowest and greatest value found in the element data. This feature increases the possibility to achieve higher compression as more bits can be saved when both bounds are supplied. Schema generation is aimed at supporting basic types which can be handled more efficiently by encoder. For example, the `ref` element is recognised as a hexadecimal string and therefore compressed using `type="hex-string"` with a fixed length.

### 5.3.4 Character String and Basic Types Separation

An important aspect of the hybrid compression model is the separation between character string and basic types. The canon XML of code listing 5.10 is transformed into an XML containing only basic types and a string buffer as shown in code listing 5.11 and 5.12 respectively. String data for `module`, `courses` and `description` elements, is replaced with an integer which refers to the length of the data. Other elements based on basic types are not transformed. Together with the information provided in the schema, the canon XML formed with basic types is compressed using the PO schema-informed approach. This approach provides the best encoding mechanism to create efficient representations of high-level forms such as `hex-string`, `unix-timestamps` or `enumeration`. The string buffer will be compressed using a dictionary compression technique based on the DEFLATE algorithm. Depending on the use of the API, it is possible to store the schema locally or to compress it together with the string buffer. The specifications of these two compression mechanisms are described in the sections below.

LISTING 5.10: XML Document

```
<?xml version="1.0" encoding="UTF-8"?>
<student>
  <module>FuncProg</module>
  <hours>48</hours>
  <courses>CS</courses>
  <ref>AABBCCDDEE</ref>
  <description>
    Functional programming has
    its roots in lambda calculus
  </description>
</student>
```

LISTING 5.11: Basic Data Types XML Document

```
<?xml version="1.0" encoding="UTF-8"?>
<_r>
  <student>
    <module>8</module>
    <hours>48</hours>
    <courses>2</courses>
    <ref>AABBCCDDEE</ref>
    <description>68</description>
  </student>
</_r>
```

LISTING 5.12: String Buffer

```
FuncProgCS
    Functional programming has
    its roots in lambda calculus
```

#### 5.3.4.1 String Data Types compression

String data type compression is aimed at improving the encoding size of the string buffer. This buffer is generated from a stream of elements containing

string data that cannot be efficiently mapped to a lower integer form. The *zlib* library, which implements the DEFLATE algorithm, is used to compress string data stored in a memory buffer with size equal to the total amount of string data types of the XML. With this approach, it is possible to compress string data types of XML, including those that can be represented with enumeration. A dictionary compression algorithms is be able to efficiently represent multiple occurrences of string data which would otherwise require more complexity on the Knowledge Extraction process and additional information stored in the schema. For this reason, the current version of the hybrid model does not detect enumeration types.

Elements with string values are grouped into a memory buffer and replaced with an integer referring to the length of the value. This integer is needed in order to restore the string value back to its element on decompression. A similar technique is implemented in data serialisation for programming languages data structures. After data values are grouped together sequentially, the length is used to restore data on de-serialisation. During decompression, this technique is triggered by the zlib schema option of integer elements. The element with the first occurrence of zlib option is replaced with *n*-characters of decompressed buffer. Subsequence occurrences will add previous lengths and restore string elements to their original values. Table 5.2 provides a visual example of the string data type compression and decompression for the string buffer of XML data previously analysed in code listing 5.10.

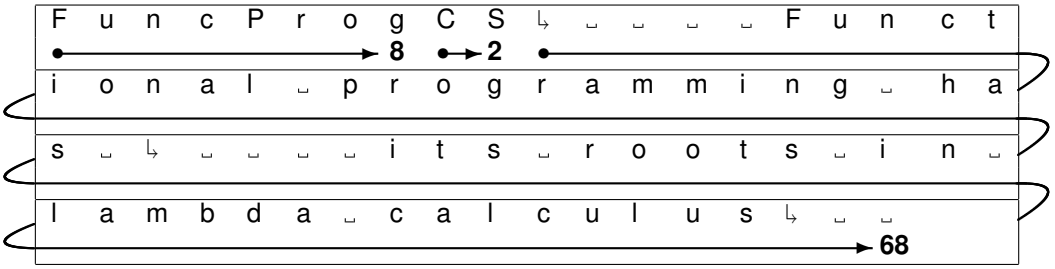


TABLE 5.2: String Buffer Data Serialisation

#### 5.3.4.2 Basic Data Types Compression

Basic data types are compressed using the schema-informed approach of PO. Schema-informed techniques provide the best compression size for many types of XML documents. This model applies PO compression for canon XML data

composed purely of basic types which are compressed efficiently due to the fixed length encoder and the schema-informed technique. The use PO instead of EXI was based on the results performed in Chapter 4. The high software complexity and the absence of a native language implementation do not allow an easy integration of EXI with this model. In addition, PO is highly portable and has proven efficient in a number of projects. Libxml2 allows the developer to parse XML and generate schemas more easily by integrating PO implementation with the hybrid model. In conclusion, basic data types compression is based on the IER of PO described in Chapter 3.

### 5.3.5 Compressed Format

The binary format of the hybrid model is based on a header followed by the compressed payloads. Figure 5.1 of section 5.3 describes the binary format as a header followed by the basic data types and complex types compressed format of PO and the string buffers compressed using DEFLATE. This format is created when the model operates on a hybrid mode. An important aspect of the model is the ability to compress data dynamically based on the data types presented. This allows the system to fully apply one encoding technique over another. For example, if the data types analysed during the knowledge extraction process consists of basic data types, the hybrid model will encode the document using the fixed-length encoder of PO. Contrary, when presented with string data, the entire document will be compressed using the DEFLATE algorithm. This feature allows the hybrid model to operate dynamically creating different binary formats depending on the XML data presented.
A header is used to provide the necessary information to decompress the binary format. In **hybrid** mode the header is followed by the PO format and the compressed string buffer and schema.

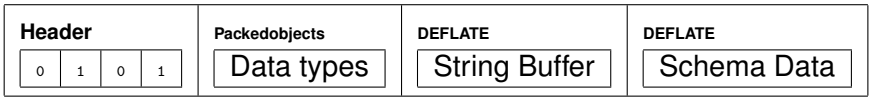


TABLE 5.3: Hybrid Mode Binary Format

In **pure mode** the header will simply be followed by text compressed data.

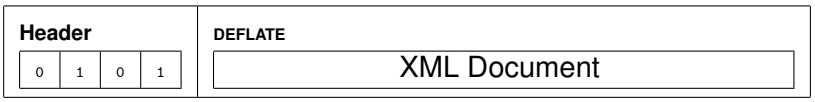


TABLE 5.4: Pure Mode Binary Format

Depending on the XML data types, the model will compress the XML file using the hybrid or pure mode.

### 5.3.6 Decompression process

The decompression process is less computational intensive and therefore faster compared to compression. The knowledge extraction and schema generation processes are not required. All the information needed to decode XML is stored in the schema and the string buffer. The header of the compressed format will inform whether the XML document was compressed using a hybrid approach or one particular compression over another. Figure 5.1, discussed in section 5.3, illustrates the output of the hybrid compression model divided into basic and character string data types compressed using PO and *zlib* respectively. The string buffer is decompressed to return the schema, in case it was not saved locally, and the values of the string buffer. PO makes use of the schema to decompress its buffer and return the canonical XML. During this process, elements containing zlib option are restored with their original value found in the string buffer. The header will inform if the document transformation process was performed on the XML. Based on this option, the canonical XML will be processed to transform structured elements back to their original components which are restored using element patterns to detect attributes, comments and other transformation processes.

A key feature of the hybrid model is the ability to recognise non-structured data and compressed it using features described above. After decompression, all information which was recognised during compression is restored back to its original format. Compared to other models, this approach does not operate on the XML documents but creates a memory representation based on what the system was able to detect. Non-structured components are transformed into PO format while others such as newlines and white spaces outside elements are ignored. It is possible to store this information in additional structured elements to return the precise format as the XML document was compressed.

However, this data does not affect the XML and can be ignored in most cases. On decompression, standard newline and white space characters are restored according to the standard specification of XML.

## 5.4 System Execution

As described in the previous section, the hybrid model is based on two compression techniques. PO and the DEFLATE algorithm of the *zlib* library are the two back-end compressors to apply a fixed and variable length encoding techniques. The hybrid model is based on the separation between basic and character string data types in order to achieve the highest level of compression. Figure 5.2 presents the system execution cycle for compressing XML documents. The initial analysis of the XML document is found during the first process of the hybrid model. During the document transformation, it is possible to opt out from the hybrid technique and apply a pure compression mode. This mode can be triggered in case data types are not suitable for the fixed length encoder or for non-valid XML structures.

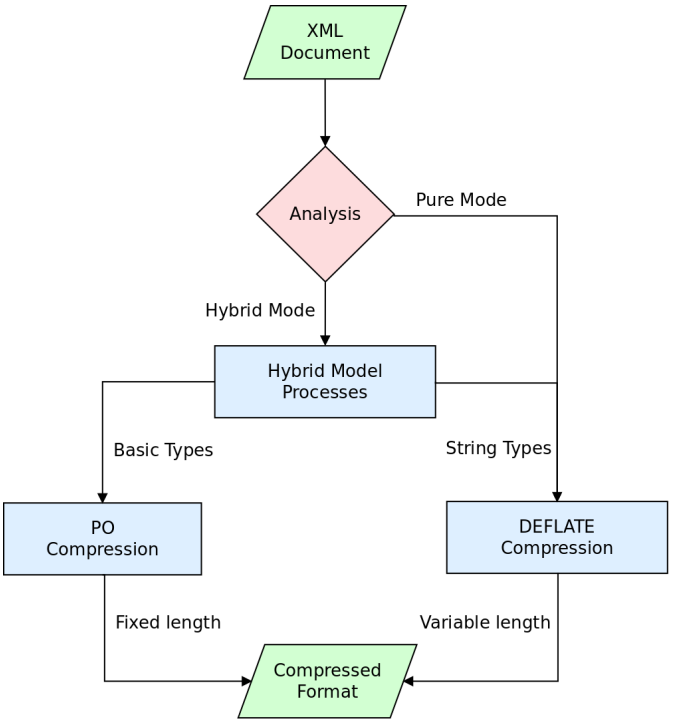


FIGURE 5.2: Hybrid Model System Execution

From a high level perspective, the hybrid model is based on two length coding techniques, fixed and variable. As this research focuses on the benefits of fixed length coding techniques to general-purpose compressors, it is vital for the hybrid model to apply both techniques in a hybrid mode. These techniques can be applied fully or partially, depending on the data types of the XML. The pure mode is enabled in case of unsuitable data types or XML structures in order to improve the performance and avoid possible encoding errors.
The idea behind the hybrid model is to encode a number of data types using the fixed-length encoder. As shown in Chapter 4, fixed length encoders, together with a schema-informed technique are able to achieve higher level of compression compared to standard general-purpose compressors. Due to the transparent schema-informed technique of the hybrid model, it is possible to apply a high level of compression for data types compressed using the fixed length encoder.

## 5.5 Code Optimisation

Code optimisation techniques have been implemented during the design and development of the hybrid model to improve the overall performance. The hybrid model can be divided into two parts the front-end and the back-end. The front-end is the part of the system in charge of transforming the markup language into a lower form which can be then passed to the fixed and variable length encoders. These represent the back-end of the system. This separation is based on the ability of the hybrid model to adapt to other markup languages. As described in Chapter 1, this work focuses on the compression of XML as the standard example of markup language. However, the research aim is to demonstrate the ability of a fixed length encoder to enhance general-purpose compressors for markup languages. Therefore, this separation is important to allow the hybrid model to be applied to other markup languages which can be validated by a data definition language.

The following sections describe some of the optimisation features implemented to improve the performance of the hybrid model. The performance optimisation is aimed at improving the execution speed, keeping memory and CPU usage low. The requirements previously described in section 5.2 focus on improving

the efficiency of the hybrid model which is the amount compression applicable to an XML file.

### 5.5.1 Front-end

Both the front-end and back-end of the hybrid model require a high level of performance. The back-end encoders have been selected using the information collected from other work conducted in this area and the results of Chapter 4. The front-end is divided into several processes as illustrated in the previous figure 5.1. Document transformation and schema generation are the most intensive tasks. The first task transforms XML documents into highly structured formats compatible with the fixed-length encoder of PO. The second task instead, gathers knowledge from the XML file in order to develop a domain-specific schema. Both tasks have been developed with the additional purpose of improving performance.

The first implementation choice for the document transformation process is XSLT. By defining the transformation rules in an XSLT script, the system requires an additional processor to parse the script and transform the document. In order to improve the performance of the hybrid model, the system implements a native implementation written using a dependency library of XSLT. *Libxml2* is used as the API to transform XML components into highly-structured formats as described in section 5.3.1. This implementation allows the system to avoid having to parse an additional script during run-time and perform better using only a subset of the transformation library.

The schema generation process is also considered for performance and efficiency optimisation. During this process the entire XML document is parsed multiple times, during which an internal representation of the schema is constructed using hash tables. This memory representation is then used to construct a domain-specific schema language. Since this validation language is only used for mapping data types to the IER of PO, the standard XML schema language was not considered. This schema language allows the hybrid model to construct a highly efficient format using less space than it would be required by the standard XML schema. For debugging purposes the current schema provide full naming of the attributes required as shown in code listing 5.9. However,

a more optimised version can be constructed using code-words to improve its compression.

### 5.5.2 Back-end

The back-end of the hybrid model is based on a variable and fixed-length encoder, *zlib* and PO respectively. These two tools have been selected based on their results achieved in the experiments of Chapter 4 and the low amount of dependencies needed. As shown in section 4.4.2, *zlib* and PO are the fastest implementations when compressing highly structured XML files. The low number of dependencies needed by both encoders allows the hybrid model to be build using only *libxml2* which is required for the front-end processes. This property allows the hybrid model to be highly portable to various platforms which are able to satisfy the dependencies requirement.

## 5.6 System Requirements Support

Based on the documentation of compression stages it is possible to describe how the proposed model is able to meet the requirements defined in the section 5.2. The model was designed based on the advantages of each tool combined together in an elegant approach. The first requirement R1 aims at supporting a wider range of XML data sets. So far most the tools analysed are aimed at a specific area of XML compression. The compression size efficiency of these tools depends on the size and nature of XML. This model was designed to support a wider range of data sets ranging from small to large, document to data-centric, regular to irregular XML. Thanks to its ability to adjust an XML document to a subset of PO, the transformation process is the major part of the model which is able to satisfy the requirement of R1.
An important feature of the hybrid model is the ability to manage schema uninformed compression efficiently defined in requirement R2. Due to the knowledge extraction and the schema generation processes, the hybrid model is able to perform a transparent schema-informed compression. This novel schema uninformed compression implements an informed approach which is not visible to the end user. This technique enables the model to benefit from a defined

structure and data types, which are vital components for the IER back-end compressor.

The third requirement R3 is based on the concept of separating structure from data. Chapter 4 analysed how compressors based on this principle are able to outperform other tools with homomorphic properties. In order to meet the requirements of R3, PO and zlib have been integrated as back-end compressors. Using these compressors it was possible to benefit from the properties of the schema informed compression of PO and the dictionary compression of zlib. PO provides the ability to separate structure from data using a schema as reference. In addition, the compression model will separate data between basic and character string types. Basic and complex types will be compressed using PO while character string types using zlib. With this approach the best compression techniques can be used for these two categories of data types in order to achieve better results.

## 5.7 A Motivating Example

This section provides a motivating example to test the efficiency of the hybrid model. A more detailed evaluation of the hybrid model will be provided in the next chapter. XML file `lineitem.xml` is used from corpus (Miklau, 2014), a popular document generally used for testing XML compressors. A snippet of XML is provided in code listing 5.13 in order to understand the document. An analysis of the XML informs us on the data-centric highly-structured properties of `lineitem.xml`. The XML format is compatible with PO subset of XML and therefore does not require the document transformation process. From a more advanced analysis the document can be identified as a `sequence-of` element <T> containing a `sequence` of mixed ordered elements. These are based on a range of character string and basic types such as `integers`, `decimal`, `date`, and `string`. In addition, element values can be categorised as random data with a specific pattern which can be described in a schema language.

LISTING 5.13: Snippets of lineitem.xml document

```
<T>
  <L_ORDERKEY>22496</L_ORDERKEY>
  <L_PARTKEY>1913</L_PARTKEY>
  <L_SUPPKEY>2</L_SUPPKEY>
  <L_LINENUMBER>1</L_LINENUMBER>
  <L_QUANTITY>17</L_QUANTITY>
```

```
  <L_EXTENDEDPRICE>30853.47</L_EXTENDEDPRICE>
  <L_DISCOUNT>0.03</L_DISCOUNT>
  <L_TAX>0.01</L_TAX>
  <L_RETURNFLAG>A</L_RETURNFLAG>
  <L_LINESTATUS>F</L_LINESTATUS>
  <L_SHIPDATE>1994-08-24</L_SHIPDATE>
  <L_COMMITDATE>1994-11-19</L_COMMITDATE>
  <L_RECEIPTDATE>1994-09-10</L_RECEIPTDATE>
  <L_SHIPINSTRUCT>DELIVER IN PERSON</L_SHIPINSTRUCT>
  <L_SHIPMODE>SHIP</L_SHIPMODE>
  <L_COMMENT>carefully pending i</L_COMMENT>
</T>
```

The compression efficiency of this model was tested with documents such as `lineitem.xml`. From the compression model it is possible to extract the generated schema shown in code listing 5.14. The schema file demonstrates the redundancy of XML structure and the composition of the document. A current version of the model is able to identify various data types such as `integers`, `currency` and `boolean`. String types are grouped into the string buffer and compressed using DEFLATE algorithm. The following schema shows how information presented in a date format such as `1994-08-24` is sent to the string buffer. The model is not able to identify this date format and treats it as a string.

LISTING 5.14: Automatically generated schema

```
<?xml version="1.0" encoding="UTF-8"?>
<_r type="sequence">
 <table type="sequence-of" items="1" minOccurs="60175" maxOccurs="60175">
  <T type="sequence-optional" items="16">
    <L_ORDERKEY type="integer" variant="constrained" minInclusive="1"
                    maxInclusive="60000"/>
    <L_PARTKEY type="integer" variant="constrained" minInclusive="1"
                    maxInclusive="2000"/>
    <L_SUPPKEY type="integer" variant="constrained" minInclusive="1"
                    maxInclusive="100"/>
    <L_LINENUMBER type="integer" variant="constrained" minInclusive="1"
                    maxInclusive="7"/>
    <L_QUANTITY type="integer" variant="constrained" minInclusive="1"
                    maxInclusive="50"/>
    <L_EXTENDEDPRICE type="currency"/>
    <L_DISCOUNT type="currency"/>
    <L_TAX type="currency"/>
    <L_RETURNFLAG type="boolean" zlib="1"/>
    <L_LINESTATUS type="boolean" zlib="1"/>
    <L_SHIPDATE type="integer" variant="constrained" minInclusive="10"
                    maxInclusive="10" zlib="1"/>
    <L_COMMITDATE type="integer" variant="constrained" minInclusive="10"
                    maxInclusive="10" zlib="1"/>
    <L_RECEIPTDATE type="integer" variant="constrained" minInclusive="10"
                    maxInclusive="10" zlib="1"/>
    <L_SHIPINSTRUCT type="integer" variant="constrained" minInclusive="4"
                    maxInclusive="17" zlib="1"/>
    <L_SHIPMODE type="integer" variant="constrained" minInclusive="3"
                    maxInclusive="7" zlib="1"/>
    <L_COMMENT type="integer" variant="constrained" minInclusive="10"
                    maxInclusive="43" zlib="1"/>
  </T>
```

```
 </table>
</_r>
```

The compression size difference was tested using the most efficient tools analysed in Chapter 4. The performance of the hybrid model was tested against the schema-uninformed approach of EXI and GZIP. Various options are available for the EXI implementation. This example focused on a standard EXI encoding without the additional DEFLATE compression applied to the aligned bit stream. The results listed below show the compression size difference of achieved by each tool. From the results of the hybrid model it is possible to analyse the amount of data that has been passed to the string buffer. From the 34.3 MB of `lineitem.xml`, the amount of raw string data sent to the buffer is only 4.5 MB. Together with the generated schema, the string buffer is compressed to a 1.0 MB format. The rest of the XML is composed with redundant element tags, ignorable characters (newlines and white spaces), basic data types and integers pointing to the string buffer. From this canon XML, the PO encoder creates a compressed format of merely 0.9 MB. Together with the compressed string buffer a total size of 1.9 MB was achieved. The compression results for selected tools are listed below.

**XML: 34.3** MB **EXI: 4.6** MB **GZIP: 2.9** MB **HPO: 1.9** MB

## 5.8 Compression Models Comparison

The hybrid model was designed based on the properties and features found in a number of XML compression tools. As described in section 5.2, the requirements on which HPO is based on, can be found in a number of tools. These tools can share one to two of the system requirements of HPO. For example, EXI and XMill are both able to support a wider range of XML data sets and apply a semantic compression. In addition, both tools apply fixed length encoding through the use of built-in or atomic data types. For example, XMill applies a fixed length encoding for atomic (basic) data types such as *integer* values. The main difference is the use of users defined data types knowledge to increase compression by separating element values into semantic containers. EXI presents a similar design and level of compression to HPO. The following section highlights the differences between these two models.

### 5.8.1 EXI vs. HPO

As discussed in Chapter 1, EXI implements a fixed length encoding technique. This is the most popular and a standard format recommended by the W3C consortium. This format allows both a fixed and variable length encoding technique to achieve a highly efficient format for storage and transmission purposes. The hybrid model is based on the same technique. Using both fixed and variable length coding techniques, HPO has shown a substantial level of compression mainly for synthetic data sets. Although both tools implement these length coding techniques, a major difference can be highlighted between the two tools.
EXI uses both encoding techniques sequentially. Using its own built-in grammar and data type representation, EXI transforms XML data into a coded stream. This stream is generated using a string table for repeating data and a fixed length encoding technique for some of the data types recognised by EXI. A variable length compression is applied after the EXI stream is encoded. After combining smaller channels together and arranging data semantically, EXI applies a DEFLATE compression to the newly aligned stream body. The result is a compressed encoded stream. Therefore, the two length encoding techniques are applied sequentially.
HPO implements both encoding techniques independently. After converting the XML document into a lower format, HPO extracts knowledge from the XML file to construct an internal schema. This domain-specific definition language is used to provide the encoding rules for the fixed length encoding/decoding processes. Therefore, these processes require a schema for the low-level encoder rather than for validation purposes. XML data is subsequently separated into basic and character string types depending on the data types recognised by the schema generation process. This process is the main difference between HPO and EXI. This separation allows HPO to control the amount of data sent to each encoder and improve compression size when compressing high level data types.
An important feature introduced by HPO is the introduction of high-level basic data types encoded using PO. As shown in the results, this feature allows data sets based on these types to achieve high level of compression. Compared to EXI, HPO allows encoding high level data types such as IPv4, Enumeration, Unix-Timestamp and hexadecimal without the need of an external or user defined schema language. In addition, due to the separation between front-end

and back-end, the hybrid model can easily include support for other markup languages by extending the front-end of the system.

## 5.9 Applicability and Limitations

The application of the hybrid model can be found in a number of fields. It is possible to apply the model to a number of scenarios due to its ability to support a wide range of XML data sets. Based on the performance evaluation of different tools analysed in previous chapters, the application of a specific technique over another highly depends on the XML data presented. The nature of XML used for network messaging is different from XML documents representing databases. Categorising XML documents based on size, structure, data types and validity provides information on which compression technique is best to apply.

### 5.9.1 Document Support

The hybrid model is able to handle structural and textual documents without concerns on the validity. Schema-informed techniques are not able to apply compression when presented with XML documents that violate the schema specifications. In order to overcome this issue, EXI introduced a strict and non-strict approach in order to compress data efficiently even during schema violations. However, most XML parsers are able to recognise data presented in irregular documents. The model makes use of this ability to convert an irregular document to highly-structured XML which can be efficiently compressed using a transparent schema-informed compression. This feature is possible due to transformation process which is aimed specifically at converting general documents to the PO subset of XML.

### 5.9.2 Dynamic Application

Knowledge of XML data is a key feature to achieve better compression ratios. The previous section analysed an XML document with regular structural properties. This specific document did not require a transformation process since it

was already presented in a PO-compatible format. The knowledge extraction process provided the information to generate a schema and recognise few basic data types compressed using PO. In a similar scenario it is possible to avoid triggering the transformation process and focus on the subsequent parts of the model. This approach can be described as a dynamic application of the hybrid model. This feature allows the user to process the XML using only the required stages of the hybrid model. For example, it is not needed to generate a schema if the XML document is linked to an external validation language. Therefore, a key feature of the hybrid model is the ability to apply different processes depending on the data presented. In an XML database scenario it is possible to save the schema locally and compress several documents with identical structure but different data. Limitations of this technique are linked to the development of simple and complex types which over time can violate encoding rules in case of enumeration or `sequence-of` types. Similar scenario is found when compressing XHTML documents. A powerful feature of the hybrid compression model is its ability to be extended to other markup languages which can be defined by a schema language. The front-end part of the system can be expanded enabling the ability to compress other markup languages.

### 5.9.3 Hybrid and Pure Mode

Based on the analysis of XML data sets gathered from XML compression and data management research areas, this study is aware of the structure and components used to construct XML documents. Research conducted in these fields is aimed specifically at providing developers with information on the nature of XML documents. This knowledge is crucial for understanding which are the most recurring and popular components of XML documents. Current research has shown how the majority of XML documents heavily rely on attributes, followed by comments, DTDs and PIs. This information can be used to design the transformation model to support the most common components of XML. For example, the current implementation is aimed at supporting attributes, comments and sequence transformation, however, additional components can be supported. The effort spent on supporting every possible feature of XML can increase software complexity and limit the performance of the model. Similar issues arise in the presence of XML documents mainly composed with structural

or textual data. For example, in presence of small structural XML documents with very minimal textual data, one particular compression techniques can suffice. A hybrid model can add overhead due to the poor efficiency of dictionary compressors for small string data types. However, large textual XML documents with very minimal structure and basic types can achieve a better compression if a dictionary compression is applied. The use of a hybrid model would require additional complexity and processing time to transform the textual document and generate the schema.

### 5.9.4 Near-lossless Compression

The concept of near-lossless XML compression can be found in a number of research as discussed in Chapter 2. DTDPPM is the first tool that has introduced ignorable white space stripping for schema-informed compression. White spaces and newlines characters are common in many XML documents and standards to improve the readability of XML. While these characters can be defined *ignorable* for structural documents, some application may heavily rely on the layout of the XML with particular attention to textual documents. This category of XML may hold data outside element tags, increasing the difficulty in distinguish between ignorable and meaningful white spaces. Therefore, near-lossless techniques can be applied in order to recreate these characters. Practically, some of the ignorable white spaces or newlines may be lost during compression cycles, even if this data does not hold any information and it is not relevant to the presentation of XML.

## 5.10 Conclusion

This chapter presented a hybrid compression model capable of improving the compactness of the compressed format for a wider range of data sets. The hybrid model is based on the key features that allow XML compressors analysed in previous chapters to achieve better compression size. These features are based on evidence of the performance of compression tools for large textual data sets available in most performance comparisons and small structure data sets analysed in Chapter 4.

The main objective is to improve the compactness of the compressed format using tools capable of achieving substantial results for specific data sets. Two tools have been selected as back-end compressors to provide the best performance and compression size. In Chapter 4, it was noticed how software complexity is directly proportional to the computation time required to compress XML. For this reason tools with the lowest complexity and the best performance have been selected as back-end compressors of the hybrid model. In addition, these tools have been selected to manage one particular type of XML document. The transparent schema-informed approach of PO is capable of achieving the optimal performance and compression size when compressing highly-structured XML documents. This compression also includes PO complex types encoding. The DEFLATE algorithm of zlib, instead, implements dictionary compression with optimal results for medium to large textual data.

In summary, the hybrid model described in this chapter is able to compress XML documents using both fixed and variable length encoding techniques. The ability of this tool to decide the types of data to be compressed using a specific technique is essential in order to investigate how fixed length encoder can enhance general-purpose compressors.

# Chapter 6

# Schema-uninformed compression comparison

This chapter compares the efficiency and performance of the hybrid model with a number of XML-conscious and general-purpose compressors. The first section describes the methodology and environment used to produce replicable experiments. This part introduces the compression tools used to compress the XML corpus, the system resources where the experiments were performed and the XML corpus used. An introduction to the data sets used to test the efficiency of the hybrid model is provided. These are divided into synthetic and real XML. The second section provides the results of the experiments. These results are discussed for synthetic and real XML data sets, providing comparison analysis to illustrate the compression difference between individual data sets. The third section discusses the compression difference between synthetic and real XML data sets. Subsequently, this section provides an explanation to this compression difference and introduces the concept of synthetic data types. An analysis is conducted for those real XML data sets which have not shown a significant level of compression with the hybrid model. The data types found are analysed to provide future directions. The analysis section also discusses the performance evaluation. Here, the different types of processes of the hybrid model are explained to illustrate the ideal scenario in which it can perform.

## 6.1 Experimental Methodology

The performance of the proposed model is empirically compared with XML-conscious and general-purpose compressors to demonstrate the effectiveness of the hybrid model. The hybrid model can be extended to other markup languages that benefit from a descriptive schema definition. However, current implementation only supports XML. For this reason, the tests were performed against the best XML-conscious compressor evaluated in Chapter 4. In addition, the experiments present comparison results for general-purpose text compressors. These compressors are used as a standard to determine the performance and efficiency of proposed tools against well known compression algorithms. The experiments consist of a series of compression tools presented with different types of XML data. In order to demonstrate the behaviour of the hybrid system, the experiments were performed on synthetic and real XML data sets. For future referencing the hybrid model will be called HPO. The following sections provide a description of the experimental environment including tools and data sets used to achieve the results.

### 6.1.1 Compression tools

The performance and efficiency of the hybrid model have been tested against EXI, GZIP and 7ZIP. EXI Processor (Garrett, 2012) is a command-line tool used to encode XML to binary EXI and decode EXI to text XML. EXIProcessor is written in Java and uses the open source EXI library EXIficient (Peintner, 2012) developed by Daniel Peintner of Siemens AG. EXIficient is one of the suggested implementations of EXI version 1.0 listed in the format specifications. The experiments were performed using the highest level of compression achievable by each tool. EXI options allow a more compact binary representation of XML data using strict encoding options and additional compression for aligned binary representations. The experiments were performed using EXI *strict* and *compression* options enabled.

In addition to the XML-conscious compression of EXI, HPO has been tested against GZIP and 7ZIP. The application of these general-purpose compressor spans from database to network compression. These tools have been included to demonstrate the efficiency of HPO compared to standard general-purpose

| Tool | Command-line | Options |
|---|---|---|
| EXI | java -jar ExiProcessor.jar -compression -xml_in file.xml -exi_out file.exi | -compression: Increases compactness using additional computational resources |
| GZIP | gzip -c -9 file.xml > file.xml.gz | -c: Write output on standard output<br>-9: Indicates best compression method |
| 7ZIP | 7zr a -mx9 file.xml.7z file.xml | *a*: Create an archive<br>-mx=9: level of compression=9 (Ultra) |

TABLE 6.1: Command-line Tool Options

compression algorithms. Options to enable the highest level of compression were used for both GZIP and 7ZIP compressors. Table 6.1 presents the list of command-line tools used to compress the data sets.

### 6.1.2 System Resources

The experiments were performed on a Ubuntu 12.04 (precise) 32-bit machine equipped with an i7 CPU @ 2.80GHz x 4, 2 x 2048 MB DDR3 @ 1067 MHz and SATA 2.6/ATA-8 500GB @ 7200 rpm. Although the hybrid model is designed to perform on low-powered constrained devices, the system architecture does not affect the results of the experiments in terms of compression efficiency. The results of a performance evaluation, instead, would present a similar patterns in terms of compression speed. The experiments would also require the same amount of memory. Additional information about the system specifications can be found in table B.2 of Appendix C. Data sets were stored on local disk and with local references to DTD/Schema files.

### 6.1.3 XML Corpus

Constructing an XML corpus capable of demonstrating the application of the hybrid model is a challenging task. To demonstrate its efficiency, the XML corpus must contain a set of data relevant to its use case. The model specifications described in the previous chapter illustrate how the hybrid model is capable of achieving better compression for a specific set of data types. Although most XML structures adhere to the schema specification, element data is rarely presented in a consistent data type format. For example, an element of a `sequence-of` node with a series of decimal data types would break the

recognition process if data is not presented in the decimal format. In these circumstances, the hybrid model is not able to apply its own data types resulting in a less efficient encoding scheme. Another example can be found for the `date` data type analysed during the compression of `lineitem.xml` of Chapter 5. The hybrid model does not recognise the specific date type format, that is encoded in the string buffer. Therefore, the corpus was specifically selected based on data types of XML documents.

The corpus is based on a series of data sets with different structures and data types. In addition, synthetic XML data sets have been constructed to demonstrate the full potential of the hybrid model. The corpus can be divided into two major categories listed below.

**Synthetic XML Data**

Synthetic XML data sets have been created to demonstrate the full potential of the hybrid model when presented with ideal data types. Current implementation supports a limited set of basic data types which can be found in few real XML files. However, using synthetic XML data, it is possible to create data types consistent with the use case of the hybrid model. The data sets are based on XML containing single and mixed data types of randomly generated data. For each type of XML data set, a total number of 500 files were created with size ranging from 5KB to 5MB. A total of 8 different types of XML data are evaluated totalling a number of 4000 files.

**Real XML Data**

The term *real* defines a set of XML data manually collected from government sites, open-source repositories, database representations, and website exports. The data set includes well-known XML files such as `lineitem.xml` and `supplier.xml` from corpus (Miklau, 2014) which have been tested in a number of experimental evaluation. A total number of 16 files ranging from 5KB to 30MB in size have been collected and made publicly available online[1].

Real XML data sets are based on a richer variety of data types which are not always recognised by the hybrid compression model. However, a good level of compression can be achieved with a minimal understanding of these types. The structures and types of the data sets will demonstrate

[1] https://mobile.uwl.ac.uk/xml/corpus/

the potential of the hybrid compression to apply semantic knowledge and improve the compression size. This compression can be achieved by balancing the fixed and variable length encoders of the hybrid model. In addition, real XML data will be used in order to demonstrate the potential to generate *synthetic* data types.

## 6.2 Experimental Evaluation

The following sections illustrate the experimental results comparing the hybrid model to EXI, GZIP and 7ZIP. These sections focus on the compression efficiency in order to demonstrate the performance of the hybrid model when compressing XML data. A number of graphs will highlight the compactness of the compressed format for files ranging from 5KB to 40MB. The corpus consists of 5GB of XML files divided into synthetic and real XML data. Compression size graphs demonstrate the efficiency of the hybrid model compared to XML-conscious and general-purpose compressors. The comparison is divided into two categories, synthetic and real XML data sets.

### 6.2.1 Synthetic XML Data

The following figures illustrate the compression efficiency when compressing synthetic XML data. Figures 6.1 to 6.4 show the compression results of HPO compared to other tools. Each figure is divided into four different graphs illustrating the efficiency for specific XML data types and structures. The $x$-axis represents the size of XML files while the $y$-axis illustrates the size of the compressed format achieved by each tool. Both axes are plotted in a logarithmic scale. The term “fixed” and “random” is used to define synthetic XML data set with a single and multiple randomly generated data types respectively.
Graphs where compression size lines cross or present similar results are highlighted with a sub-diagram for a better visualisation of the data.

#### 6.2.1.1 Fixed Data Types

Figure 6.1 and 6.2 presents the result for data types such as `integer`, `decimal`, `dateTime` and `string` for file ranging from 5KB to 2.5MB in size. The data sets used for these experiments are based on XML files containing a single, randomly generated, basic or string data type within a fixed structure. The compression results of these data sets illustrate the efficiency of HPO when compressing a unique data type. In this scenario, only a single encoder of HPO is used to compress XML files. For example, for an XML file containing only integer data types, HPO will apply the fixed length encoding system for all elements. Any data types which are not recognised by HPO, or based on string types, will be compressed using the variable length encoding technique of the DEFLATE algorithm.

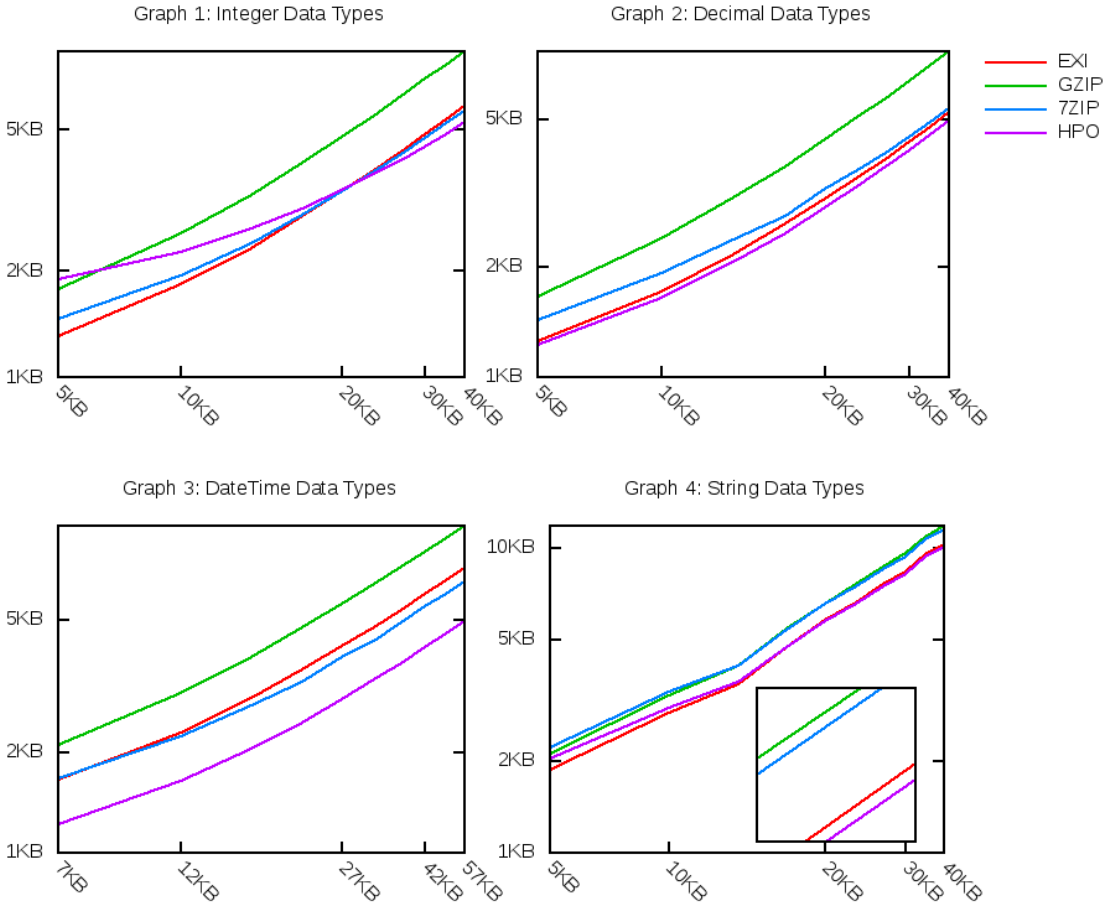


FIGURE 6.1: Fixed Synthetic Data Types - 5KB-50KB

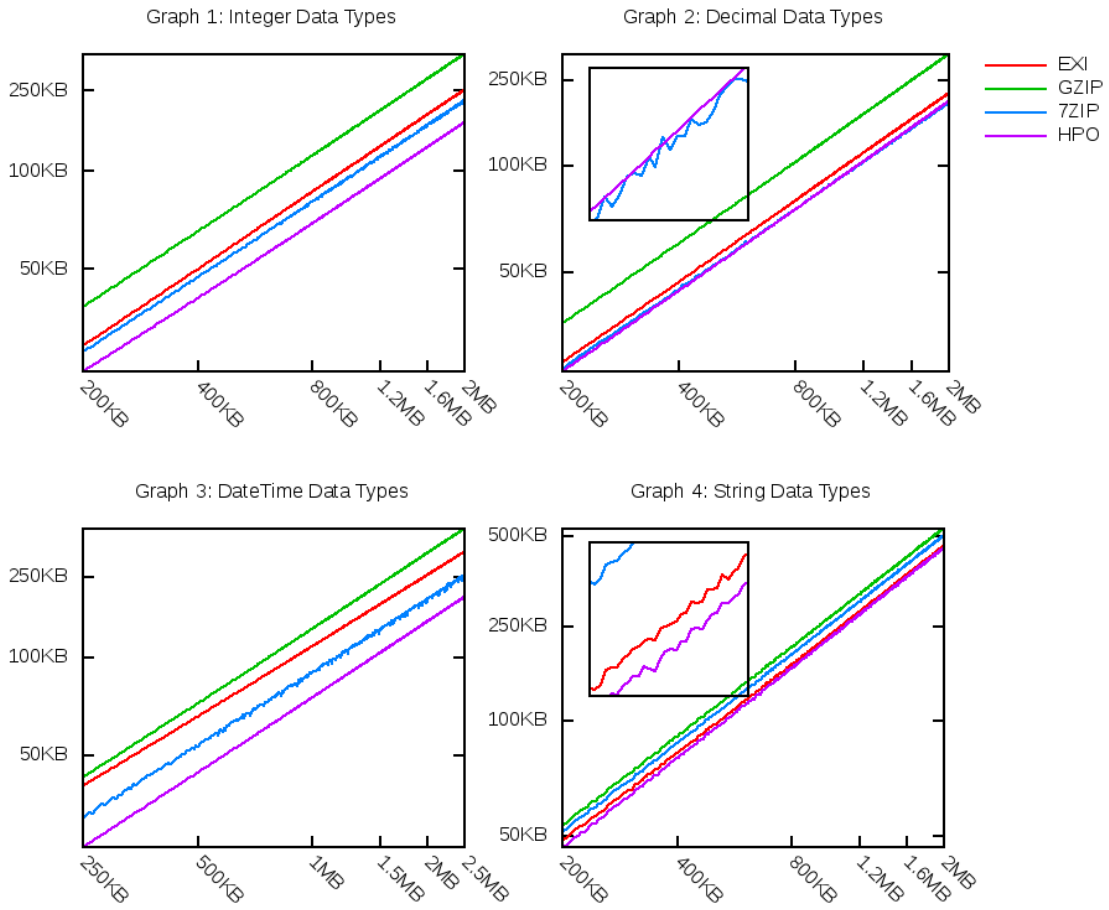


FIGURE 6.2: Fixed Synthetic Data Types - 200KB-2.5MB

Data sets of graphs 1, 2 and 3 of figure 6.1 and 6.2 are recognised as XML files containing a sequence of basic types only and compressed efficiently by HPO using the fixed length encoder. Data types of graph 4, instead, are based on randomly generated strings with lengths ranging from 5 to 30 characters. HPO recognises this data set as XML containing string types only and compresses it using the variable length encoder. Graphs of figures 6.1 and 6.2 are divided into small and medium sized files in order to highlight the compression difference for small XML files. For example, in figure 6.1, a less efficient compression is found for files of small size based on fixed integer and string data types. For fixed integer data types, the compression difference between the EXI and HPO starts at 10% in favour of EXI. In graph 1 of figure 6.1, HPO compression is more efficient for files starting from 20KB in size. Although HPO is able to map integer data types to their lowest encoding scheme, the compressed schema increases the overall size of the HPO compressed format. In addition, the XML files of

these data sets are based on a sequence of basic data types, encoded using fixed length encoder. Therefore, the string buffer is empty and the DEFLATE algorithm is only used to compress the schema file. Variable length encoders are not efficient at handling small sized files, resulting in a worse compression size for small XML files in general. Analogous results are found for the string data type, graph 4, of figure 6.1. Here, the compression difference between EXI and HPO is less noticeable. The difference in compression efficiency is around 3% in favour of EXI for XML file of 5KB in size, and around 0.5%, 200KB, in favour of HPO for the XML file of size 40KB of the graph.

EXI demonstrates a better encoding size for small XML files of graphs 1 and 4 of figure 6.1. In graph 3, HPO demonstrates a significant level of compression beyond those achieved by other compressors. HPO achieves an additional overall level of compression of around 5%, compared to EXI and 10% compared to GZIP. An analysis of the XML files of this data set illustrates the use of the RFC 3339 Timestamps described in section 3.1.8. Data presented in formats such as `1990-12-31T23:59:60Z` is encoded as a string data type by EXI and encoded within similar compressed formats to 7ZIP. However, semantic knowledge allows HPO to apply a fixed length encoding mechanism, transforming the high-level Timestamps format to efficient unsigned integer forms. These high-level PO derived data types discussed in Chapter 3 are encoded into efficient formats. The ability of mapping similar data types to an efficient format will provide better results compared to variable length encoders as illustrated in figure 6.1 and 6.2.

Figure 6.2 shows the efficiency of HPO when compressing data sets containing XML files ranging between 200KB to 2.5MB in size. HPO is able to create a more efficient encoding format with significant results for data sets of graphs 1 and 3. Compared to the second most efficient tool, 7ZIP, HPO is able to apply an additional 2-3% of compression for graph 1 and 3. Data sets of graphs 2 and 4, instead, present a compression size similar 7ZIP and EXI respectively. The compression difference between HPO and EXI for graph 4 is around 0.5% in favour of HPO. HPO presents a linear compression for graphs of figure 6.2 based on basic data types. This feature is attributed to the fixed length encoder, which compared to the statistical compression of 7ZIP, presents a linear encoding style. However, string data types of graph 4, present similar pattern to variable length encoders. This is because the string data types of this data set are sent to the *zlib* string buffer and compressed using DEFLATE algorithm.

#### 6.2.1.2 Random Data Types

Figures 6.3 and 6.4 present the compression size for XML files based on random generated data types. Graphs 1 and 2 of figure 6.3 and 6.4 are based on XML files with random data types but different element name sizes. Graphs 3 and 4, instead, illustrate the compression efficiency for XML files with mostly numeric and string data types respectively. Figure 6.3 presents the results for small XML files of around 5KB to 50KB in size where the presence of the compressed schema file has a negative impact on the final compression. However, as analysed in the previous data sets, this issue is mainly found for XML files of size lower than 20KB.

The main difference between this data set and the previous, is the presence of

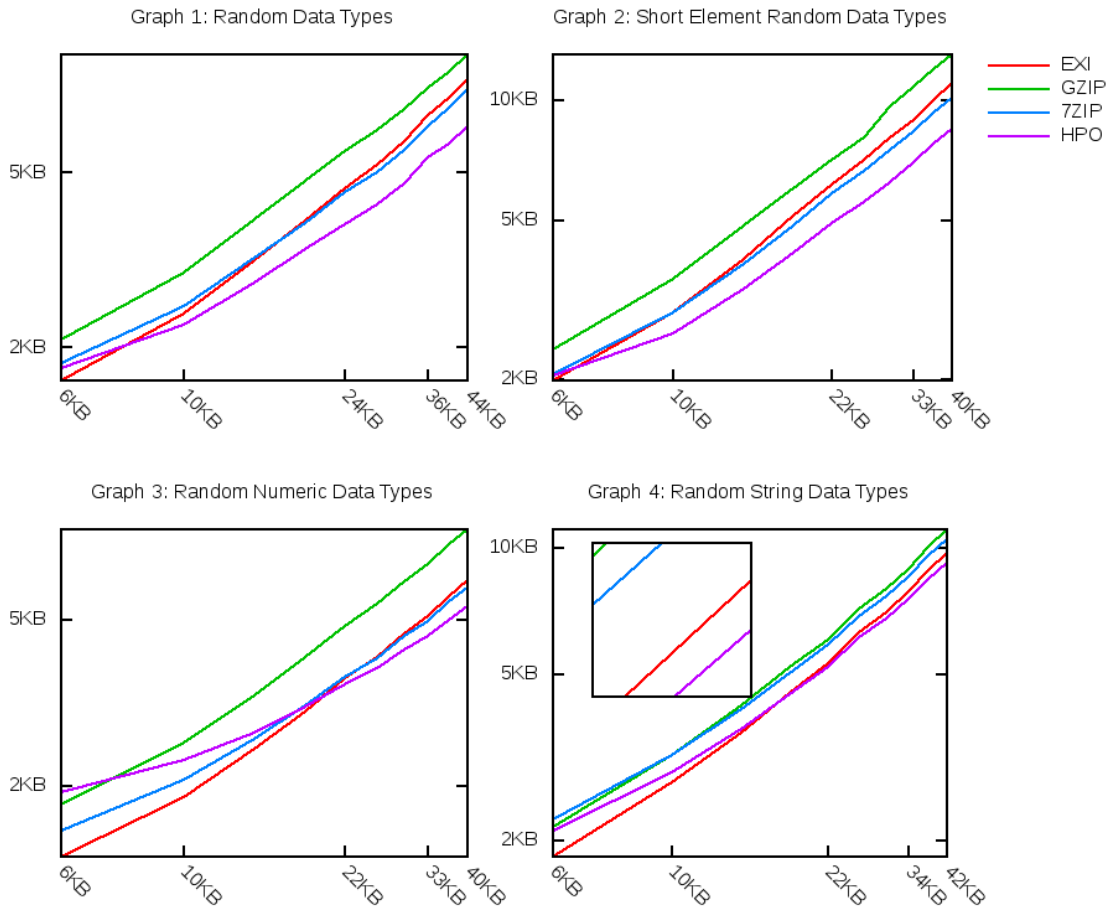


FIGURE 6.3: Random Synthetic Data Types - 5KB-50KB

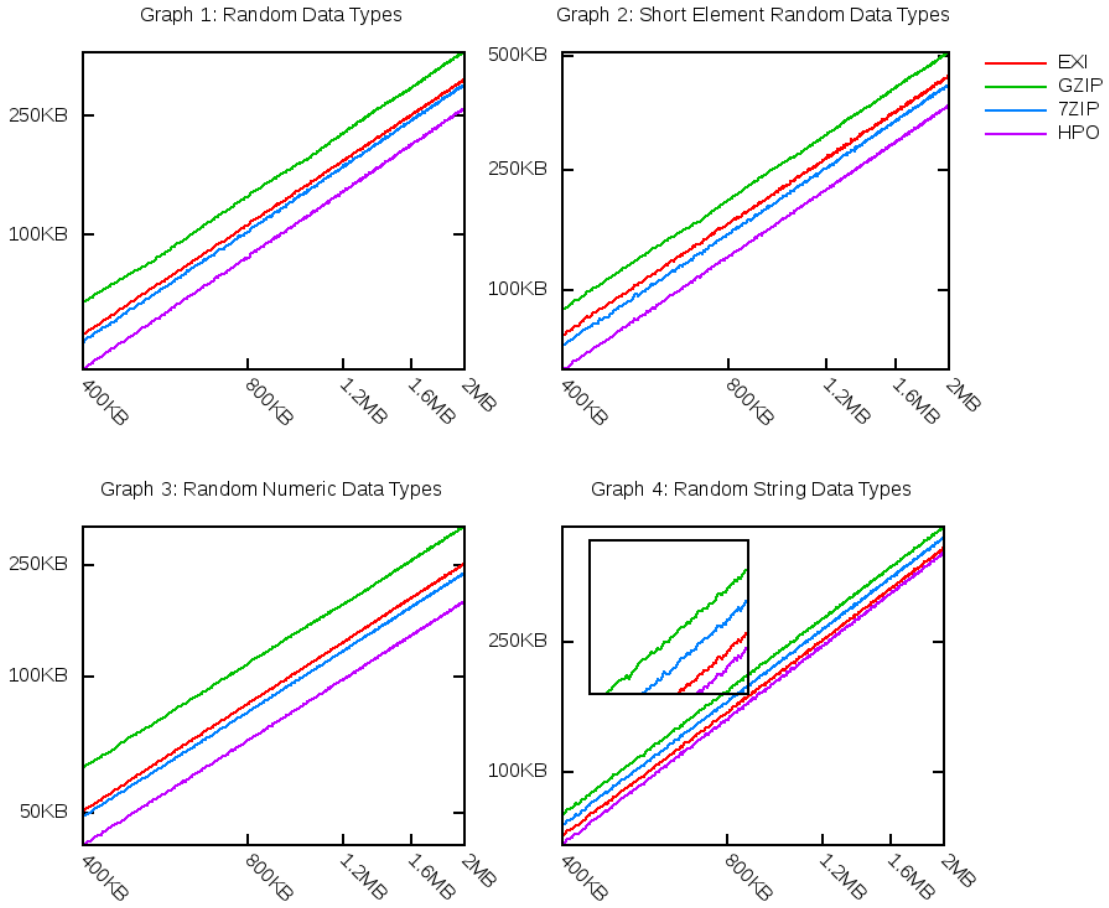


FIGURE 6.4: Random Synthetic Data Types - 200KB-2.5MB

random data types. An XML node with variable length and depth is automatically generated and populated with a mixture of basic and string types. The structure of the node is used to construct `sequence-of` nodes with identical list of randomly generated data types. The result of the generation process is an XML file with a repeating structure and fixed data types with random values. Maintaining a fixed sequence of data types is important in order to allow the schema generation process to allocate correct data types to each element of the XML file.

Having a mixture of basic and string data types allows HPO to implement a full hybrid approach. The fixed length encoder will be used to compressed basic data types while the variable length encoder will manage string data types and other undetected types. A quick analysis demonstrates an efficient compression size in the presence of basic types. Graph 4 of figures 6.3 and 6.4 presents a less efficient compression size compared to other graphs. HPO is

able to achieve an additional 0.5% to 1.5% of compression for XML files greater than 20KB in size. In this scenario, the transparent schema-informed technique improves compression by avoiding encoding the structure of XML. However, encoding the zlib reference integer does not allow HPO to achieve a significant level of compression compared to other graphs of figure 6.4.

Compared to the basic data types graphs of figures 6.1 and 6.2, a less linear encoding is found due to the presence of string data throughout the data set which is compressed using DEFLATE algorithm. Graphs of figure 6.4 show how the transparent schema-informed compression of HPO does not influence the overall compression. HPO is able to compress XML files by an additional 3% to 5% for graphs 1 to 3. The top graphs of Figure 6.3 illustrate how a more compact schema is created for elements with shorter names, resulting in a better compression for files of 5KB to 10KB in size. However, the same graphs in figure 6.4 illustrate how the size of XML element can have a negative impact on the performance of EXI compared to 7ZIP. Despite the additional zlib integer reference encoding, HPO performs better compared to other tools to compress random data sets with mainly numeric data. Here, the key advantage is the efficient high-level data type encoding and the transparent schema-informed compression.

### 6.2.2 Real XML Data

Figure 6.5 shows the compression size comparison for real XML data. A total number of 16 files ranging from 5KB to 30MB have been compressed. The results of figure 6.5 demonstrate how the efficiency of HPO varies according to the type of XML compressed. Optimal results are achieved for XML files such as `baseball.xml`, `orderkey.xml` and `rand1-1988.xml`. An analysis of these XML files shows that the data types within these XML files are consistent with the data types of HPO. Although a small percentage of string types are found, the structured nature of these XML files allows the zlib buffer to compress string data efficiently. For example, `baseball.xml` consists of a list of repeating nodes where 15% are string types and the remaining 85% are integer types.
A good level of compression is found for XML files containing `DateTime`, `Integer`, and `Decimal` data types. XML files based on these data types are able to achieve higher compression sizes as demonstrated for synthetic XML data sets.

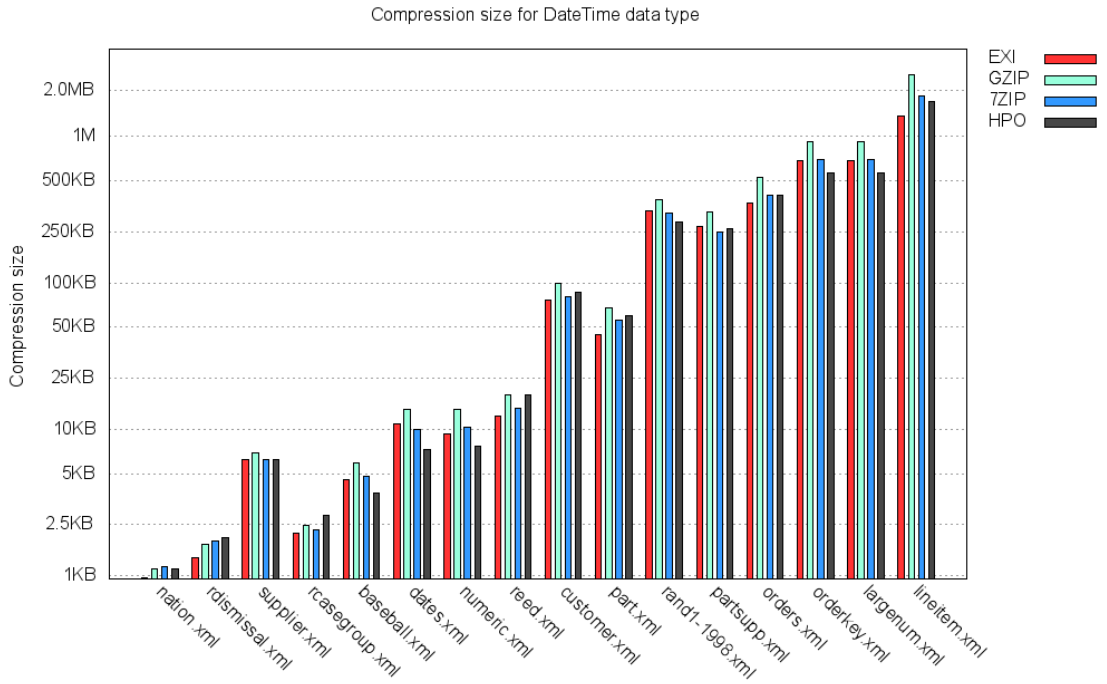


FIGURE 6.5: Real XML Data Set

Negative results are instead found for XML files which do not benefit of HPO basic types or structural format.

Data used to populate these XML files is inconsistent and cannot be associated to any of the HPO data types. XML files such as `reed.xml`, `customer.xml` and `part.xml` contains data types which are not recognised by HPO and therefore compressed using DEFLATE. The importance of recognising XML data types is found in XML files such as `supplier.xml` where only 40% of data types are recognised as `integer` and `decimal`. The remaining 60% is divided into string types and alphanumeric types with a consistent pattern. However, these types do not belong to any existing data types and therefore are compressed as strings. Recognising only 40% of the data types for `supplier.xml` allows HPO to achieve similar level of compression to other compressors. A detailed lists of the results is listed in table D.1 of Appendix E.

#### 6.2.2.1 Compression Ratio

The compression ratio is calculated using the size of the various compressed formats. Figure 6.6 presents the compression ratio for each of the tools examined using data from figure 6.5. This ratio is used to examine the significance of

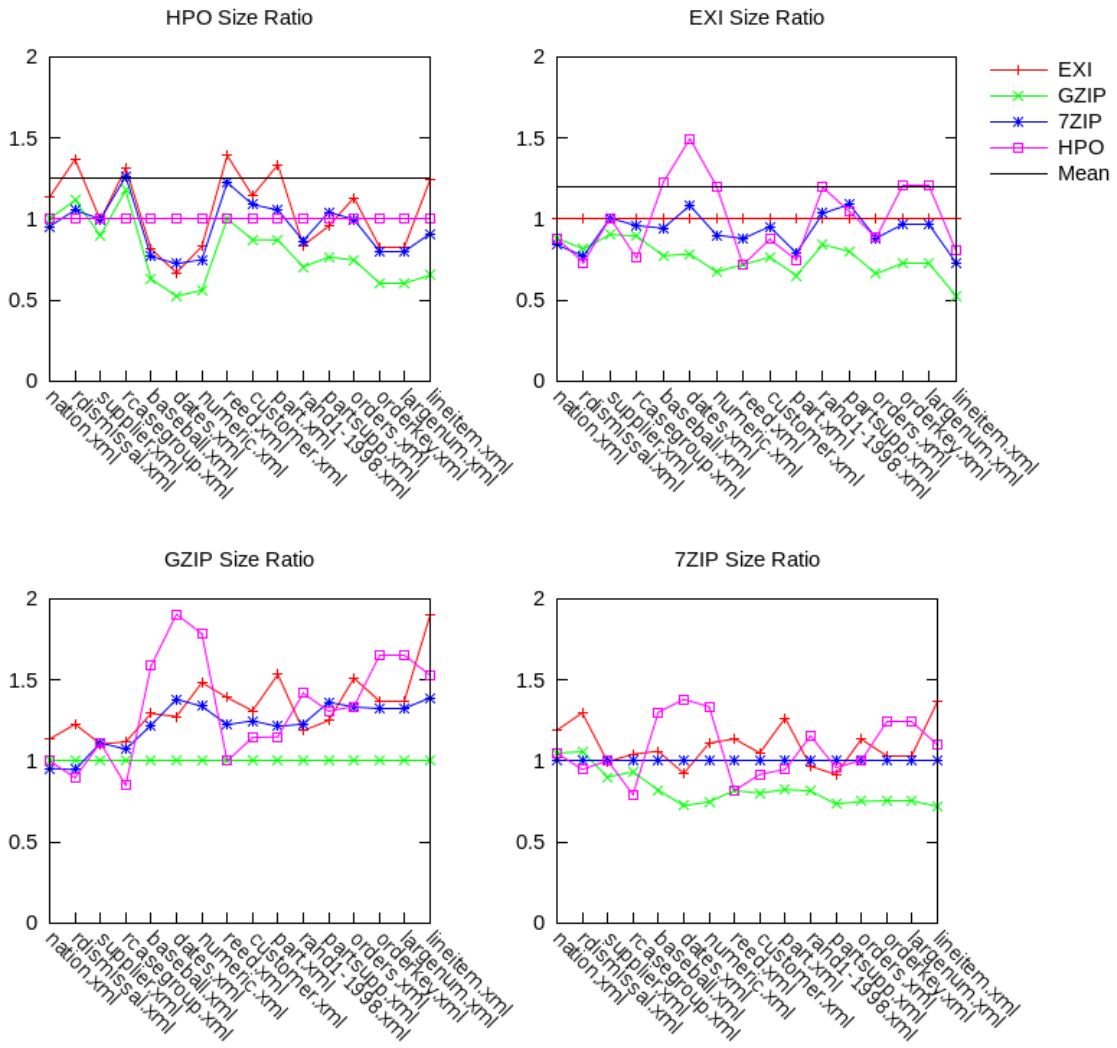


FIGURE 6.6: Real XML Data Set Compression Ratio

encoding size when contrasting tools. Thus, any score above 1 would indicate a tool that outperforms the analysed one. Using the information presented in the graphs, it is possible to isolate each compressor and evaluate its efficiency. In addition, it is possible to compare the efficiency of each tool against the others. The raw data is provided in tables D.2 to D.5 of Appendix E.

The top graph presents the compression ratio of HPO and EXI. A similar encoding size is found for both tools using the current data set. A total of 8 XML files are compressed more efficiently by HPO against EXI and vice versa. The top graphs include an additional line representing the mean value for XML files with ratio above 1 uniquely for HPO and EXI. This line was included to illustrate the overall efficiency of XML-conscious compression of HPO against EXI. The

mean of EXI ratio values above 1 for HPO is 1.25, while the mean of HPO ratio values above 1 for EXI is 1.20. Therefore, EXI presents a slightly better compression ratio for the real XML data set. As discussed in the previous section, this data set is based on unstructured data types which are not recognised by HPO and therefore encoded as strings. This property allows tools based on semantic compression such as EXI to achieve a higher efficiency. The compression of XML Schema in HPO is also a severe burden for small XML files.
The bottom graphs illustrate the efficiency of the general-purpose compressors. GZIP presents the worst results in compression size ratio. With the exception of small XML files, most results present a ratio above 1. XML-conscious compressors usually perform better compared to general-purpose.

## 6.3 Analysis

The results demonstrate the performance of HPO when compressing different types of XML data sets. The XML corpus used for these experiments contains a set of synthetic and real XML data. The aim of the hybrid model is to improve the compression efficiency for XML data that can be found in a variety of fields. Results demonstrate how HPO is able to achieve a better compression for XML documents containing a sequence of basic data types. Synthetic data sets have been used to demonstrate the results of HPO in its ideal domain. An optimal level of compression is found for synthetic XML based on high-level data types such as DateTime which can be efficiently mapped to a lower form by HPO. However, these high-level data types are not usually found in real XML data. The results of this data set presents a less efficient encoding scheme due to the inability of HPO and other compressors to recognise data as specific types and apply a fixed length encoding technique.

Figures 6.7 and 6.8 present a detailed analysis for real XML data. The first graph shows an analysis of the XML file before compression. It is possible to identify the percentage of hybrid compression which was used to compress real XML files. Each bar represents the size of XML file where two stacked columns consist of the percentage of data which is compressed using the fixed length or variable length encoder. As discussed in the previous chapter, the String Buffer of figure 6.7 is the amount of string data extracted from the XML document. This

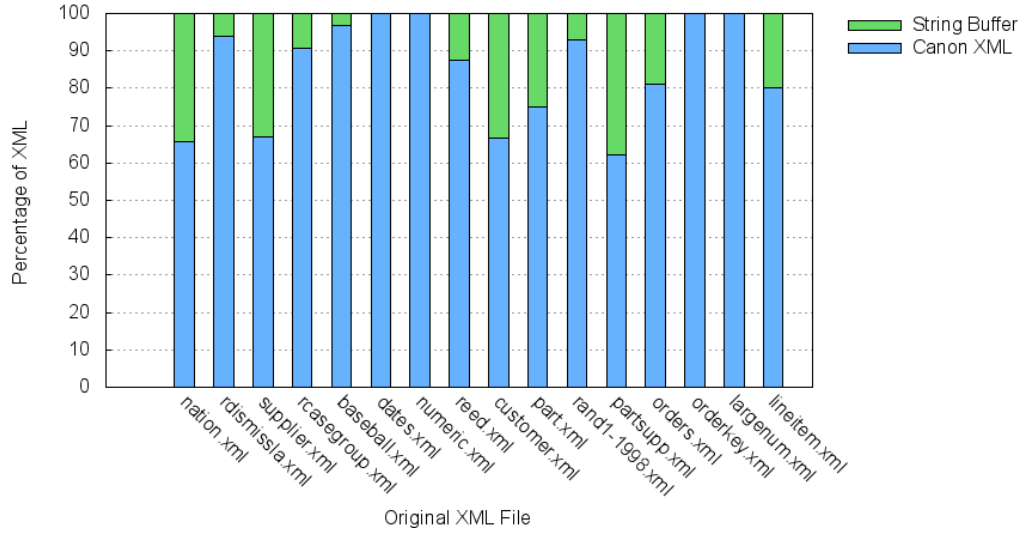


FIGURE 6.7: Real XML Compression analysis - Original XML

data consists of all the data types which have not been recognised by HPO or are based on string types. These results demonstrate that based on the composition of real XML data, a better compression is achieved when the fixed length encoding technique of HPO is implemented between 80 to 100% of the size of the XML file. Therefore, the use of a fixed length encoder, associated with the ability of the knowledge extraction process to recognise basic data types, is the key feature to improve compression size for real XML data sets. A 10% composition of string data types is allowed to be processed by the variable length encoder. As demonstrated by the `rand1-1988.xml` file, this is the ideal amount of character string data types for HPO to achieve a good compression size.

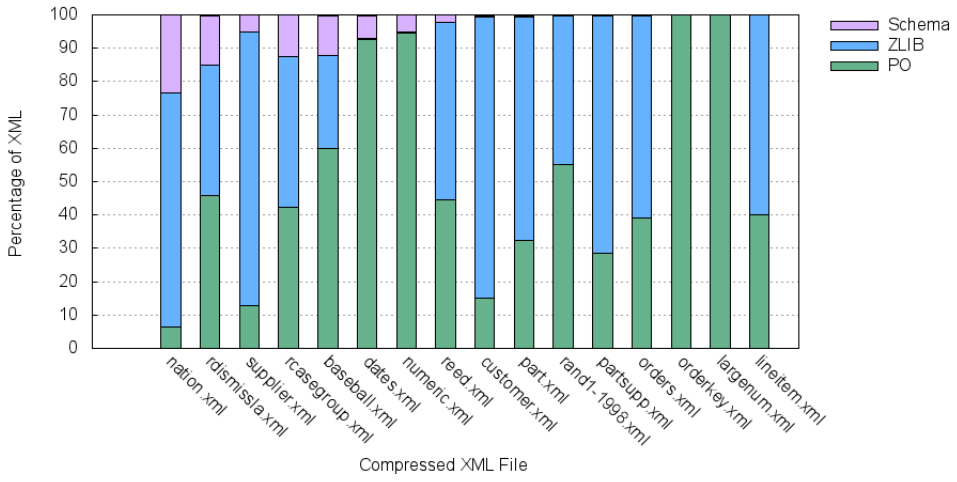


FIGURE 6.8: Real XML Compression analysis - Compressed XML

Graph of figure 6.8 illustrates an analysis of the real XML files after compression. Each bar represents the size of the compressed Canon XML, String Buffer and Schema definition. This graph demonstrates how a better compression is achieved when most data types are recognised, triggering a fixed length encoding technique. For example, in the first graph, only 4% of XML data is passed to the string buffer for `baseball.xml`. However, the compressed equivalent represents the 28% of the compressed format. A detailed analysis for both graphs is provided in table D.6 of Appendix E. These results demonstrate how the fixed length encoder is capable of achieving a more efficient compression format compared to variable length techniques. The second graph also highlights how the compressed schema format is irrelevant for the compression of files above 150KB.

### 6.3.1 Compression Comparison

A compression comparison can be made between synthetic and real XML data. As shown in previous graphs, there is a noticeable difference in compression when the HPO is not able to recognise XML data types. In these scenarios, most of the data is sent to the string buffer to be compressed using the DEFLATE algorithm. Synthetic data set results present a good level of compression for XML files based on string data types as shown in previous figures 6.2 and 6.4. However, the results for real XML data sets do not present similar levels of compression. Based on the results for synthetic XML data, a good level of compression should be achieved when the hybrid model is not able to recognise the XML data types. In the worst scenario, the level of compression achieved by the hybrid model should equal text compression as shown is figure 6.2. From an analysis of real XML data it is possible to relate the performance of HPO to the nature of the data. Most data used is based on alphanumeric characters with repeating strings. The nature of this data allows tools with semantic compression to apply a more efficient encoding technique. For example EXI encodes string data into semantic tables, using reference codes for future instances. This technique, associated to the DEFLATE compression of the semantically aligned encoded streams, enables EXI to achieve a better rate of compression.

Based on these results, it is possible to relate the difference in compression between synthetic and real XML data to the unstructured nature of the data. While synthetic data sets are based on well-known data types, most of data found in real XML is unstructured and specific to the purpose of the XML. For the scope of this research, these data types are defined as *synthetic*. These types are based on a specific pattern with repeating characters which do not allow tools to apply fixed lenght encoding. Due to the way the data is generated, it is not possible to generalise the mapping process of these data types. In addition, data type recognition can be easily broken if a data type does not adhere to previous pattern. Although these data types cannot be defined in a practical tool, it is possible to compress this data efficiently using a fixed length encoding technique.

As discussed in the previous chapter, the hybrid model is based on the compression of complex and basic data types using PO. String and unrecognised data types are instead compressed using the DEFLATE algorithm. Unrecognised data types can be divided into two categories, patterns that exist but are not currently recognised by the PO back-end encoder and patterns specific to the purpose of the XML. The latter are classified as synthetic. The knowledge extraction process of the hybrid model is aimed at recognising repeating sequences of basic data types to be compressed using the fixed length encoder. Here, it is possible to recognise data types such as IPv4, Hexadecimal, and DateTime based on RFC 3339 format. For example, to most compressors the IPv4 format `10.250.56.34` would not match to any standard data type. However, it is possible to encode this format as a 32-bit integer using 4 bytes. A similar encoding technique can be applied to synthetic data types increasing the compression efficiency of HPO.

#### 6.3.1.1 Synthetic Data Types

Synthetic data types have been defined as unique formats with fixed patterns. As shown in previous results, the use of predefined data types does not always apply to real XML data. These documents are based on inconsistent formats mostly constructed using alphanumeric data. Compressing this data using a dictionary compression technique does not present a good compression ratio, even when semantic compression is applied. The lack of local redundancy in

these semantic buffers does allow tools such as EXI to achieve a better compression compared to GZIP and 7ZIP. This poor compression ratio can be found from the results of XML files such as `supplier.xml` of figure 6.5. When presented with data following patterns similar to ‘‘`([a-z][A-Z])*[#_-]([0-9])*`’’, compressors such as EXI would move all data types instances in a semantic buffer to be encoded using DEFLATE. However, the separation between string and numeric data would result in a less efficient compression.
A fixed length encoding technique can be applied to compress synthetic data types from a difference perspective. Using the full potential of the schema language, it is possible to define a synthetic data type which would encode the string pattern using the zlib buffer while encoding the numeric data as integers. From an analysis of real XML data it is possible to identify these synthetic data types and construct an efficient encoding scheme. A snippet of `supplier.xml` is listed below.

LISTING 6.1: Snippet of supplier.xml document

```
<T>
  <S_SUPPKEY>1</S_SUPPKEY>
  <S_NAME>Supplier#000000001</S_NAME>
  <S_ADDRESS> N kD4on9OM Ipw3,gf0JBoQDd7tgrzrddZ</S_ADDRESS>
  <S_NATIONKEY>17</S_NATIONKEY>
  <S_PHONE>27-918-335-1736</S_PHONE>
  <S_ACCTBAL>5755.94</S_ACCTBAL>
  <S_COMMENT>requests haggle carefully. accounts</S_COMMENT>
</T>
<T>
  <S_SUPPKEY>2</S_SUPPKEY>
  <S_NAME>Supplier#000000002</S_NAME>
  <S_ADDRESS>89eJ5ksX3ImxJQBvxObC,</S_ADDRESS>
  <S_NATIONKEY>5</S_NATIONKEY>
  <S_PHONE>15-679-861-2259</S_PHONE>
  <S_ACCTBAL>4032.68</S_ACCTBAL>
  <S_COMMENT>furiously stealthy frays thrash alongside</S_COMMENT>
</T>
<T>
  <S_SUPPKEY>3</S_SUPPKEY>
  <S_NAME>Supplier#000000003</S_NAME>
  <S_ADDRESS>q1,G3Pj6OjIuUYfUoH18BFTKP5aU9bEV3</S_ADDRESS>
  <S_NATIONKEY>1</S_NATIONKEY>
  <S_PHONE>11-383-516-1199</S_PHONE>
  <S_ACCTBAL>4192.40</S_ACCTBAL>
  <S_COMMENT>furiously regular instructions </S_COMMENT>
</T>
```

From this XML file a *synthetic* data type for `<S_NAME>` and `<S_PHONE>` elements is recognised. Standard basic and character string data types can be applied the other elements. The first synthetic data type is based on a repeating string followed by an integer. It is possible to define format `Supplier#000000001` to match the pattern ‘‘`[Supplier#][0-9]{9}`’’. Based on this pattern it is

possible to encode this format using a fixed length encoder. By defining the `Supplier#` string in the schema, this string can be referenced using an approach similar to enumeration. One bit is used to represent `Supplier#`, while the 9 digit integer is encoded in 30 bits. A more efficient encoding can be applied based on the maximum integer found in the data type. For example, if the maximum value found is `Supplier#000000999`, it is possible to encode the synthetic data type using the pattern ``[Supplier#000000][0-9]{3}''. Therefore, string `Supplier#000000` can be encoded in 1 bit while the remaining integer can be encoded using only 10 bits. A similar encoding technique can be applied to the <S_PHONE> element. Pattern ``[0-9]{2}[-][0-9]{3}[-][0-9]{3}[-][0-9]{4}'' can be use to encode this format in 40 bits.
This feature would require a system, similar to HPO, to efficiently compress synthetic data types into a lower form. However, these data types are mostly unique to each XML files and would require additional computational time and memory to allow an optimal encoding.

### 6.3.2 Real XML Data Types

Analysing real XML data types dictates the current possible level of compression achievable by HPO. Figures 6.9 and 6.10 illustrate a detailed analysis of the data types for the real XML data set. As discussed in Chapter 5, the efficiency of the hybrid model highly depends on the data types of the XML. Therefore, in addition to the information provided by other work discussed in Chapter 3, this study present an in-depth analysis of the data types which are used for this data set. Using the information provided in the graphs, it is possible to highlight the correlation between the number of basic data types and the compression efficiency of the hybrid model. Figures 6.9 and 6.10 present the total count of each data type for the subset of real XML data sets where HPO does not provide a significant level of compression. The *x*-axis of the graph lists the data types found in each XML file listed in the *y*-axis. The vertical *z*-axis illustrates the amount of data types found in each XML file. As illustrated in the results section, HPO is able to achieve substantial results when a high number of data types are detected. Data types of the *x*-axis are listed from a low to high level using the encoding efficiency of the fixed length encoder as reference. For example, a constrained `integer` of `6` digits can encoded efficiency into `17` bits using the IER

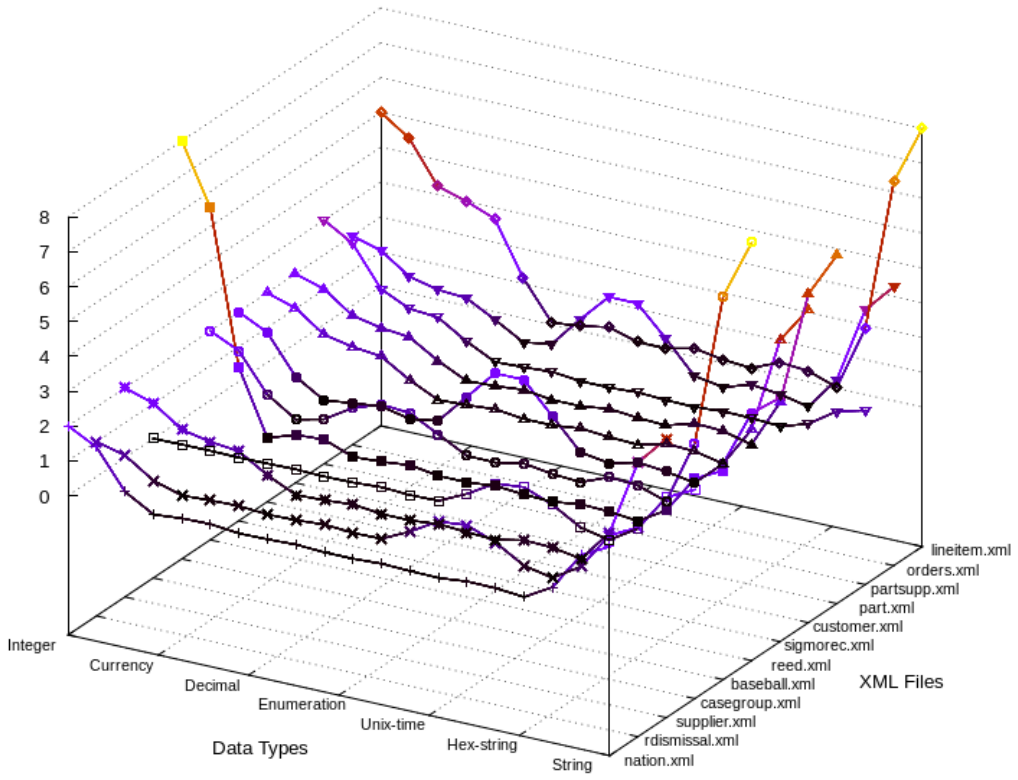


FIGURE 6.9: Data Types analysis for Real XML Data Sets

of PO as opposed to a 6 characters `hex-string` type that requires 24 bits. Both data types are considered basic types, however, the efficiency depends on the integer mapping process. Data types such as `integer`, `bit-string`, `unix-time` are able to achieve a more compact integer representation which adheres to the IER. The additional space required to store information in the schema, as required for enumeration, is also considered. In summary, the *x*-axis data types are listed in order to understand the current and possible levels of compression that can be achieved by HPO.

The data type analysis is performed on a subset of the real XML data sets used in previous results. This subset was selected from the XML files where the compression efficiency of HPO does not present a significant level of compactness compared to other tools. Graph 6.9 illustrate the current data types recognised by HPO. This list is based on the knowledge extraction process and data type recognition of the hybrid model. A good amount of low-level basic data types are recognised, however, a higher number of string types are found in most XML

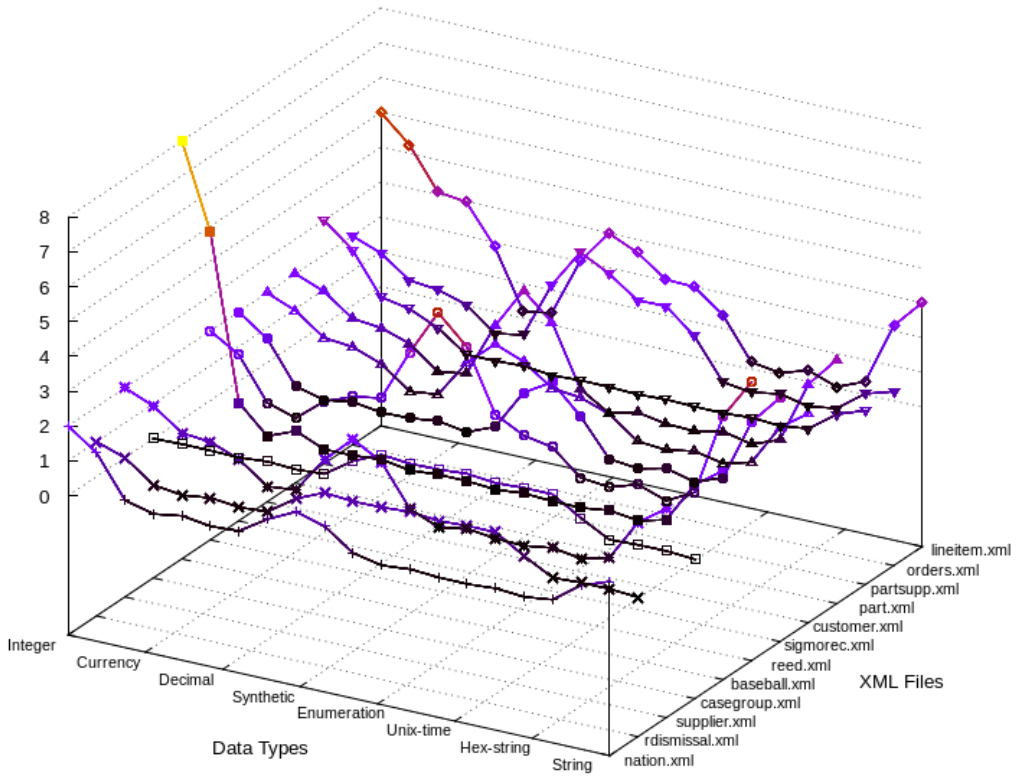


FIGURE 6.10: Synthetic Data Types in Real XML Data Set

files. In addition, data types such as `hex-string` `enumeration` and `unix-time` are not recognised or implemented. In conclusion, data type analysis for current data types shows a higher level of string data types for most XML files. Based on the results of EXI, where string types are moved into semantic streams, a more efficient compression can be achieved using the repetitive patterns found in these string types. Similar levels of compression can be achieved using enumeration. This technique can store repeating element data into the schema. During compression, a semantic technique will be applied to the enumerated values stored in the schema, emulating the semantic compression implemented by EXI and XMill.

Graph 6.10 illustrates an analysis of XML files using synthetic data types. A `Synthetic` data type is listed in the *x*-axis of the graph referring to all those types that can be mapped to a low-level format. Several patterns have been used to match against a sequence of XML elements. The graph demonstrates how most of the string data types of graph 6.9 are now recognised as synthetic

types. Therefore, this implementation would result in less data being sent to the zlib string buffer. In addition, a higher amount of enumerated values is shown. This is achieved mostly in conjunction with the synthetic types in order to allow a more semantic compression. In summary, the second graph illustrates the possibility to improve compression size for real XML data sets using synthetic data types.

#### 6.3.2.1 Data Types Patterns

Real XML data set analysis illustrates the possibility of improving compression size using synthetic data types that can be mapped to a lower form and compressed efficiently using fixed length encoder. These data types are specific to the domain where XML file belongs to and cannot be generalised for practical tools. However, the nature of synthetic data types is based on simple patterns that can be detected using regular expressions. These patterns are mostly based on the union of several alphanumeric characters separated by punctuation. Figure 6.11 illustrates the relationship between regular expression patterns within these data types.

The relationship between these patterns illustrates the variety and combinations that are possible within these data types. These patterns are divided in four types: `[:digit:]` for integers and decimal, `[:alpha:]` for alphanumerical characters and punctuation, `[:fixed:]` for fixed characters in fixed positions,

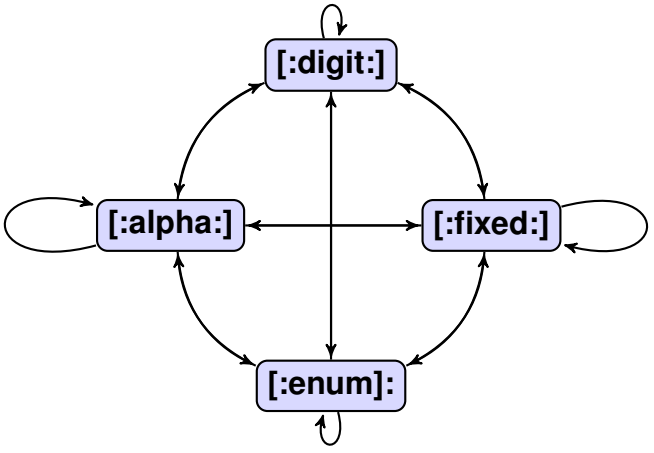


FIGURE 6.11: Data Types Regular Expression Patterns Relationship

and `[:enum:]` for enumerated values. For every regular expression, each pattern can occur multiple times with different combinations.
Regular expressions can be divided into two major categories: numeric characters separated by punctuation and alphanumeric characters separated by enumerated values. The first category allows a simple encoding mechanisms using a similar technique implemented to data types such as `IPv4` and `bit-string`. The second type, instead, requires the alphanumeric characters to be encoded as basic or character string data types, and the enumerated values to be stored in the schema file. Enumeration applied to synthetic data types can improve the compactness of the compressed format for large lists of enumerated values as it also applies a semantic compression of the values.

Figure 6.12 presents an analysis of the patterns found in the data sets of figure 6.10. Each regular expression usually consists of one or many patterns. As shown in figure 6.12, `[:digit:]` is the most common pattern found in these types. This suggests that it is possible to achieve more efficient compression using the fixed length encoder. The second most common patterns are `[:enum:]` and `[:fixed:]`. Encoding these patterns requires data to be stored in the schema and to be referenced using the enumeration techniques implemented by PO. Lastly, these results illustrate that the amount of alphanumerical characters, encoded as strings, is minimal. In summary, since the regular expressions for synthetic data types are mostly based on low-level patterns, a higher level of compression can be achieve using the fixed length encoder.

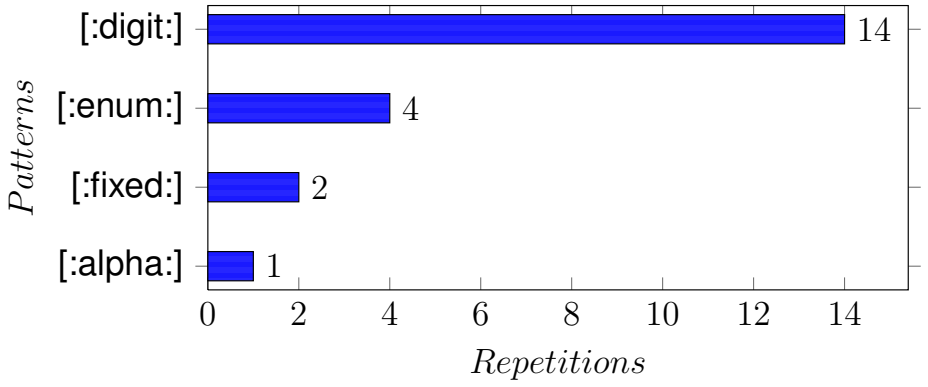


FIGURE 6.12: Regular Expression Patterns

### 6.3.3 Performance Evaluation

The additional compression efficiency achieved by the hybrid model comes at a cost. Previous sections compared the efficiency of HPO against EXI and other general text compressors. As described in Chapter 5, the hybrid model is based on several processes which are required to extract knowledge and compress the XML file using different encoders depending on the data types found. As demonstrated in Chapter 4, the complexity of a tool has a direct impact on the overall performance. Software with a higher complexity and code functions tend to perform slower compared to minimal implementations. Therefore, the additional computational time required by HPO is directly related to the amount of processes needed to achieve an optimal compression.

Figure 6.13 illustrates the performance of the selected tools using the real XML data set. The compression time difference between HPO and other native code implementations is relatively low for files up to 5MB in size. For larger data sets, the time required to compress each file increases drastically. GZIP is the tool with the fastest implementation and the lowest level of compression compared

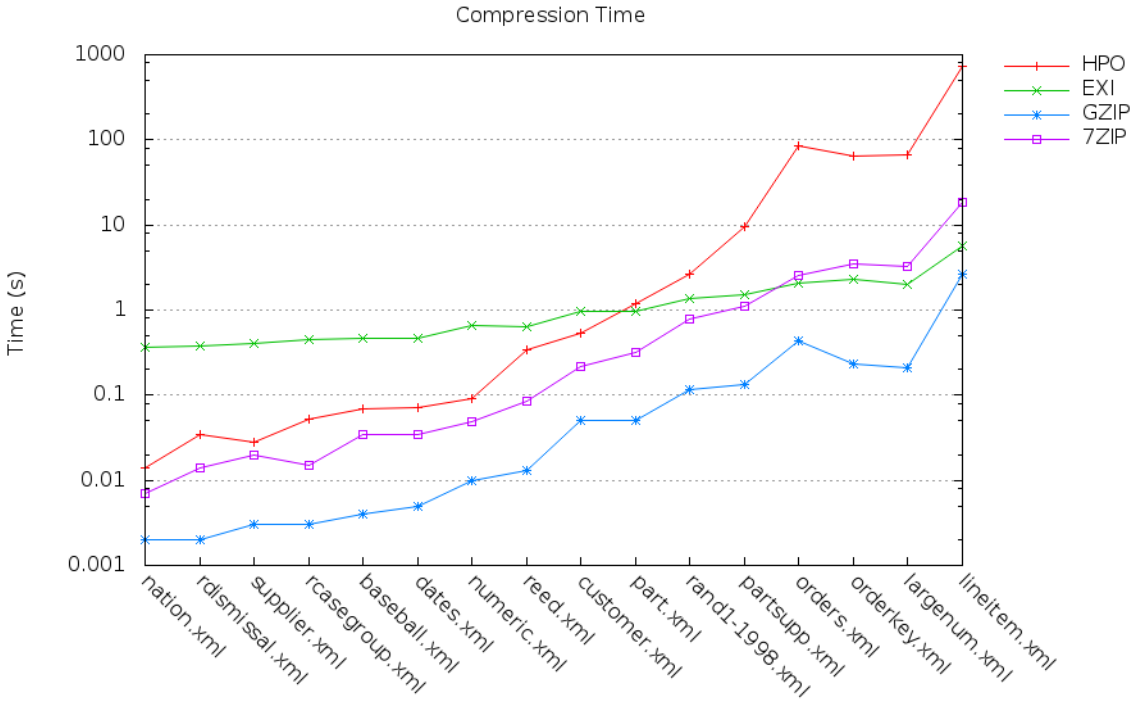


FIGURE 6.13: Synthetic Data Types in Real XML Data Set

to HPO, 7ZIP.
HPO requires more time independently from compression size. The time required to compress an XML file depends on the size of XML and not the final compressed format. Whether or not an optimal compression is achieved, HPO requires the same time to compress XML file as shown in graph 6.13. For example, XML files `rcasegroup.xml` and `baseball.xml` are examples of low and high compression efficiency respectively. HPO is able to achieve a higher compression efficiency for `baseball.xml` due to the presence of basic data types and very few character string types. This is not the case of `rcasegroup.xml` which is based on synthetic data types, which are treated as string data. Although the compression efficiency achieved for `baseball.xml` is higher, the time required to compress this file is not greater compared to `rcasegroup.xml`.
Figure 6.13 also compares the performance of EXI against HPO, 7ZIP and GZIP. It is not possible to evaluate the performance of EXI fully without considering the JVM *effect*. Several implementations of the EXI standard exists and all of them are written in Java. Therefore, the processing time required by EXI can be faster since no byte code interpretation and garbage collection, through the stages of the algorithm, would be needed. Actual results would require an isomorphic implementation written native code. However, current results illustrate an optimal compression size in comparison with 7ZIP and HPO. Raw data for the performance evaluation of figure 6.13 is provided in table D.7 of Appendix E.
In summary, as predicted from the evaluation of XML compressors for network management, the higher software complexity leads to additional compression time. Future work would require an analysis of each process to identify the bottleneck of the hybrid model and improve its performance.

#### 6.3.3.1 Front-end and Back-end Processes

The performance of HPO can be divided into front-end and back-end processes. The front-end of HPO can be described as the part of the system that processes the XML document and converts it into a lower form. This form is subsequently passed to the back-end of the system, the fixed and variable length encoders, PO and *zlib*. This separation allows the hybrid model to adapt to other markup languages that can be validated by a descriptive schema language.
Chapter 4 evaluated the performance of the back-end compressors used by the

hybrid model. Here, PO and *zlib* are the tools with the fastest implementation for small highly-structured XML files. The performance of these tools varies depending on the size of the XML file compressed. However, performance results of section 6.3.3 demonstrate a good speed for GZIP tool which is based on the *zlib* library. Therefore, it is possible to conclude that the additional computational time is mainly required by the front-end and the fixed length encoder.

The main aim of this research is to investigate the benefit of a fixed length encoder applied to markup languages compression. The additional level of compression introduced affects the performance of the hybrid model. As shown in figure 6.13, the performance of tools such as GZIP and 7ZIP decreases with the additional level of compression in respect to figure 6.5. Due to the additional level of compression achieved, HPO is expected to perform as depicted in figure 6.13 for small and medium size XML files. Future work will focus on the optimisation of the hybrid model in order to achieve better performance mainly for large XML documents. This optimisation will focus on both the front-end and the fixed length encoder.

#### 6.3.3.2 Efficiency versus Performance

As described in section 5.3.5 of Chapter 5, HPO is a dynamic system that can run in a hybrid or pure mode. These modes are triggered based on the data types analysed during the initial processes of the hybrid model. During the first parsing process, if the data presented is not suitable for a fixed length encoder, it is possible to directly compress the entire file using the variable length encoding technique of *zlib*. For example, it was evaluated how the compression efficiency of HPO is triggered in the presence of recognised basic data types. In scenarios where the XML data is dominated by string data types, it is possible to compress the entire document in pure mode. This mode can also be triggered based on the amount of processing time required to compress the XML file.
The correct balance between efficiency and performance can be controlled by the hybrid model. This is based on the requirement and the scenario where the application is used. For a high level of performance, the model can be applied in a pure mode. The hybrid mode can be applied only for medium sized files or for XML files containing a subset of basic data types. As shown in figure 6.13, the tools with the highest level of compression are also the most computational

intensive. Therefore, the level of compression achievable by HPO would require additional computational time.

Figure 6.14 presents the results of the compression rate for the real XML data set. The graph illustrates the amount of uncompressed data processed by HPO per second. This rate was calculated using the equation below.

$$\textit{Speed} = \frac{UncompressedData}{SecondstoCompress} \tag{6.1}$$

The *x*-axis represents the size of the uncompressed XML data on a logarithmic scale, while the *y*-axis represents the compression rate. For example, the first result on the *x*-axis represent a file of size $0.5 * 10^4$ with a compression rate of $0.4 * 10^6$ bytes per seconds. This represents the compression rate for the XML file `nation.xml`. The best compression rate is achieved for XML files between $0.1 * 10^5$ and $0.5 * 10^6$ in size. The compression rate peak is reached by XML file `numeric.xml` with a speed of $1.3 * 10^6$ bytes per seconds. More detailed information are available in table D.8 of Appendix E.

The dashed ellipse symbolise the ideal scenarios in which the hybrid model can perform. With this information it is possible to conclude that, for the current implementation of the hybrid model, the best performance is found for XML data sets of size $0.1 * 10^5$ to $0.5 * 10^6$ in bytes. Within this range, most of the XML data sets are able to achieve the best compression rate in comparison to smaller

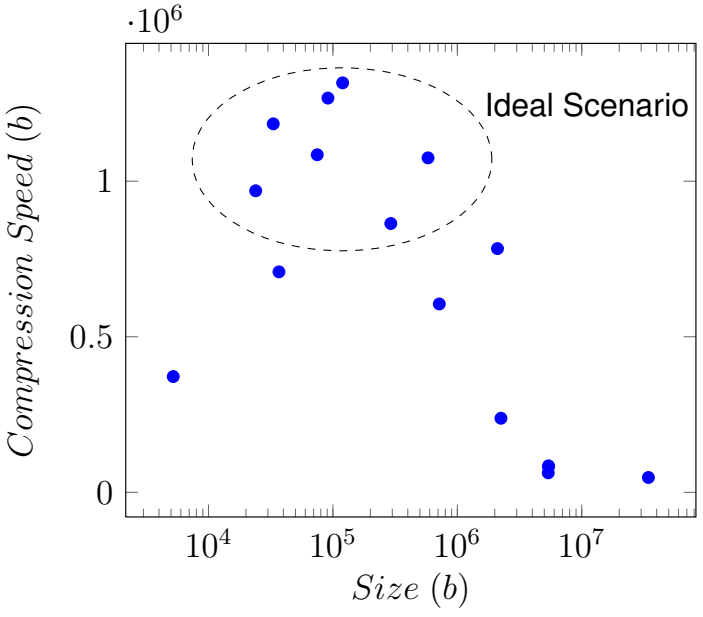


FIGURE 6.14: HPO Compression Rate

and larger XML files. XML files with size greater than $0.5 * 10^7$ are subjects to lower compression rates. The main issue with these XML files is related to the size of the data rather than the data types. As highlighted in the performance evaluation, the current implementation is not able to perform well for large XML file.

## 6.4 Conclusion

This chapter evaluated the performance of the hybrid model. The experiments were performed using an XML corpus based on two major data sets, synthetic and real. The results of the experiments illustrate additional level of compression for most synthetic XML data sets compared to other tools. HPO is not able to achieve an optimal compression for files below 5KB to 10KB in size due to the XML schema compression. All the results above 20KB in size demonstrate substantial levels of compression. Experiment results for real XML data sets present an optimal compression for few XML files. Here, the compressed format size achieved by HPO is considerable compared to other tools. An analysis for real XML data sets where HPO was not able outperform other compression tools was performed. This analysis led to the introduction of synthetic data types. These domain-specific types are mainly dominant in real XML data sets and cannot be generalised for a practical tool. A detailed analysis of the patterns of these data types revealed the possibility of applying a fixed length encoding technique leading to compression efficiency improvements.
This chapter considered the performance issue of HPO and provided an evaluation of the compression speed. The current HPO implementation is affected by a performance issue mainly for larger XML data sets. The ideal scenario in which it can perform is discussed using the compression speed rate.
In conclusion, the results of the experiments illustrate the possibility of using fixed length encoding to improve current general-purpose compressors. Furthermore, with additional level of complexity it is possible to recognise most data types and allow fixed length encoding to be the dominant compression technique in a hybrid model.

# Chapter 7

# Conclusions

This research investigates the use of fixed length encoders to enhance general-purpose compression techniques. The main objective is to improve compression for XML data. However, this research can be extended to other markup languages. The use of high-level data types to enhance compression is studied using a fixed length encoder. Furthermore, this work describes a system to automatically allocate more efficient data types to their closest match. The results presented were achieved using a hybrid model which implements both fixed and variable length encoding. These techniques are used to compress specific part of XML data when their best use case is triggered. A hybrid model was developed based on knowledge of current XML compressors. This implementation is compared to other XML-conscious and general-purpose compressors and has demonstrated a significant level of compression for synthetic XML data sets and promising results for real XML data sets.

## 7.1 Discussion

Chapter 6 presented the results for the compression efficiency of HPO and an insight on its performance. The hybrid model demonstrates the possibility of achieving a better compression format exploiting the XML data type knowledge. Through an analysis of data types, it is possible to apply a fixed length encoder to compress blocks of data into their lowest binary form. This technique has

proven efficient when applied together with a variable length encoder. However, results demonstrate how the fixed length encoding technique can be more efficient to compress XML documents when all the data types found can be mapped to a specific encoding rule. This mainly applies for low-level basic data types e.g. `integer`, `decimal`, `bit-string`, and for high-level basic data types e.g. `IPv4`, `enumeration`, `unix-time`.
The aim of this research is to investigate the use of structured XML data types to improve compression. The results presented provide useful insight to the compression of XML data, and other markup languages, using a fixed length encoder. For this reason, this research mainly focuses on the compression efficiency by exploring the additional level of compression that can be achieved using a fixed length encoder. However, as shown in the performance evaluation, the additional compression comes at a cost. The processing time required to compress XML data is higher compared to general-purpose techniques.

### 7.1.1 Findings

The following sections discuss the results and analysis of the hybrid compression in order to address the research questions raised in Chapter 1.

#### 7.1.1.1 Main Research Question

The main research question raised in the introduction chapter investigates the use of fixed length encoder to enhance general-purpose compression techniques. This research focuses on the compression of markup languages with particular attention to XML as the most pervasive and widely used language across a number of areas. The results of the experiments demonstrate how an additional level of compression can be achieved using a fixed length encoder to compress XML. The implementation described in Chapter 5 focuses uniquely on XML. However, this technique can be applied to all other markup languages that can be thoroughly described by a data definition language. The hybrid model in particular is able to adapt to other markup languages by changing the front-end of the application. By default, the fixed length encoder chosen is able

to compress data in s-expression formats for representing both the data structure and definition language. Therefore, this technique can be extended to all data structures and serialisation formats such as JSON, YAML and ASN.1 notation.

The first set of experiments demonstrate the efficiency of the hybrid model when presented with synthetic XML data types. By controlling the data types constructed in an XML file, it was possible to generate data sets with a range of basic and character string data types. Experiments demonstrate how HPO outperforms other tools when compressing low-level and high-level basic data types. This is possible due to the efficient encoding rules of the fixed length encoder and the transparent schema-informed technique. However, this technique requires the schema to be compressed using a variable length encoder or to be saved locally. By using a domain-specific schema, it is possible to reduce the size of the definition language and improve compression. In addition, as shown in the analysis section, the disadvantages of this technique are negated for large XML data sets. Therefore, the results of the compression efficiency demonstrate that a fixed length encoding technique can be successfully applied to general-purpose compressors in order to enhance markup languages compression. In addition, fixed length encoding can be used as a standalone technique to compress XML data sets with low-level and high-level basic data types.

Recent XML compression techniques are mainly based on variable length encoders. Tools such as XMill, XMLPPM, and to some extent EXI, fully or partially rely on a general-purpose variable length encoder. Although these implementations allow fast and error-free encoding mechanisms, a better compression efficiency can be achieved in conjunction with a fixed length encoder. As shown in the synthetic data sets compression results, this technique, together with a schema-informed approach, can be more efficient compared to variable length encoders. Therefore, it is possible to conclude that a fixed length encoder can not only be used to enhance general-purpose techniques, but it can be used as a standalone back-end compressor. However, this can only be achieved when the front-end application is able to recognise and map XML data types to an efficient lower integer form.

#### 7.1.1.2 Sub Research Questions

The sub-questions raised in Chapter 1 focus on the use of structured data types to aid compression. As discussed in Chapter 3, only few tools investigate the use of XML data types. Most advanced techniques exploit the DTD or XML Schema file to avoid encoding the verbose structure of XML. Sections 5.8.1 of Chapter 5 discussed the difference between the hybrid compression model of EXI and HPO. The major difference between the two models is the compression of high-level basic data types. In addition to low-level basic data types, HPO is able to recognise data types such as `unix-time` and `IPv4` and apply a fixed length encoding technique. This difference is highlighted during the compression of synthetic and real XML data sets in files such as `dates.xml`. For these data sets, the compression of high-level data types is the key feature that allows HPO to perform better than other compressors.
Section 6.3.1 of Chapter 6 discussed the compression comparison between synthetic and real XML data sets and the presence of synthetic data types. It was demonstrated how most string data types can be recognised as synthetic, reducing the amount of data passed to the variable length encoder. However, the different patterns available for synthetic data types do not allow the generalisation of the mapping process to be used in a practical tool. Due to the compression of the additional integer reference required to serialise the string buffer, a better compression can be achieved by forcing synthetic data types encoding. These types can be mapped to their closest built-in data type, as long as it allows a correct decoding. This feature is possible thanks to the presence of suitable pattern for these data types. As shown in previous graph 6.12, these types are mainly based on digit and enumerated values which are perfect candidates for the fixed length encoder.
In conclusion, high level data types are the key components that allow HPO to achieve a substantial level of compression. These types can be investigated further to develop efficient encoding for domain-specific application.

An important feature of the hybrid model is the ability to recognise data types and construct a domain-specific schema. While it is relatively simple to identify low-level basic data types, most of XML files are based on synthetic data types that cannot be recognised by the hybrid model. XML compressors such as EXI, are able to achieve a good compression by separating these data types into semantic models and by applying a variable length encoding. Although this

technique has proven efficient, fixed length encoding is able to achieve more substantial compression results. In order to achieve this level of compression, it is possible to map these types to their closest built-in data types that can ensure a correct decoding. This can be applied to all data types recognised by the hybrid model. For example, data value `<prize>19.50</prize>` can be assigned both decimal and currency data type. This decision is based on the data type that can ensure the lowest level of compression to be achieved. As shown in section 6.3.2, data types can be categorised based on the level of the encoding efficiency, low to high. Therefore, the same concept can be applied to data types in order to improve compression.

## 7.2 Summary of the Thesis

This thesis explores the use of fixed length encoder and the possibility of improving compression for markup languages through the use of data type information. The first chapter provides an introduction to the research area and introduce the aims and objectives. After providing the background knowledge on XML and data compression, the thesis continues with an analysis of relevant XML and general-purpose compressors. In addition, this work studies XML data sets providing an analysis of current results in this area. Chapter 4 investigates the use of XML compression techniques for use in network management. The results of this study, together with the analysis of XML compressors allowed us to have a broader understanding on these compressors. With this background knowledge a hybrid compression model was designed to further investigate the use of fixed length encoders. The motivation and the requirements of the new model are discussed with the goal of improving the compactness of the compressed format.

The implementation is based on the use of data types to be mapped to the fixed length encoder. The performance of the hybrid model is compared against a similar hybrid implementation and best performing general-purpose compressors. Synthetic data and real data are the two major data sets are used for this compression comparison. The results of the experiments demonstrate the additional level of compression that can be achieved using data type information. These data types allow the hybrid model to map data to a lower form used by

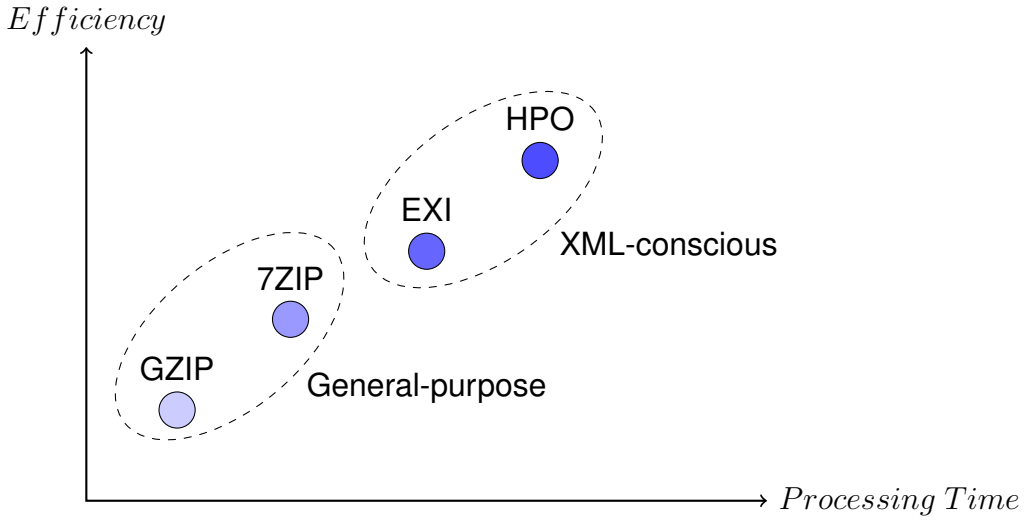


FIGURE 7.1: Compressors Performance

the back-end encoder. Furthermore, this research analysed those XML data sets where the hybrid model fails to recognise data types. Synthetic data types are introduced as patterns that are specific to the data set and that cannot be generalised for a practical tool. A section of the comparison results evaluates the performance and describes the ideal scenario in which the hybrid model can perform. Finally, the thesis discusses the findings and addresses the research questions raised in the introduction.

Figure 7.1 provides an overview of the hybrid model compared to the tools to which it is compared. As shown in the diagram, HPO is capable of achieving higher compression formats at the expense of processing time. Future work intends to explore the performance of HPO to decrease the processing time required to compress markup languages.

## 7.3 Technical Contributions

The thesis presents the following contributions.

**Efficient XML representations**

This research studies a number of XML compressors and present a survey on current implementations. The compression model of these tools is

thoroughly analysed and tested. The survey includes relevant compressors and serialisation techniques found at the basis of many compression tools. This includes a study of EXI and Packedobjects as the two main compressors that are based on fixed length encoding techniques. For both tools, the compression model and functionality are studied. Encoding examples are provided to demonstrate how these tools are able to achieve efficient XML representations.

The best performing tools with native implementations are subject to initial experiments to support network management applications. These results provide an understanding on the compression efficiency and performance of the XML compressors studied for relatively small, highly-structured XML files. Together with the compression comparisons provided by other research, these results allowed us to have a broader understanding on the performance of XML compression tools for a wider range of XML files. The results of this contribution can be used as reference for researchers and developers working in the area of XML compression techniques.

**Hybrid Compression Model**

This work presents a hybrid model to efficiently compress XML data. This model can be extended to other markup languages and notations that can be defined by a data definition language. Based on encoding rules extended by telecommunication and networking notations, this compression model includes an efficient fixed length encoder designed to exchange XML data. The hybrid model is designed based on native libraries in order to extend its use to different platforms and optimise performance.

The model is based on a number of processes followed by two encoding techniques, fixed and variable length. The compression model is based on a transparent schema-informed technique which revolves around the use of fixed length encoder to compress complex and basic data types and the variable length encoder to compress string data types. The process of recognising XML data types is performed during the knowledge extraction of XML. Here, the hybrid model attempts to map its built-in data types to XML elements. String and other unrecognised data types are compressed using the variable length encoder.

The hybrid model is compared against a number of XML-conscious and general-purpose compressors. Results demonstrate the potential of a more efficient compression when most data types are recognised and

compressed using the fixed length encoder. An analysis on the real XML data set is provided to illustrate the existence of synthetic data types. These types are categorised and thoroughly analysed to provide future directions to improve compression using the fixed length encoder.

**High-Level Data Types**

This research presents a system to compress high-level data types without the need of a predefined schema. The design of the hybrid model allows us to recognise XML data types and apply a fixed length encoding technique. This technique has shown substantial results for the compression of basic data types. These types are extended to include high-level formats which are able to efficiency map to a lower form. While existing solutions focus on variable length encoders to compress these data types, this research applies a fixed length encoding through the use of a transparent schema-informed technique. This technique has demonstrated substantial results compared to the semantic compression of other models.

## 7.4 Limitations

This research focuses on the efficiency of fixed length encoder to support general-purpose compressors. Performance was considered but not thoroughly analysed. An overall evaluation was provided to calculate the compression rate speed of the hybrid model and understand the ideal scenario where it can perform. However, more work can be done to address the performance issues of the hybrid model for files over $0.5 * 10^7$ in size. This applies to the performance difference between the front-end and the back-end of the model.

This research does not compare the efficiency of the hybrid model with queriable XML compressors. Few implementations are discussed in the XML compressors survey, however, due to the known efficiency limitation, these tools were discarded from the analysis of XML compression technique for network management and schema-uninformed compression comparison.

One of the main limitations of this approach is associated to the use of whitespaces and newlines to beautify and improve the readability of XML. Ignorable whitespaces have been explored by other XML compressors which have been

defined as near-lossless. This limitation does not allow the hybrid model to consider this data. However, this can be solved with the introduction of a new element to store whitespaces and newlines data.

## 7.5 Future Work

Future work will focus on improving the overall performance and efficiency of the hybrid model. The hybrid model is currently capable of recognising a number of basic data types. However, more work in the knowledge extraction process will allow the hybrid model to recognise more data types and compress them using the fixed length encoder. Since it is not always possible to generalise this process, synthetic data types will be mapped to their closest built-in data type that can ensure a correct decoding. Alternatively, it is possible to provide domain-specific support for more efficient data type detection during the schema generation. This system would required prior knowledge of XML data types to develop synthetic data types support for a specific use case. Users can develop fixed length encoding mechanisms for specific data types to be easily recognised by the schema generation process. Based on specific patterns, both the complex and simple data type analysis can be removed from the run-time load of the hybrid model resulting in improved efficiency and performance. This future work will mainly focus on the front-end of the hybrid model.

The back-end fixed and variable length encoders will also be subject to future work. As discussed in section 5.8.1 of Chapter 5, the main difference between EXI and HPO is based on the encoding technique sequence. While EXI implements the fixed and variable length encoding sequentially on the same stream, HPO categorises the data to be sent to a specific compressor. This allows current implementation to be executed in parallel rather than in sequence. Future work intents to explore the use of a hardware encoder to perform the fixed length encoding compression in parallel. This implementation can result in performance improvements as a dedicated hardware will perform one part of the compression reducing the total processing time. In addition, this research intends to further investigate the performance of the hybrid model. As shown in the performance evaluation section, the hybrid model performance decreases

for files over $0.5 * 10^7$ in size. This issue is be investigated in order to improve the fixed length encoder performance.

Working with large XML files and creating memory representations to assign the correct data types inevitably leads to the disadvantages of using a DOM parser. Although current implementation has a high level of optimisation, the size of the memory representation is still considerable. Future work will investigate the use of optimised DOM parsers capable of constructing efficient memory trees. This work will be performed together with a memory and power consumption study of the hybrid model for embedded devices.

The efficiency of the hybrid model can be applied to other markup languages. HTML will be the first markup language to which this work will be extended. This will lead to the use of a new domain-specific schema, which is capable of easily mapping to the back-end encoder and provide more support for synthetic data types. Here, the application of near-lossless and fully-lossless compression for markup language will be investigated, including whitespaces and newlines support.

Finally, future work plans to improve the efficiency of the hybrid model. This will be achieved by implementing more semantic compression for string data. Instead of constructing a string buffer, this approach plans to enumerate each string type, storing its data into the schema. This feature will be compared to the current implementation to test the compression difference.

# Appendix A

# Data and Protocol Listing

LISTING A.1: ASN.1 protocol

```
personnel DEFINITIONS ::= BEGIN

personnel ::= SEQUENCE {
        name            nameType,
        title           IA5String,
        number          INTEGER,
        dateOfHire      IA5String (SIZE (8)),
        nameOfSpouse    nameType,
        children        childrenType
}
nameType ::= SEQUENCE {
       givenName       IA5String (SIZE (1..64)),
       initial         IA5String (SIZE (1)),
       familyName      IA5String (SIZE (1..64))
}

children ::= SEQUENCE OF {
       ChildInformation childType
}

childType ::= SEQUENCE {
       name           nameType  OPTIONAL,
       dateOfBirth    IA5String (SIZE (8))  OPTIONAL
}

END
```

LISTING A.2: SCM protocol

```
(define personnel
  '(personnel sequence
              (name sequence
                    (givenName string (size 1 64))
                    (initial string (size 1))
                    (familyName string (size 1 64)))
              (title string)
              (number integer)
              (dateOfHire string (size 8))
```

```
            (nameOfSpouse sequence
                          (givenName string (size 1 64))
                          (initial string (size 1))
                          (familyName string (size 1 64)))
      (children sequence-of
                (ChildInformation sequence-optional
                        (name sequence
                              (givenName string (size 1 64))
                              (initial string (size 1))
                              (familyName string (size 1 64)))
                        (dateOfBirth string (size 8))))))
```

LISTING A.3: XSD protocol

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:include schemaLocation="http://zedstar.org/xml
                              /schema/packedobjectsDataTypes.xsd" />
  <!-- User defined types -->
  <xs:simpleType name="nameString">
    <xs:restriction base="string">
      <xs:minLength value="1" />
      <xs:maxLength value="64" />
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name="nameType">
    <xs:sequence>
      <xs:element name="givenName" type="nameString" />
      <xs:element name="initial">
        <xs:simpleType>
          <xs:restriction base="string">
            <xs:length value="1" />
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:element name="familyName" type="nameString" />
    </xs:sequence>
  </xs:complexType>

  <xs:element name="personnel">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name" type="nameType" />
        <xs:element name="title" type="string" />
        <xs:element name="number" type="integer" />
        <xs:element name="dateOfHire">
          <xs:simpleType>
            <xs:restriction base="string">
              <xs:length value="8" />
            </xs:restriction>
          </xs:simpleType>
        </xs:element>
        <xs:element name="nameOfSpouse" type="nameType" />
        <xs:element name="children">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="ChildInformation" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="name" type="nameType" minOccurs="0"/>
                    <xs:element name="dateOfBirth" minOccurs="0">
```

```
                    <xs:simpleType>
                      <xs:restriction base="string">
                        <xs:length value="8" />
                      </xs:restriction>
                    </xs:simpleType>
                  </xs:element>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```

LISTING A.4: ASN.1 data

```
myPersonnel personnel ::= {
            name ::=
            {
                  givenName   "John",
                  initial     "P",
                  familyName  "Smith"
            }
            title       "Director",
            number      51,
            dateOfHire  "19710917",
            nameOfSpouse::=
            {
                  givenName "John",
                  initial "P",
                  familyName "Smith"
            }
            children ::=
            {
            ChildrenInformation ::=
                  {
                  name ::=
                        {
                        givenName   "Ralph",
                        initial     "T",
                        familyName  "Smith"
                        }
                  dateOfBirth "19571111",
                  name ::=
                        {
                        givenName   "Susan",
                        initial     "B",
                        familyName  "Jones"
                        }
                  dateOfBirth "19590717",
                  }
            }
}
```

LISTING A.5: SCM data

```
(define personnel-values '(personnel
                           (name
```

```
 (givenName "John")
 (initial "P")
 (familyName "Smith"))
(title "Director")
(number 51)
(dateOfHire "19710917")
(nameOfSpouse
 (givenName "Mary")
 (initial "T")
 (familyName "Smith"))
(children
 (ChildInformation
   (name
    (givenName "Ralph")
    (initial "T")
    (familyName "Smith"))
   (dateOfBirth "19571111"))
 (ChildInformation
   (name
    (givenName "Susan")
    (initial "B")
    (familyName "Jones"))
   (dateOfBirth "19590717")))))
```

LISTING A.6: XML data

```
<?xml version="1.0" encoding="UTF-8"?>
<personnel>
  <name>
    <givenName>John</givenName>
    <initial>P</initial>
    <familyName>Smith</familyName>
  </name>
  <title>Director</title>
  <number>51</number>
  <dateOfHire>19710917</dateOfHire>
  <nameOfSpouse>
    <givenName>Mary</givenName>
    <initial>T</initial>
    <familyName>Smith</familyName>
  </nameOfSpouse>
  <children>
    <ChildInformation>
      <name>
        <givenName>Ralph</givenName>
        <initial>T</initial>
        <familyName>Smith</familyName>
      </name>
      <dateOfBirth>19571111</dateOfBirth>
    </ChildInformation>
    <ChildInformation>
      <name>
        <givenName>Susan</givenName>
        <initial>B</initial>
        <familyName>Jones</familyName>
      </name>
      <dateOfBirth>19590717</dateOfBirth>
    </ChildInformation>
  </children>
</personnel>
```

LISTING A.7: XML data

```
<?xml version="1.0" encoding="UTF-8"?>
<student>
  <module>FuncProg</module>
  <hours>48</hours>
  <courses>CS</courses>
  <ref>AABBCCDDEE</ref>
</student>
```

LISTING A.8: XSD protocol

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:include schemaLocation="http://zedstar.org/xml
                              /schema/packedobjectsDataTypes.xsd" />
  <xs:element name="student">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="module">
          <xs:simpleType>
            <xs:restriction base="string">
              <xs:minLength value="0"/>
              <xs:maxLength value="10"/>
            </xs:restriction>
          </xs:simpleType>
        </xs:element>
        <xs:element name="hours">
          <xs:simpleType>
            <xs:restriction base="integer">
              <xs:minInclusive value="30" />
              <xs:maxInclusive value="60" />
            </xs:restriction>
          </xs:simpleType>
        </xs:element>
        <xs:element name="courses">
          <xs:simpleType>
            <xs:restriction base="string">
              <xs:enumeration value="CS"/>
              <xs:enumeration value="CIS"/>
              <xs:enumeration value="ITMB"/>
            </xs:restriction>
          </xs:simpleType>
        </xs:element>
        <xs:element name="ref">
        <xs:simpleType>
          <xs:restriction base="hex-string">
            <xs:minLength value="1" />
            <xs:maxLength value="64" />
          </xs:restriction>
        </xs:simpleType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

LISTING A.9: Example of ASN.1 notation protocol

```
1 EmailProtocol DEFINITIONS ::== BEGIN
2
```

```
3       EmailHeader ::== SEQUENCE {
4           idNo            INTEGER,
5           from            IA5String
6       }
7
8       EmailBody   ::== SEQUENCE {
9           message         IA5String,
10          confirmation    BOOLEAN
11      }
12
13  END
```

LISTING A.10: Example of ASN.1 notation message

```
1  JohnEmail EmailHeader ::== {
2            idNo         34,
3            from         "John"
4  }
```

### PO Encoding

The transformation sequence can be broken down into various stages using XML and XSD examples provided in code listing A.7 and A.8 of Appendix A. Using s-expressions it is possible to represent data and protocol in a concise format which can then be easily mapped to the IER. The first stage is to create a combined form by merging information from both data and protocol.

LISTING A.11: PO Normal form

```
1  ;; Normal form
2  ((student sequence)
3   (module string (size 0 10) "FuncProg")
4   (hours integer (range 30 60) 48)
5   (courses enumerated 0 2)
6   (ref hex-string (size 1 64) "AABBCCDDEE"))
```

XML data is *assisted* by the constrains provided by the XSD protocol. This example provides four simple types: `string`, `integer`, `enumerated`, `hex-string` and one complex type `sequence`. This complex type does not affect how information is encoded, therefore, it does not increase the size of the compressed format.

The following code presents the second and third stages of encoding process. Line 2 of code listing A.11 is transformed into the following *integer* form which is then converted into a lower *core* form.

LISTING A.12: String type PO Integer form

```
(integer (range 0 10) 8)
(integer (range 0 127) 70)
(integer (range 0 127) 117)
(integer (range 0 127) 110)
(integer (range 0 127) 99)
(integer (range 0 127) 80)
(integer (range 0 127) 114)
(integer (range 0 127) 111)
(integer (range 0 127) 103)
```

LISTING A.13: String type PO Lower form

```
(unsigned (bits 4) 8)
(unsigned (bits 7) 70)
(unsigned (bits 7) 117)
(unsigned (bits 7) 110)
(unsigned (bits 7) 99)
(unsigned (bits 7) 80)
(unsigned (bits 7) 114)
(unsigned (bits 7) 111)
(unsigned (bits 7) 103)
```

The IER encodes data efficiently using a sequence of unsigned integers. The first unsigned integer encodes the length of the string, value `8`, using 4 bits. Similar to PER, each character is encoded using 7 bits.
The following integer and core form is the representation of the value in line 3 of code listing A.11.

LISTING A.14: Integer type PO Integer form

```
(integer (range 30 60) 48)
```

LISTING A.15: Integer type PO Lower form

```
(unsigned (bits 5) 18)
```

Constrained values do not require the length to be encoded. The lower bound is subtracted from the integer value (ensuring a positive integer) and then encoded using 5 bits.
Enumeration encodes complex simple types efficiently using information provided by the schema. Integer range is produced using the number of enumerated items listed in the protocols.

LISTING A.16: Enumeration type PO Integer form

```
(integer (range 0 2) 0)
```

LISTING A.17: Enumeration type PO Lower form

```
(unsigned (bits 2) 0)
```

An unsigned integer `0` is encoded in two bits representing the first enumeration choice.

`Hex-string` is an example of how to encode a high level data type efficiently using the knowledge provided by the protocol. Hexadecimal characters are used to represent 4-bits in a human-readable format. Therefore, it is possible to

encode each character using 4 bits instead of 7 bits required to encode string types.

LISTING A.18: Hexadecimal type PO Integer form

```
(integer (range 1 64) 10)
(integer (range 0 15) 10)
(integer (range 0 15) 10)
(integer (range 0 15) 11)
(integer (range 0 15) 11)
(integer (range 0 15) 12)
(integer (range 0 15) 12)
(integer (range 0 15) 13)
(integer (range 0 15) 13)
(integer (range 0 15) 14)
(integer (range 0 15) 14)
```

LISTING A.19: Hexadecimal type PO Lower form

```
(unsigned (bits 6) 9)
(unsigned (bits 4) 10)
(unsigned (bits 4) 10)
(unsigned (bits 4) 11)
(unsigned (bits 4) 11)
(unsigned (bits 4) 12)
(unsigned (bits 4) 12)
(unsigned (bits 4) 13)
(unsigned (bits 4) 13)
(unsigned (bits 4) 14)
(unsigned (bits 4) 14)
```

The first unsigned integer encodes the length of the `hex-string` value using 6 bits. 4 bits are then required to store each of the hexadecimal characters.

# Appendix B

# Compressors Execution and Ratio Results

TABLE B.1: Compressors Usage

| Compressor | Command |
|---|---|
| XMLPPM | xmlppm <doc.xml>doc.xppm |
| DTDPPM | dtdppm dtddoc.xml xmldoc.xml [xmldoc.xml.xppm] |
| WBXML | xml2wbxml -o output.wbxml input.xml |
| XMILL | xcmill -f -P file.xml |
| ZLIB | zpipe <file>encodedxml.zlib |
| PO | packedobjects –schema schema.xsd –in file.xml –out pofile.po |

TABLE B.2: System Specification

| **Category** | **Value** |
|---|---|
| Model | Dell Latitude E6510 |
| OS | Ubuntu 12.04 (precise) 32-bit |
| kernel | Linux 3.2.0-25-generic-pae |
| CPU | Intel® Core™ i7 CPU M 640 @ 2.80GHz x 4 |
| RAM | 2 x 2048 MB DDR3 @ 1067 MHz |
| Hard Disk | ATA Disk TOSHIBA MK5056GS 500GB |
| Compiler | gcc (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3 |
| Flags | -O3 -march=i386 |

Details of the compression ratio highlighting the compression tools and datasets where the ratio is above 1

TABLE B.3: PO Compression ratio results

| Compressor | PO | DTDPPM | XMLPPM | WBXML | XMILL | ZILB |
|---|---|---|---|---|---|---|
| Temper-sens | 1 | 0.1086037863 | 0.091159941 | 0.0108493252 | 0.0396910401 | 0.1399782135 |
| personnel | 1 | 0.1284728328 | 0.1367134203 | 0.0155510361 | 0.0545845411 | 0.1449082537 |
| router-qos | 1 | 0.166353116 | 0.2299531565 | 0.0215677261 | 0.0889202604 | 0.2564516339 |
| sensor | 1 | 0.0668997148 | 0.0609221923 | 0.0068491092 | 0.0222221222 | 0.066417959 |
| switch-config | 1 | 0.159775 | 0.1089493854 | 0.0154283535 | 0.0549905352 | 0.18308334 |
| purchaseorder | 1 | 0.1803207845 | 0.1396748946 | 0.0179490988 | 0.0628317625 | 0.1940324971 |
| router-disc | 1 | 0.0928991451 | 0.1045390889 | 0.0122126708 | 0.0486458513 | 0.1624891962 |
| router-addnet | 1 | 0.1865529073 | 0.1558728882 | 0.0211609736 | 0.0643687708 | 0.1688821094 |
| iptel-ethinf | 1 | 0.0477291674 | 0.0500611315 | 0.006491381 | 0.0206122032 | 0.0727677449 |
| iptel-devinfo | 1 | 0.1163385203 | 0.123902112 | 0.0168777197 | 0.0461929473 | 0.1526259125 |

TABLE B.4: DTDPPM Compression ratio results

| Compressor | PO | DTDPPM | XMLPPM | WBXML | XMILL | ZILB |
|---|---|---|---|---|---|---|
| Temper-sens | 9.2077821012 | 1 | 0.8393808734 | 0.0998982228 | 0.3654664484 | 1.2888888889 |
| personnel | 7.7837467921 | 1 | 1.0641426466 | 0.1210453276 | 0.424872247 | 1.1279291549 |
| router-qos | 6.0113091009 | 1 | 1.3823195026 | 0.1296502683 | 0.5345271708 | 1.5416100408 |
| sensor | 14.9477468354 | 1 | 0.9106495077 | 0.1023787505 | 0.3321706569 | 0.992798836 |
| switch-config | 6.2588014395 | 1 | 0.6818925701 | 0.0965630009 | 0.3441748408 | 1.1458822721 |
| purchaseorder | 5.5456724138 | 1 | 0.7745912101 | 0.0995398222 | 0.3484443722 | 1.0760406665 |
| router-disc | 10.7643617021 | 1 | 1.1252965648 | 0.131461606 | 0.5236415387 | 1.7490924806 |
| router-addnet | 5.3604096262 | 1 | 0.8355425302 | 0.1134314864 | 0.3450429784 | 0.9052772849 |
| iptel-ethinf | 20.9515492402 | 1 | 1.0488582614 | 0.136004489 | 0.4318575899 | 1.5245969913 |
| iptel-devinfo | 8.5956052876 | 1 | 1.0650136488 | 0.1450742168 | 0.3970563423 | 1.3119121009 |

TABLE B.5: XMLPPM Compression ratio results

| Compressor | PO | DTDPPM | XMLPPM | WBXML | XMILL | ZILB |
|---|---|---|---|---|---|---|
| Temper-sens | 10.9697306584 | 1.1913542847 | 1 | 0.1190141758 | 0.4354000193 | 1.5355233002 |
| personnel | 7.3145708583 | 0.9397236387 | 1 | 0.1137491557 | 0.3992624939 | 1.0599416896 |
| router-qos | 4.3487117774 | 0.7234217546 | 1 | 0.0937918246 | 0.3866885838 | 1.1152342406 |
| sensor | 16.4143797468 | 1.0981173235 | 1 | 0.1124238795 | 0.3647623527 | 1.0902096005 |
| switch-config | 9.1785740364 | 1.4665066667 | 1 | 0.1416102846 | 0.5047346986 | 1.6804439912 |
| purchaseorder | 7.1594827586 | 1.2910035473 | 1 | 0.1285062636 | 0.4498429206 | 1.3891723176 |
| router-disc | 9.5657998424 | 0.8886546278 | 1 | 0.1168239646 | 0.4653364767 | 1.5543391274 |
| router-addnet | 6.415483871 | 1.1968271678 | 1 | 0.1357578847 | 0.4129568106 | 1.0834604489 |
| iptel-ethinf | 19.9755772647 | 0.9534176702 | 1 | 0.1296690831 | 0.4117406572 | 1.4535777116 |
| iptel-devinfo | 8.0708874459 | 0.9389551027 | 1 | 0.1362181761 | 0.3728180787 | 1.2318265615 |

TABLE B.6: WBXML Compression ratio results

| Compressor | PO | DTDPPM | XMLPPM | WBXML | XMILL | ZILB |
|---|---|---|---|---|---|---|
| Temper-sens | 92.1716307039 | 10.0101880878 | 8.4023604201 | 1 | 3.6583878887 | 12.902020202 |
| personnel | 64.3043969204 | 8.2613680324 | 8.7912740428 | 1 | 3.5100259989 | 9.3182378627 |
| router-qos | 46.3655739369 | 7.7130576982 | 10.6619100805 | 1 | 4.1228389093 | 11.8905271925 |
| sensor | 146.0043881857 | 9.7676519273 | 8.8949074188 | 1 | 3.244527357 | 9.6973134636 |
| switch-config | 64.8157304543 | 10.3559333333 | 7.0616339964 | 1 | 3.5642517065 | 11.8666804173 |
| purchaseorder | 55.7131034483 | 10.0462305184 | 7.7817218543 | 1 | 3.5005524862 | 10.8101525831 |
| router-disc | 81.8821710008 | 7.6067836874 | 8.5598875525 | 1 | 3.9832279148 | 13.3049681488 |
| router-addnet | 47.2568049155 | 8.8158943459 | 7.366054668 | 1 | 3.0418624427 | 7.9808288973 |
| iptel-ethinf | 154.0504243142 | 7.352698483 | 7.7119385474 | 1 | 3.1753186461 | 11.2099019854 |
| iptel-devinfo | 59.2497101113 | 6.8930236009 | 7.3411642163 | 1 | 2.7369187383 | 9.0430410737 |

TABLE B.7: XMILL Compression ratio results

| Compressor | PO | DTDPPM | XMLPPM | WBXML | XMILL | ZILB |
|---|---|---|---|---|---|---|
| Temper-sens | 25.1946030623 | 2.736229288 | 2.2967385296 | 0.2733444431 | 1 | 3.5266955267 |
| personnel | 18.3202053037 | 2.3536486724 | 2.5046179275 | 0.2848981746 | 1 | 2.6547489579 |
| router-qos | 11.2460309405 | 1.8708122893 | 2.5860603131 | 0.2425513152 | 1 | 2.8840630095 |
| sensor | 45.0002025316 | 3.0105007147 | 2.7415109938 | 0.3082113017 | 1 | 2.9888216053 |
| switch-config | 18.1849475825 | 2.9055 | 1.9812388624 | 0.2805637992 | 1 | 3.3293609415 |
| purchaseorder | 15.9155172414 | 2.869898554 | 2.2229981939 | 0.2856691919 | 1 | 3.0881275529 |
| router-disc | 20.5567375887 | 1.9097033487 | 2.148982618 | 0.2510526692 | 1 | 3.3402477672 |
| router-addnet | 15.535483871 | 2.8981896822 | 2.4215607401 | 0.3287459636 | 1 | 2.6236652866 |
| iptel-ethinf | 48.5149496744 | 2.3155781522 | 2.4287132748 | 0.3149290233 | 1 | 3.530323484 |
| iptel-devinfo | 21.648326376 | 2.518534257 | 2.6822733586 | 0.3653743847 | 1 | 3.3040955682 |

TABLE B.8: ZLIB Compression ratio results

| Compressor | PO | DTDPPM | XMLPPM | WBXML | XMILL | ZILB |
|---|---|---|---|---|---|---|
| Temper-sens | 7.1439688716 | 0.775862069 | 0.6512437811 | 0.0775072418 | 0.2835515548 | 1 |
| personnel | 6.9009181637 | 0.8865805052 | 0.9434481253 | 0.1073164277 | 0.3766834514 | 1 |
| router-qos | 3.89937075 | 0.6486724746 | 0.8966726124 | 0.0841005604 | 0.3467330626 | 1 |
| sensor | 15.0561687764 | 1.0072533969 | 0.91725481 | 0.1031213443 | 0.3345800225 | 1 |
| switch-config | 5.4619934283 | 0.87269 | 0.595080827 | 0.0842695653 | 0.3003579418 | 1 |
| purchaseorder | 5.1537758621 | 0.9293329064 | 0.7198531005 | 0.0925056323 | 0.3238208211 | 1 |
| router-disc | 6.1542553191 | 0.5717250581 | 0.6433602438 | 0.0751598943 | 0.2993789891 | 1 |
| router-addnet | 5.9212903226 | 1.1046339245 | 0.9229686243 | 0.125300268 | 0.3811461794 | 1 |
| iptel-ethinf | 13.7423524768 | 0.6559110412 | 0.6879577143 | 0.089206846 | 0.2832601614 | 1 |
| iptel-devinfo | 6.5519673779 | 0.7622461896 | 0.8118025957 | 0.1105822689 | 0.302654684 | 1 |

# Appendix C

# XML Document Transformation Process

This research defines **denormalisation** as the process of expanding the structure of XML data in order to meet the requirements of the PO compressor. By increasing the size of the markup language and adding elements, the XML document is transformed into structured data with compact and concise elements which benefits of a unique (highly unlikely to be adopted) pattern. This pattern is needed for the **normalisation** process in order to identify the elements while traversing the tree and construct a new document.

**Attributes - Case 1**

The “foo” element contains two attributes, “bar” and “baz” and the value of the element “ONE”. In the denormalised format, the element “foo” will contain an attributes sequence “a” which will contain a sequence-of attribute “a”. The attribute sequence-of will contain “a” and “v” elements which will be the name and value of the attribute respectively. All the attributes of the “foo” element are contained inside the attributes sequence. The value of “foo” is places inside a “v” element after the attribute sequence.

LISTING C.1: XML Document containing attributes

```
<?xml version="1.0" encoding="UTF-8"?>
<foo bar="Male" baz="1976">foobar</foo>
```

LISTING C.2: XML Document after transformation process

```
<?xml version="1.0" encoding="UTF-8"?>
<foo>
  <a>
```

```
    <a>
      <a>bar</a>
      <v>Male</v>
    </a>
    <a>
      <a>baz</a>
      <v>1976</v>
    </a>
  </a>
  <v>foobar</v>
</foo>
```

### Attributes - Case 2

More advanced scenario, the same rules are applied for nested elements and attributes. NB: “foobar” element attributes are the last child of the transformed “foobar” element. A parent element containing nested elements will have its attributes as the last child.

LISTING C.3: XML Document containing attributes

```
<?xml version="1.0" encoding="UTF-8"?>
<foobar atr0="NO" atr2="NO"  >
  <foo atr1="NO" more="more">ONE</foo>
  <bar atr6="ATR">
    <baz attr="123" attr222="22">TWO</baz>
  </bar>
</foobar>
```

LISTING C.4: XML Document after transformation process

```
<?xml version="1.0" encoding="UTF-8"?>
<foobar>
  <foo>
    <a>
      <a>
        <a>atr1</a>
        <v>NO</v>
      </a>
      <a>
        <a>more</a>
        <v>more</v>
      </a>
    </a>
    <v>ONE</v>
  </foo>
  <bar>
    <baz>
      <a>
        <a>
          <a>attr</a>
          <v>123</v>
        </a>
        <a>
          <a>attr222</a>
          <v>22</v>
        </a>
      </a>
```

```
      <v>TWO</v>
    </baz>
    <a>
      <a>
        <a>atr6</a>
        <v>ATR</v>
      </a>
    </a>
  </bar>
  <a>
    <a>
      <a>atr0</a>
      <v>NO</v>
    </a>
    <a>
      <a>atr2</a>
      <v>NO</v>
    </a>
  </a>
</foobar>
```

### Comments - Case 1

Comments before/after root. A new root will be created to contain both the foo element and the comment transformed into a new element. Similar transformation is applied to PI and DTD components.

LISTING C.5: XML Document containing comments

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- relevant comment-->
<foo>bar</foo>
```

LISTING C.6: XML Document after transformation process

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
    <c>relevant comment</c>
  <foo>bar</foo>
</root>
```

### Comments - Case 2

Comments inside a root node will be transformed into elements of the node.

LISTING C.7: XML Document containing comments

```
<?xml version="1.0" encoding="UTF-8"?>
<foo>
    <!--relevant comment -->
    <bar>bar</bar>
    <!--another comment -->
</foo>
```

LISTING C.8: XML Document after transformation process

```
<?xml version="1.0" encoding="UTF-8"?>
<foo>
```

```
    <c>relevant comment</c>
    <bar>bar</bar>
    <c>another comment</c>
</foo>
```

### Comments - Case 3

Exmple of comments and attributes transformation

LISTING C.9: XML Document containing attributes and comments

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- this is a comment -->
<foo bar="1234" baz="5678">ONE</foo>
<!--another comment -->
```

LISTING C.10: XML Document after transformation process

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <c> this is a comment </c>
  <foo>
    <a>
      <a>
        <a>bar</a>
        <v>1234</v>
      </a>
      <a>
        <a>baz</a>
        <v>5678</v>
      </a>
    </a>
    <v>ONE</v>
  </foo>
  <c>another comment </c>
</root>
```

### Sequence - Case 1

The compressor needs to have sequence-of elements inside a parent. Unordered sequence-of elements will be moved inside parents node “s”.

LISTING C.11: XML Document containing unordered sequence

```
<?xml version="1.0" encoding="UTF-8"?>
<foo>
  <bar>bar</bar>
  <bar>bar</bar>
  <foobar>foobar</foobar>
  <foobar>foobar</foobar>
  <foobar>foobar</foobar>
  <bar>bar</bar>
  <bar>bar</bar>
</foo>
```

LISTING C.12: XML Document after transformation process

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<foo>
  <s>
    <s>
      <bar>bar</bar>
      <bar>bar</bar>
    </s>
    <s>
      <foobar>foobar</foobar>
      <foobar>foobar</foobar>
      <foobar>foobar</foobar>
    </s>
    <s>
      <bar>bar</bar>
      <bar>bar</bar>
    </s>
  </s>
</foo>
```

### Example

An example containing attributes, comments and sequence-of transformation. Comments needs to be reproduced the the exact order.

LISTING C.13: XML Document containing attribute, comments and unordered sequence

```
<?xml version="1.0" encoding="UTF-8"?>
<struct name="foolist">
  <!-- this is a comment -->
  <foo age="12" dob="2012">foo</foo>
  <foo age="20" dob="1999">foo</foo>
  <bar age="24">bar</foo>
</struct>
```

LISTING C.14: XML Document after transformation process

```
<?xml version="1.0" encoding="UTF-8"?>
<struct>
  <c>this is a comment</c>
  <s>
    <s>
      <foo>
        <a
          <a>
            <a>age</a>
            <v>12</v>
          </a>
          <a>
            <a>dob</a>
            <v>2012</v>
          </a>
        </a>
        <v>foo</v>
      </foo>
      <foo>
        <a>
          <a>
            <a>age</a>
            <v>20</v>
          </a>
```

```
          <a>
            <a>dob</a>
            <v>1999</v>
          </a>
        </a>
        <v>foo</v>
      </foo>
    </s>
    <s>
      <bar>
        <a>
          <a>
            <a>age</a>
            <v>20</v>
          </a>
        </a>
        <v>bar</v>
      </bar>
    </s>
  </s>
  <a>
    <a>
      <a>name</a>
      <v>foolist</v>
    </a>
  </a>
</struct>
```

# Appendix D

# Hybrid Model Compression Comparison Results

| Compressor | XML | EXI | GZIP | 7ZIP | HPO |
|---|---|---|---|---|---|
| nation.xml | 5209 | **1006** | 1141 | 1196 | 1142 |
| rdismissal.xml | 32952 | **1366** | 1675 | 1771 | 1869 |
| supplier.xml | 33162 | 6415 | 7087 | 6381 | **6370** |
| rcasegroup.xml | 36846 | **2024** | 2262 | 2109 | 2663 |
| baseball.xml | 74861 | 4666 | 6033 | 4945 | **3803** |
| dates.xml | 91229 | 11124 | 14176 | 10265 | **7442** |
| numeric.xml | 119757 | 9507 | 14096 | 10545 | **7902** |
| reed.xml | 292009 | **12591** | 17513 | 14306 | 17532 |
| customer.xml | 581544 | **78062** | 102168 | 81903 | 89237 |
| part.xml | 716208 | **45573** | 69884 | 57622 | 60839 |
| rand1-1998.xml | 2102492 | 318654 | 377993 | 307880 | **266678** |
| partsupp.xml | 2241854 | 249958 | 312277 | **229225** | 238868 |
| orders.xml | 5378833 | **356866** | 539130 | 404578 | 403668 |
| orderkey.xml | 5406703 | 696124 | 950976 | 717646 | **575414** |
| largenum.xml | 5406703 | 696124 | 950982 | 717658 | **575414** |
| lineitem.xml | 34341531 | **1422940** | 2700373 | 1942771 | 1766707 |

TABLE D.1: Real XML Data Set Compression Results (Bytes)

| Compressor | HPO | EXI | GZIP | 7ZIP |
|---|---|---|---|---|
| nation.xml | 1 | 1.1351888668 | 1.0008764242 | 0.9548494983 |
| rdismissal.xml | 1 | 1.3682284041 | 1.1158208955 | 1.0553359684 |
| supplier.xml | 1 | 0.992985191 | 0.8988288415 | 0.9982761323 |
| rcasegroup.xml | 1 | 1.3157114625 | 1.1772767462 | 1.2626837364 |
| baseball.xml | 1 | 0.8150450064 | 0.6303663186 | 0.7690596562 |
| dates.xml | 1 | 0.6690039554 | 0.5249717833 | 0.7249878227 |
| numeric.xml | 1 | 0.8311770275 | 0.560584563 | 0.7493598862 |
| reed.xml | 1 | 1.3924231594 | 1.0010849084 | 1.2254997903 |
| customer.xml | 1 | 1.1431554405 | 0.8734339519 | 1.0895449495 |
| part.xml | 1 | 1.3349790446 | 0.8705712323 | 1.0558293707 |
| rand1-1998.xml | 1 | 0.836888914 | 0.7055104195 | 0.8661751332 |
| partsupp.xml | 1 | 0.9556325463 | 0.7649234494 | 1.0420678373 |
| orders.xml | 1 | 1.1311472654 | 0.7487396361 | 0.9977507427 |
| orderkey.xml | 1 | 0.8265969856 | 0.60507731 | 0.8018075764 |
| largenum.xml | 1 | 0.8265969856 | 0.6050734925 | 0.8017941694 |
| lineitem.xml | 1 | 1.2415892448 | 0.6542455431 | 0.9093748054 |

TABLE D.2: HPO Compression Ratio

| Compressor | HPO | EXI | GZIP | 7ZIP |
|---|---|---|---|---|
| nation.xml | 0.880910683 | 1 | 0.8816827344 | 0.8411371237 |
| rdismissal.xml | 0.7308721241 | 1 | 0.8155223881 | 0.7713156409 |
| supplier.xml | 1.0070643642 | 1 | 0.9051784958 | 1.0053283184 |
| rcasegroup.xml | 0.760045062 | 1 | 0.8947833775 | 0.9596965386 |
| baseball.xml | 1.226926111 | 1 | 0.7734128957 | 0.9435793731 |
| dates.xml | 1.4947594733 | 1 | 0.7847065463 | 1.083682416 |
| numeric.xml | 1.2031131359 | 1 | 0.6744466515 | 0.9015647226 |
| reed.xml | 0.7181724846 | 1 | 0.7189516359 | 0.8801202293 |
| customer.xml | 0.8747716754 | 1 | 0.7640552815 | 0.9531030609 |
| part.xml | 0.7490754286 | 1 | 0.652123519 | 0.7908958384 |
| rand1-1998.xml | 1.1949017167 | 1 | 0.8430156114 | 1.0349941536 |
| partsupp.xml | 1.0464273155 | 1 | 0.8004367917 | 1.0904482495 |
| orders.xml | 0.8840581864 | 1 | 0.6619294048 | 0.8820697121 |
| orderkey.xml | 1.2097793936 | 1 | 0.7320100612 | 0.9700102836 |
| largenum.xml | 1.2097793936 | 1 | 0.7320054428 | 0.969994064 |
| lineitem.xml | 0.805419348 | 1 | 0.5269420188 | 0.7324280628 |

TABLE D.3: EXI Compression Ratio

| Compressor | HPO | EXI | GZIP | 7ZIP |
|---|---|---|---|---|
| nation.xml | 0.9991243433 | 1.134194831 | 1 | 0.9540133779 |
| rdismissal.xml | 0.8962011771 | 1.2262079063 | 1 | 0.9457933371 |
| supplier.xml | 1.1125588697 | 1.1047544817 | 1 | 1.1106409654 |
| rcasegroup.xml | 0.8494179497 | 1.1175889328 | 1 | 1.0725462304 |
| baseball.xml | 1.5863791743 | 1.2929704243 | 1 | 1.2200202224 |
| dates.xml | 1.9048642838 | 1.2743617404 | 1 | 1.3810034096 |
| numeric.xml | 1.7838521893 | 1.4826969601 | 1 | 1.3367472736 |
| reed.xml | 0.9989162674 | 1.390914145 | 1 | 1.2241716762 |
| customer.xml | 1.1449062609 | 1.308805821 | 1 | 1.2474268342 |
| part.xml | 1.1486710827 | 1.5334518245 | 1 | 1.2128006664 |
| rand1-1998.xml | 1.4174135099 | 1.186217653 | 1 | 1.2277283357 |
| partsupp.xml | 1.307320361 | 1.2493178854 | 1 | 1.3623165013 |
| orders.xml | 1.335577752 | 1.5107351219 | 1 | 1.3325736941 |
| orderkey.xml | 1.6526813738 | 1.3661014417 | 1 | 1.3251324469 |
| largenum.xml | 1.652691801 | 1.3661100609 | 1 | 1.3251186498 |
| lineitem.xml | 1.5284781234 | 1.897741999 | 1 | 1.389959496 |

TABLE D.4: GZIP Compression Ratio

| Compressor | HPO | EXI | GZIP | 7ZIP |
|---|---|---|---|---|
| nation.xml | 1.0472854641 | 1.1888667992 | 1.0482033304 | 1 |
| rdismissal.xml | 0.9475655431 | 1.2964860908 | 1.0573134328 | 1 |
| supplier.xml | 1.0017268446 | 0.9946999221 | 0.9003809793 | 1 |
| rcasegroup.xml | 0.7919639504 | 1.0419960474 | 0.9323607427 | 1 |
| baseball.xml | 1.3002892453 | 1.0597942563 | 0.8196585447 | 1 |
| dates.xml | 1.3793335125 | 0.9227795757 | 0.7241111738 | 1 |
| numeric.xml | 1.3344722855 | 1.1091827075 | 0.748084563 | 1 |
| reed.xml | 0.8159936117 | 1.1362084028 | 0.81687889 | 1 |
| customer.xml | 0.9178143595 | 1.0492044785 | 0.8016502232 | 1 |
| part.xml | 0.9471227338 | 1.2643890023 | 0.8245378055 | 1 |
| rand1-1998.xml | 1.1545009337 | 0.9661890326 | 0.8145124381 | 1 |
| partsupp.xml | 0.9596304235 | 0.9170540651 | 0.7340438137 | 1 |
| orders.xml | 1.0022543278 | 1.1336972421 | 0.7504275407 | 1 |
| orderkey.xml | 1.2471820289 | 1.0309169056 | 0.7546415472 | 1 |
| largenum.xml | 1.2472028835 | 1.0309341439 | 0.7546494045 | 1 |
| lineitem.xml | 1.0996565927 | 1.3653217985 | 0.7194454248 | 1 |

TABLE D.5: 7ZIP Compression Ratio

| Compressor | Canon XML | String Buffer | PDU | ZLIB | Schema |
|---|---|---|---|---|---|
| nation.xml | 65.800 | 34.100 | 6.300 | 70.200 | 23.400 |
| rdismissla.xml | 93.800 | 6.100 | 45.700 | 39.200 | 14.900 |
| supplier.xml | 67.000 | 32.900 | 12.700 | 82.300 | 4.900 |
| rcasegroup.xml | 90.600 | 9.300 | 42.300 | 45.300 | 12.300 |
| baseball.xml | 96.900 | 3.000 | 59.800 | 28.100 | 11.900 |
| dates.xml | 100.000 | 0.000 | 92.700 | 0.400 | 6.700 |
| numeric.xml | 100.000 | 0.000 | 94.500 | 0.400 | 5.000 |
| reed.xml | 87.600 | 12.300 | 44.600 | 53.200 | 2.100 |
| customer.xml | 66.600 | 33.300 | 15.100 | 84.400 | 0.300 |
| part.xml | 75.100 | 24.800 | 32.400 | 66.900 | 0.500 |
| rand1-1998.xml | 92.800 | 7.100 | 55.100 | 44.500 | 0.300 |
| partsupp.xml | 62.200 | 37.700 | 28.600 | 71.200 | 0.100 |
| orders.xml | 81.000 | 18.900 | 39.000 | 60.800 | 0.000 |
| orderkey.xml | 100.000 | 0.000 | 99.900 | 0.000 | 0.000 |
| largenum.xml | 100.000 | 0.000 | 99.900 | 0.000 | 0.000 |
| lineitem.xml | 79.000 | 21.000 | 40.000 | 60.000 | 0.000 |

TABLE D.6: Real XML Data Set Compression Analysis Results (%)

| **Compressor** | **HPO** | **EXI** | **GZIP** | **7ZIP** |
|---|---|---|---|---|
| nation.xml | 0.014 | 0.360 | 0.002 | 0.007 |
| rdismissal.xml | 0.034 | 0.377 | 0.002 | 0.014 |
| supplier.xml | 0.028 | 0.402 | 0.003 | 0.020 |
| rcasegroup.xml | 0.052 | 0.446 | 0.003 | 0.015 |
| baseball.xml | 0.069 | 0.474 | 0.004 | 0.035 |
| dates.xml | 0.072 | 0.471 | 0.005 | 0.035 |
| numeric.xml | 0.091 | 0.657 | 0.010 | 0.048 |
| reed.xml | 0.338 | 0.628 | 0.013 | 0.084 |
| customer.xml | 0.541 | 0.975 | 0.050 | 0.216 |
| part.xml | 1.183 | 0.950 | 0.051 | 0.321 |
| rand11998.xml | 2.684 | 1.384 | 0.115 | 0.780 |
| partsupp.xml | 9.417 | 1.537 | 0.134 | 1.110 |
| orders.xml | 85.844 | 2.053 | 0.428 | 2.583 |
| orderkey.xml | 63.479 | 2.323 | 0.231 | 3.460 |
| largenum.xml | 65.703 | 2.011 | 0.207 | 3.298 |
| rand-8998.xml | 188.353 | 4.985 | 0.849 | 9.357 |
| DeviceInformation.xml | 353.243 | 6.823 | 0.797 | 17.945 |
| lineitem.xml | 723.873 | 5.693 | 2.684 | 18.380 |

TABLE D.7: Real XML Data Set Compression Time Results (Seconds)

| Data Set | XML Size | Compression Rate |
|---|---|---|
| nation.xml | 5209 | 372071.428571429 |
| rdismissla.xml | 32952 | 969176.470588235 |
| supplier.xml | 33162 | 1184357.14285714 |
| rcasegroup.xml | 36846 | 708576.923076923 |
| baseball.xml | 74861 | 1084942.02898551 |
| dates.xml | 91229 | 1267069.44444444 |
| numeric.xml | 119757 | 1316010.98901099 |
| reed.xml | 292009 | 863931.952662722 |
| customer.xml | 581544 | 1074942.6987061 |
| part.xml | 716208 | 605416.737109045 |
| rand1-1998.xml | 2102492 | 783342.771982116 |
| partsupp.xml | 2241854 | 238064.564086227 |
| orders.xml | 5378833 | 62658.2288802945 |
| orderkey.xml | 5406703 | 85173.0966146285 |
| largenum.xml | 5406703 | 82290.0476386162 |
| lineitem.xml | 34341531 | 47696.5708333333 |

TABLE D.8: Real XML Data Set Compression Rate Results (b/s)

# Appendix E

# Published Material

Part of this submission has been published in a number of research papers listed as follows:

1. Moore, J., Kheirkhahzadeh, A., and Bagale, J. (2014). Towards markup-aware text compression. In *Data Compression Conference (DCC), 2014*

2. Kheirkhahzadeh, A., Moore, J., and Bagale, J. (2013). XML-compression techniques for efficient network management. In *5th IEEE International Workshop on Management of Emerging Networks and Services (IEEE MENS 2013)*

3. Moore, J., Kheirkhahzadeh, A., and Bagale, J. (2013b). Domain-Specific XML Compression. In *Data Compression Conference (DCC), 2013*, pages 510–510

Each of these publications is based on parts of the thesis and presents some of the contributions listed in section 1.4 of Chapter 1. Paper [1] presents the preliminary results of the hybrid model evaluating its efficiency against a number of relevant XML compression techniques. In this paper we discuss contribution 5. Paper [2] presents our study on XML compression techniques to improve network management. In this paper we discuss contributions 1 and 3. Paper [3] discusses the idea of compressing highly structured XML documents using a fixed length encoder for high-level basic data types. In this paper we discuss contribution 3 and part of contribution 2 and 4.

# Bibliography

Adiego, J., de la Puente, P., and Navarro, G. (2004). Merging prediction by partial matching with structural contexts model. In *Data Compression Conference, 2004. Proceedings. DCC 2004*, page 522. IEEE.

Adler, M. (2005). Example of proper use of zlib's inflate() and deflate(). [Online] Available at: http://www.zlib.net/zpipe.c [Accessed on 20 January 2013].

Alliance, O. M. (2001). Binary XML Content Format Specification. *OMA Wireless Access Protocol WAP-192-WBXML-20010725-a*.

Álvarez Gutiérrez, D. and Soler, F. O. (2008). Applying lightweight flexible virtual machines to extensible embedded systems. In *Proceedings of the 1st workshop on Isolation and integration in embedded systems*, IIES.

Arion, A., Bonifati, A., Manolescu, I., and Pugliese, A. (2007). XQueC: A query-conscious compressed XML database. *ACM Transactions on Internet Technology (TOIT)*, 7(2):10.

Augeri, C. J., Bulutoglu, D. A., Mullins, B. E., Baldwin, R. O., and III, L. C. B. (2007). An analysis of XML compression efficiency. In *Experimental Computer Science*, page 7. ACM.

Badea, C., Nicolau, A., and Veidenbaum, A. V. (2007). A simplified java bytecode compilation system for resource-constrained embedded processors. In *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '07, pages 218–228, New York, NY, USA. ACM.

Barbosa, D., Mignet, L., and Veltri, P. (2005). Studying the XML Web: gathering statistics from an XML sample. *World Wide Web*, 8(4):413–438.

Bell, M. and Jehanne, A. (2006). WBXML Library. [Online] Available at: https://libwbxml.opensync.org [Accessed on 01 May 2012].

Bex, G. J., Neven, F., and Van den Bussche, J. (2004). DTDs Versus XML Schema: A Practical Study. In *Proceedings of the 7th International Workshop on the Web and Databases: Colocated with ACM SIGMOD/PODS 2004*, WebDB '04, pages 79–84, New York, NY, USA. ACM.

Boncz, P., Grust, T., van Keulen, M., Manegold, S., Rittinger, J., and Teubner, J. (2006). MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 479–490. ACM.

Bournez, C. (2009). Efficient XML Interchange Evaluation - W3C Working Draft. [Online] Available at: http://www.w3.org/TR/exi-evaluation [Accessed on 20 January 2013 ].

Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F. (1998). Extensible markup language (XML). *World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210*.

Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F. (2008). Extensible Markup Language (XML) 1.0 (Fifth Edition). *W3C Recommendation http://www.w3.org/TR/REC-xml/*.

Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., Yergeau, F., and Cowan, J. (2004). Extensible Markup Language (XML) 1.1 - Well-Formed XML Documents. http://www.w3.org/TR/2004/REC-xml11-20040204/#sec-well-formed.

Buneman, P., Choi, B., Fan, W., Hutchison, R., Mann, R., and Viglas, S. (2005). Vectorizing and querying large XML repositories. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 261–272.

Buneman, P., Grohe, M., and Koch, C. (2003). Path queries on compressed XML. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 141–152. VLDB Endowment.

Cannataro, M., Carelli, G., Pugliese, A., and Sacca, D. (2001). Semantic Lossy Compression of XML Data. In *Knowledge Representation Meets Databases (KRDB)*.

Case, J., Fedor, M., Schoffstall, M., and Davin, J. (1990). RFC 1157: Simple network management protocol (SNMP). *IETF, April*.

Castellani, A., Gheda, M., Bui, N., Rossi, M., and Zorzi, M. (2011). Web Services for the Internet of Things through CoAP and EXI. In *Communications Workshops (ICC), 2011 IEEE International Conference on*, pages 1–6.

Cheney, J. (2001). Compressing XML with Multiplexed Hierarchical Models. In Storer, J. A. and Cohn, M., editors, *Proceedings of the 2001 IEEE Data Compression Conference (DCC 2001)*, pages 163–172, Snowbird, UT. IEEE Press.

Cheney, J. (2005). An empirical evaluation of simple DTD-conscious compression techniques. In *Eighth International Workshop on the Web and Databases*. Citeseer.

Cheney, J. (2006a). DTDPPM: DTD-Conscious Compression . [Online] Available at: http://xmlppm.sourceforge.net/dtdppm Accessed on 20 May 2012].

Cheney, J. (2006b). Tradeoffs in XML compression. In *Proceedings of the 2006 IEEE Data Compression Conference (DCC 2006)*, pages 392–401. IEEE Press.

Cheney, J. (2006c). XMLPPM: XML-Conscious PPM Compression. [Online] Available at: http://xmlppm.sourceforge.net [Accessed on 20 May 2012].

Clark, J. (1999). Output - XSL Transformations (XSLT) Version 1.0. [Online] Available at: http://www.w3.org/TR/xslt#output [Accessed on 10 July 2013].

Cleary, J. G. and Witten, I. H. (1984). Data Compression using Adaptive Coding and Partial String Matching. *IEEE Transactions on Communications*, 32(4):396–402.

Colver, B. (2004). XMill: The XML Compressor. [Online] Available at: http://sourceforge.net/projects/xmill [Accessed on 20 May 2012].

Coombs, J. H., Renear, A. H., and DeRose, S. J. (1987). Markup Systems and the Future of Scholarly Text Processing. *Communications of the ACM*, 30(11):933–947.

Corrente, A. and Tura, L. (2004). Security performance analysis of SNMPv3 with respect to SNMPv2c. In *Network Operations and Management Symposium, 2004. NOMS 2004. IEEE/IFIP*, volume 1, pages 729–742 Vol.1.

Delpratt, O. (2009). *Space efficient in-memory representation of XML documents*. PhD thesis, Dept. of Computer Science.

Delpratt, O., Raman, R., and Rahman, N. (2008). Engineering succinct DOM. In *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*, pages 49–60. ACM.

Deutsch, J. G. L. and Gailly, J.-L. (1996). RFC 1950–ZLIB Compressed Data Format Specification version 3.3. *IETF/IESG, May*.

Deutsch, L. P. (1996a). RFC 1951 - DEFLATE compressed data format specification version 1.3. *IETF*.

Deutsch, L. P. (1996b). *RFC 1952 - GZIP file format specification version 4.3*.

Dubuisson, O. (2001). *ASN. 1 communication between heterogeneous systems*. Morgan Kaufmann.

Estrin, D., Govindan, R., Heidemann, J., and Kumar, S. (1999). Next Century Challenges: Scalable Coordination in Sensor Networks. In *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, MobiCom '99, pages 263–270, New York, NY, USA. ACM.

Fablet, Y. and Peintner, D. (2012). Efficient XML Interchange (EXI) profile. [Online] Available at: http://www.w3.org/TR/exi-profile/ [Accessed on 01 July 2012].

Ferragina, P., Luccio, F., Manzini, G., and Muthukrishnan, S. (2006). Compressing and searching XML data via two zips. In *Proceedings of the 15th international conference on World Wide Web*, WWW '06, pages 751–760, New York, NY, USA. ACM.

Frye, R., Wijnen, B., Routhier, S. A., and Levi, D. B. (2003). *RFC3584 - Coexistence between Version 1, Version 2, and Version 3 of the Internet-standard Network Management Framework*.

Gailly, J.-L. and Adler, M. (1999). The gzip home page. *[Online] Available at: http://www. gzip. org/ [Accessed on 20 May 2012]*.

Garrett, C. (2012). EXIProcessor. [Online] Available at: http://sourceforge.net/p/exiprocessor/home/Home/ [Accessed on 10 September 2013].

Girardot, M. and Sundaresan, N. (2000). Millau: an encoding format for efficient representation and exchange of XML over the Web. *Computer Networks*, 33(1):747–765.

Grijzenhout, S. (2010). University of Amsterdam XML Web Collection. [Online] Available at: http://data.politicalmashup.nl/xmlweb/ [Accessed on 10 September 2013].

Grijzenhout, S. and Marx, M. (2013). The quality of the XML web. *Web Semantics: Science, Services and Agents on the World Wide Web*, 19:59–68.

Gudgin, M. (2004). Understanding Infosets. [Online] Available at: http:// msdn.microsoft.com /en-us/library/ aa468561.aspx [Accessed on 01 November 2013].

Harrusi, S., Averbuch, A., and Yehudai, A. (2006). XML syntax conscious compression. In *Data Compression Conference, 2006. DCC 2006. Proceedings*, pages 10 pp.–411.

Hoeller, N., Reinke, C., Neumann, J., Groppe, S., Lipphardt, M., Schuett, B., and Linnemann, V. (2010). Stream-Based XML Template Compression for Wireless Sensor Network Data Management. In *MUE*, pages 1–9. IEEE.

Hoeller, N., Reinke, C., Neumann, J., Groppe, S., Werner, C., and Linnemann, V. (2009). XML data management and XPath evaluation in wireless sensor networks. In *Proceedings of the 7th International Conference on Advances in Mobile Computing and Multimedia*, MoMM '09, pages 218–230, New York, NY, USA. ACM.

Huffman, D. A. et al. (1952). A method for the construction of minimum redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101.

ITU-T (2005). *X.891 : Information technology - Generic applications of ASN.1: Fast infoset*.

ITU-T (2008a). *X.680 : Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation*.

ITU-T (2008b). *X.691 : Information technology - ASN.1 encoding rules: Specification of Packed Encoding Rules (PER)*.

ITU-T (2008c). *X.695 : Information technology - ASN.1 encoding rules: Registration and application of PER encoding instructions*.

Iyer, B. R. and Wilhite, D. (1994). Data compression support in databases. In *VLDB*, volume 94, pages 695–704.

Jaiswal, G. and Mishra, M. (2013). Why use Efficient XML Interchange instead of Fast Infoset. In *Advance Computing Conference (IACC), 2013 IEEE 3rd International*, pages 925–930.

Jehanne, A. (2009). WBXML Library. [Online] Available at: http://sourceforge.net/projects/wbxmllib/ [Accessed on 20 May 2012].

Jelliffe, R. (2006). Schematron specification (ISO/IEC 19757-3), 2006.

Jones, M. T. (2005). Optimization in GCC. *Linux Journal*, Issue #131 [Online] Available at: http://www.linuxjournal.com/article/7269?page=0,0 [Accessed on 10 January 2012].

Kamiya, T. and Bournez, C. (2012). Efficient XML Interchange Working Group. [Online] Available at: http://www.w3.org/XML/EXI/ [Accessed on 01 July 2013].

Kay, M. (2012). XSL Transformations (XSLT) Version 3.0 - W3C Working Draft. [Online] Available at: http://www.w3.org/TR/xslt-21 [Accessed on 15 May 2013].

Kheirkhahzadeh, A., Moore, J., and Bagale, J. (2013). XML-compression techniques for efficient network management. In *5th IEEE International Workshop on Management of Emerging Networks and Services (IEEE MENS 2013)*.

Larmouth, J. (2000). *ASN. 1 complete*. Morgan Kaufmann.

League, C. and Eng, K. (2007a). Schema-based compression of XML data with relax NG. *Journal of Computers*, 2(10):9–17.

League, C. and Eng, K. (2007b). Type-based compression of xml data. In *Data Compression Conference, 2007. DCC'07*, pages 273–282. IEEE.

Lelewer, D. A. and Hirschberg, D. S. (1987). Data Compression. *ACM Computing Surveys (CSUR)*, 19(3):261–296.

Levene, M. and Wood, P. (2002). XML structure compression. In *Proceedings of the Second International Workshop on Web Dynamics*, pages 1–14.

Li, W. (2003). *Xcomp: An XML compression tool*. PhD thesis, School of Computer Science, University of Waterloo.

Liefke, H. and Suciu, D. (2000). XMill: An Efficient Compressor for XML Data. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, pages 153–164, New York, NY, USA. ACM.

Lifton, J., Feldmeier, M., Ono, Y., Lewis, C., and Paradiso, J. A. (2007). A platform for ubiquitous sensor deployment in occupational and domestic environments. In *Proceedings of the 6th international conference on Information processing in sensor networks*, IPSN '07, pages 119–127, New York, NY, USA. ACM.

Marrón, P. J., Lachenmann, A., Minder, D., Gauger, M., Saukh, O., and Rothermel, K. (2005). Management and configuration issues for sensor networks. *International Journal of Network Management*, 15(4):235–253.

Martin, B. and Jano, B. (1999). WAP binary XML content format. W3C note. *World Wide Web Consortium (June). Cambridge, MA*.

McDowell, A., Schmidt, C., and Yue, K.-b. (2004). Analysis and Metrics of XML Schema. In *Software Engineering Research and Practice*, pages 538–544.

Megginson, D. et al. (2001). Sax 2.0: The simple api for xml. *SAX project*.

Miklau, G. (2014). XML Data Repository - Datasets, Details, and Download. [Online] Available at: http://www. cs.washington.edu/ research/xmldatasets /www/repository.html [Accessed on 10 September 2013].

Min, J.-K., Park, M.-J., and Chung, C.-W. (2003). XPRESS: a queriable compression for XML data. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 122–133. ACM.

Mlynkova, I., Toman, K., and Pokorny, J. (2006). Statistical Analysis of Real XML Data Collections. In *In COMAD'06: Proceedings of the 13th International Conference on Management of Data*, pages 20–31. Tata McGraw-Hill Publishing Company Limited.

Montenegro, G., Kushalnager, N., Hui, J., et al. (2007). IETF - RFC 4944 Transmission of IPv6 packets over IEEE 802.15. 4 networks. *Internet proposed standard RFC*, 802(15.4).

Moore, J. (2009). Get stuffed: Tightly packed abstract protocols in Scheme. The $10^{th}$ Scheme and Functional Programming Workshop.

Moore, J. (2010a). Everything counts in small amounts. International Workshop on Dynamic languages for Robotic and Sensor systems (DYROS).

Moore, J. (2011). Executable Rules of Encoding. In *5th Workshop on Dynamic Languages and Applications*.

Moore, J. (2012). Packedobjects Reference Manual. [Online] Available at: http://zedstar.org/packedobjects/ [Accessed on 30 January 2013].

Moore, J., Bagale, J., and Kheirkhahzadeh, A. (2013a). Teaching Networking Fundamentals with Sound. In *Advanced Learning Technologies (ICALT), 2013 IEEE 13th International Conference on*, pages 369–370.

Moore, J., Bagale, J., Kheirkhahzadeh, A., and Komisarczuk, P. (2012). Fingerprinting Seismic Activity across an Internet of Things. In *New Technologies, Mobility and Security (NTMS), 2012 5th International Conference on*, pages 1–6.

Moore, J., Collins, T., and Shrestha, S. (2010). An Open Architecture for Detecting Earthquakes Using Mobile Devices. In *Communications and Mobile Computing (CMC), 2010 International Conference on*, volume 1, pages 437–441.

Moore, J., Kheirkhahzadeh, A., and Bagale, J. (2013b). Domain-Specific XML Compression. In *Data Compression Conference (DCC), 2013*, pages 510–510.

Moore, J., Kheirkhahzadeh, A., and Bagale, J. (2014). Towards markup-aware text compression. In *Data Compression Conference (DCC), 2014*.

Moore, J. P. (2010b). A dynamic data encoder for embedded systems. *White paper*.

Muldner, T., Corbin, T., Fry, C., et al. (2012). Design and Implementation of an Online XML Compressor for Large XML Files. *International Journal On Advances in Internet Technology*, 5(3 and 4):141–161.

Müldner, T., Leighton, G., and Diamond, J. (2005). Using XML compression for WWW communication. In *Proceedings of the IADIS WWW/Internet 2005 Conference*.

Ng, W., Lam, W.-Y., and Cheng, J. (2006a). Comparative analysis of XML compression technologies. *World Wide Web*, 9(1):5–33.

Ng, W., Lam, W.-Y., Wood, P. T., and Levene, M. (2006b). XCQ: A queriable XML compression system. *Knowledge and Information Systems*, 10(4):421–452.

Nicol, G., Wood, L., Champion, M., and Byrne, S. (2001). Document object model (DOM) level 3 core specification.

Pak, J. and Park, K. (2012). Efficient message encoding method for personal health device monitoring system. In *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*, page 19. ACM.

Pavlov, I. (2015). 7z Format. [Online] Available at: http://www.7-zip.org/7z.html[Accessed on 28 March 2015].

Peintner, D. (2012). EXIficient - XML becomes efficient. [Online] Available at: http://sourceforge.net/p/exiprocessor/home/Home/ [Accessed on 20 July 2012].

Pizlo, F., Ziarek, L., Blanton, E., Maj, P., and Vitek, J. (2010). High-level programming of embedded hard real-time devices. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 69–82, New York, NY, USA. ACM.

Robie, J., Chamberlin, D., Dyck, M., and Snelson, J. (2013). XML Path Language (XPath) 3.0. [Online] Available at: www.w3.org/TR/xpath-30/ [Accessed on 01 March 2014].

Sakr, S. (2008). An Experimental Investigation of XML Compression Tools. *The Computing Research Repository (CoRR)*, abs/0806.0075.

Sakr, S. (2009). XML compression techniques: A survey and comparison. *Journal of Computer and System Sciences*, 75(5):303–322.

Sakr, S. (2011). Investigate state-of-the-art XML compression techniques. Technical report, National ICT Australia, IBM.

Sandoz, P., Triglia, A., and Pericas-Geertsen, S. (2004). Fast Infoset. http://www.oracle.com /technetwork/articles/java/ fastinfoset-139262.html.

Schalnat, G. E., Randers-Pehrson, G., et al. (2002). libpng-Portable Network Graphics (PNG) Reference Library, 2002.

Schneider, J. and Kamiya, T. (2011). Efficient XML Interchange (EXI) Format 1.0. [Online] Available at: http://www.w3.org/TR/exi/ [Accessed on 01 October 2012].

Seward, J. (2000). The bzip2 and libbzip2 official home page. [Online] Available at: http://sources. redhat. com/bzip2 [Accessed on 01 February 2014].

Shannon, C. E. (1948). A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1):3–55.

Shelby, Z. (2010). Embedded web services. *Wireless Communications, IEEE*, 17(6):52–57.

Shin, D. and Shim, C. (2005). XNMP - an XML based network management protocol over VoIP. In *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, 2005 and First ACIS International Workshop on Self-Assembling Wireless Networks. SNPD/SAWN 2005. Sixth International Conference on*, pages 208–213.

Skibinski, P. and Swacha, J. (2007). Combining Efficient XML Compression with Query Processing. In Ioannidis, Y. E., Novikov, B., and Rachev, B., editors, *ADBIS*, volume 4690 of *Lecture Notes in Computer Science*, pages 330–342. Springer.

Sperberg-McQueen, C. M. and Thompson, H. (2000). XML Schema. *W3C Recommendation [Online] Available at: http://www.w3.org/XML/Schema [Accessed on 20 January 2013]*.

Steedman, D. (1993). *Abstract Syntax Notation One (ASN.1): The Tutorial and Reference*. Technology Appraisals, Twickenham, UK.

Subramanian, H. and Shankar, P. (2006). Compressing XML documents using recursive finite state automata. In *Implementation and Application of Automata*, pages 282–293. Springer.

Thompson, H. S., Beech, D., Maloney, M., and Mendelsohn, N. (2004). XML Schema Part 1: Structures Second Edition - Constraints and Validation Rules. [Online] Available at: http://www.w3.org/TR/xmlschema-1/#concepts-schemaConstraints [Accessed on 20 February 2014].

Thompson, H. S., Tebbutt, J., and Cincotta, T. (2011). W3C XML Schema Test Collection. [Online] Available at: http://www. w3.org/XML/ 2004/xml-schema-test-suite/ [Accessed on 01 July 2012].

Tolani, P. and Haritsa, J. (2002). XGrind: a query-friendly XML compressor. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 225–234.

Toman, V. et al. (2004). Syntactical compression of XML data. In *Proc. Int'l Conf. on Advanced Information Systems Engineering CAiSE'04*.

Varda, K. (2011). Protocol buffers: Google's data interchange format. [Online] Available at: https://code.google.com/p/protobuf [Accessed on 25 March 2014].

Wade, G. (1994). *Signal coding and processing*. Cambridge university press.

Wang, F., Li, J., and Homayounfar, H. (2007). A space efficient XML DOM parser. *Data & Knowledge Engineering*, 60(1):185–207.

Werner, C. and Buschmann, C. (2004). Compressing SOAP messages by using differential encoding. In *Web Services, 2004. Proceedings. IEEE International Conference on*, pages 540–547.

White, G., Brutzman, D., and Williams, S. (2006). Efficient XML Interchange Measurements Note - W3C Working Draft. [Online] Available at: http://www.w3.org/TR/2006/WD-exi-measurements-20060718 [Accessed on 20 September 2013].

Winkler, M., Tuchs, K., Hughes, K., and Barclay, G. (2008). Theoretical and practical aspects of military wireless sensor networks. *Journal of Telecommunications and Information Technology*, 2:37–45.

Yoon, J.-H., Ju, H.-T., and Hong, J. W. (2003). Development of SNMP-XML translator and gateway for XML-based integrated network management. *International Journal of Network Management*, 13(4):259–276.

Zhang, N., Kacholia, V., and Ozsu, M. T. (2004). A succinct physical storage scheme for efficient evaluation of path queries in XML. In *Data Engineering, 2004. Proceedings. 20th International Conference on*, pages 54–65. IEEE.

Ziv, J. and Lempel, A. (1977). A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343.